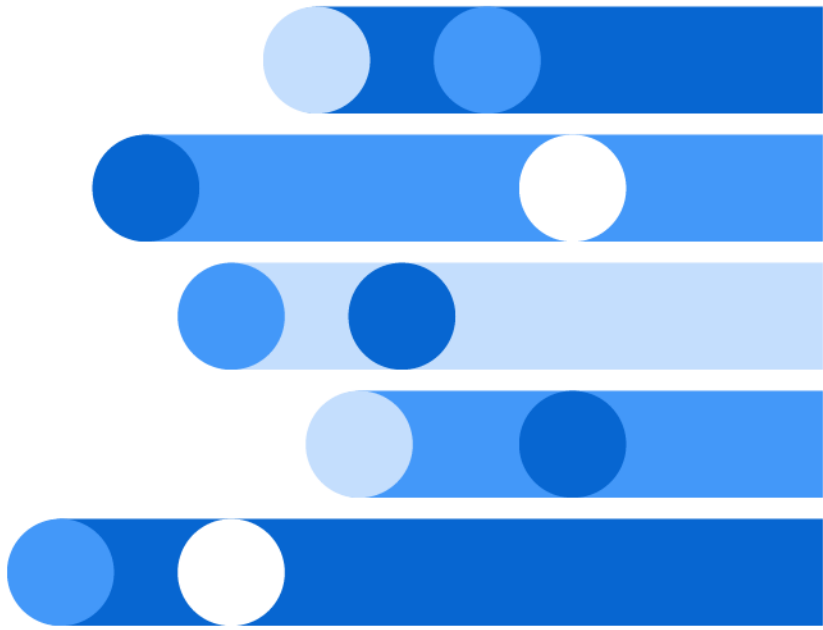
SAS® 9.4 DS2 Language Reference, Sixth Edition



SAS® Documentation
September 11, 2026

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® 9.4 DS2 Language Reference, Sixth Edition*. Cary, NC: SAS Institute Inc.

SAS® 9.4 DS2 Language Reference, Sixth Edition

Copyright © 2016, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2026

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

9.4-P12:ds2ref

Contents

What's New in SAS 9.4 DS2 Language Reference . . . . . *vii*

PART 1 Introduction 1

Chapter 1 / Introduction to the DS2 Language **3**
Introduction to the DS2 Language 3
Running DS2 Programs 4
Supported Data Sources 5
Intended Audience 6
When to Use DS2 7
Converting DATA Step Programs to DS2 Programs 7
Syntax Conventions for the DS2 Language 8

PART 2 Getting Started

Chapter 2 / DS2 and the DATA Step **13**
What Is DS2? 13
Similarities between DS2 and the DATA Step 13
Differences between DS2 and the DATA Step 14

Chapter 3 / Learning by Example: Using the Sample Programs **19**
About the Getting Started Sample Programs 19
Your First Sample Programs 21
See Also 30

Chapter 4 / Building Blocks of DS2 Programs **33**
Basic DS2 Language Concepts 33
What Is a DS2 Program? 40
Example: Block Scope 40
Example: A Simple Thread Program 42

Chapter 5 / Understanding DS2 Methods and Packages **45**
Modularity, Encapsulation, and Abstraction in DS2 45
Getting Started with DS2 Methods 46
Getting Started with DS2 Packages 49
Example: Introduction to DS2 Methods 51
Example: Introduction to DS2 Packages 53

PART 3 DS2 Language Reference

Chapter 6 / DS2 Formats **57**
Overview of Formats 59
General Format Syntax 59
Using Formats in DS2 60
Validation of DS2 Formats 62
DS2 Format Examples 62
Format Categories 62
Dictionary 75

Chapter 7 / DS2 Functions **229**
Overview of DS2 Functions 237
General Function Syntax 238
Using Functions 239
DS2 Function Examples 244
Function Categories 244
Dictionary 289

Chapter 8 / DS2 Informats **1195**
Overview of Informats 1195
General Informat Syntax 1195
How Informats Are Used in DS2 1196
How to Specify Informats in DS2 1197
Validation of DS2 Informats 1197
DS2 Informat Examples 1197

Chapter 9 / DS2 Operators **1199**
Dictionary 1199

Chapter 10 / DS2 Statements **1203**
Overview of Statements 1204
Block Statements 1205
Global Declaration Statements 1207
Local Statements 1208
Including Comments in a DS2 Program 1209
Dictionary 1210

Chapter 11 / DS2 System Methods **1333**
Overview of System Methods 1333
Dictionary 1335

Chapter 12 / DS2 System Options **1343**
Overview of System Options 1343
Dictionary 1344

Chapter 13 / DS2 Table Options **1347**
Overview of Table Options 1349
Using Table Options in DS2 1350
How to Specify Table Options in DS2 1351
DS2 Table Option Examples 1351

DS2 Statement Table Options by Data Source . . . 1352
Dictionary . . . 1364

Chapter 14 / DS2 FCMP Package Methods, Operators, and Statements . . . **1495**
Dictionary . . . 1495

Chapter 15 / DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements . . . **1503**
Dictionary . . . 1504

Chapter 16 / DS2 HTTP Package Methods, Operators, and Statements . . . **1587**
Method Naming Convention . . . 1588
Dictionary . . . 1588

Chapter 17 / DS2 JSON Package Methods, Operators, and Statements . . . **1631**
Dictionary . . . 1632

Chapter 18 / DS2 Logger Package Methods, Operators, and Statements . . . **1665**
Dictionary . . . 1665

Chapter 19 / DS2 Matrix Package Methods, Operators, and Statements . . . **1675**
Dictionary . . . 1676

Chapter 20 / DS2 PCRXFIND Package Methods, Operators, and Statements . . . **1753**
Dictionary . . . 1753

Chapter 21 / DS2 PCRXREPLACE Package Methods, Operators, and Statements . . . **1771**
Dictionary . . . 1771

Chapter 22 / DS2 SQLSTMT Package Methods, Operators, and Statements . . . **1779**
Dictionary . . . 1780

Chapter 23 / DS2 TZ Package Methods, Operators, and Statements . . . **1849**
Dictionary . . . 1849

PART 4 Appendixes

Appendix 1 / Data Type Reference . . . **1867**
Data Types for SAS Data Sets . . . 1868
Data Types for SPD Engine Data Sets . . . 1869
Data Types for SPD Server Tables . . . 1871
Data Types for CAS . . . 1873
Data Types for Amazon Redshift . . . 1875
Data Types for Aster . . . 1877
Data Types for DB2 under UNIX and PC Hosts . . . 1879
Data Types for Google BigQuery . . . 1881
Data Types for Greenplum . . . 1883
Data Types for HAWQ . . . 1884
Data Types for HDMD . . . 1885
Data Types for Hive . . . 1887

Data Types for Impala ... 1890
Data Types for JDBC ... 1891
Data Types for MDS ... 1893
Data Types for Microsoft SQL Server ... 1895
Data Types for MongoDB ... 1898
Data Types for MySQL ... 1899
Data Types for Netezza ... 1901
Data Types for ODBC ... 1903
Data Types for Oracle ... 1905
Data Types for PostgreSQL ... 1907
Data Types for Reading from Salesforce ... 1909
Data Types for Writing to Salesforce ... 1912
Data Types for SAP ... 1913
Data Types for SAP HANA ... 1915
Data Types for SAP IQ ... 1917
Data Types for Snowflake ... 1919
Data Types for Spark ... 1921
Data Types for Teradata ... 1923
Data Types for Vertica ... 1925
Data Types for Yellowbrick ... 1928

Appendix 2 / DS2 Expressions ... **1931**
What Is an Expression? ... 1931
Types of Expressions ... 1932
Operators in Expressions ... 1949

Appendix 3 / DS2 Example Programs ... **1955**
Example: Find Minimums ... 1957
Example: SQL in a DS2 Program ... 1959
Example: Make Two New Tables Based on a Condition ... 1961
Example: Change Case of Text Output ... 1963
Example: Scope ... 1964
Example: Functions ... 1966
Example: Arrays ... 1967
Example: SELECT Statement ... 1972
Example: GOTO and LEAVE Statements with Labels ... 1973
Example: Overloaded Methods ... 1976
Example: Analyze a Table Using Multiple Threads ... 1978
Example: Using Four Threads to Compute Summary Statistics ... 1980
Example: Data Cleaning ... 1982
Example: SUBSTR Function ... 1984
Example: PUT with SAS Formats ... 1985
Example: Generate Statistics from Table Data ... 1986
Example: Matrices and Non-linear Equations ... 1988
Example: DS2 Matrices and Regression Analysis ... 1992
Example: Use the SQLSTMT Package in Another Package ... 1995
Example: Update the Values of a Table By Using Two Databases ... 1997
Example: Store FedSQL Statements in a Hash Package ... 1998
Example: Run a DS2 Program from z/OS Batch ... 2004
Example: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File ... 2005
Example: Reading JSON Text from HTTP and Creating a SAS Data Set ... 2009

What's New in SAS 9.4 DS2 Language Reference

Overview

DS2 is a SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with Base SAS and SAS Viya and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages.

Several DS2 language elements accept embedded FedSQL syntax, and the run-time-generated queries can exchange data interactively between DS2 and any supported database. This action enables SQL preprocessing of input tables, which effectively combines the power of the two languages.

The DS2 procedure enables you to submit DS2 language statements from a Base SAS session. The procedure enables the requests to be processed by the DS2 data access technology that supports a scalable, threaded, high-performance, and standards-based way to access, manage, and share relational data. For more information about PROC DS2, see *Base SAS Procedures Guide*.

If you have SAS Viya, you can also submit DS2 language statements in CAS. For more information, see "Running DS2 Programs in CAS" in *SAS DS2 Programmer's Guide*. You can use the DS2 procedure or DS2 action set to submit DS2 statements to the CAS server. For more information about the DS2 action set, see *SAS Viya: System Programming Guide*.

In SAS 9.4M7, the following features are changed or enhanced: "DS2 Functions" on page x and "General Enhancements" on page xv.

In SAS Viya 3.5, the following features are changed or enhanced:

- "DS2 Functions" on page x
- "Predefined DS2 Packages" on page xii
- "General Enhancements" on page xv

Other enhancements:

September 2024: "General Enhancements" on page xv.

August 2021: "General Enhancements" on page xv.

July 2021: "SAS In-Database Code Accelerator" on page viii

April 2021: "General Enhancements" on page xv, "DS2 HTTP Package" on page xiii

February 2021: "DS2 Loggers and Logger Packages" on page xiv

Note: Changes and enhancements for previous SAS 9.4 and SAS Viya releases are detailed in the following sections.

SAS In-Database Code Accelerator

In SAS 9.4M7 (July 2021 release, or earlier if you apply a hot fix), you can run SAS Embedded Process for Spark jobs by using the Apache Livy REST service. The June 2021 release of the SAS Embedded Process for Hadoop must be installed. For details, see HADOOPPLATFORM= System Option.

In SAS 9.4M6, the SAS In-Database Code Accelerator can be executed as either a MapReduce job or as a Spark job. A new system option, HADOOPPLATFORM, determines which execution platform is used. However, in SAS 9.4M6, the HADOOPPLATFORM=SPARK option is not supported on the Windows operating system for the SAS In-Database Code Accelerator.

SAS 9.4M4 has the following changes and enhancements:

- Three new special automatic variables enable you to subset a problem across a DS2 thread.
  - _HOSTNAME_ returns the name of the worker nodes on which the DS2 program is running.
  - _NTHREADS_ returns the total number of threads running in the program.
  - _THREADID_ can be used in a thread program to identify individual threads.

SAS 9.4M3 has the following changes and enhancements:

- A SET statement with embedded SQL and a SET statement that specifies multiple input tables are now supported.

  Note: When using the SAS In-Database Code Accelerator for Hadoop, SET statements with multiple input tables requires Hive .13 or later.

- The new DS2 MERGE statement is supported.
- The SAS In-Database Code Accelerator for Hadoop supports reading and writing of HDFS-SPD Engine file formats.

- In addition, if a Hadoop data or thread program fails and MSGLEVEL=I is set, a message is written to the SAS log that contains a link to the MapReduce job log where you can find the error messages.

In the February 2015 release of the SAS In-Database Code Accelerator for Hadoop, the following changes and additions were made:

- The SAS In-Database Code Accelerator for Hadoop supports only Cloudera 5.2 and Hortonworks 2.1 or later.
- The SAS In-Database Code Accelerator for Hadoop uses HCatalog to process complex, non-delimited files.
- The SAS In-Database Code Accelerator for Hadoop now supports Avro, ORC, RCFile, and Parquet file types.
- For the SAS In-Database Code Accelerator for Hadoop, you can use the DBCREATE_TABLE_OPTS table option to specify the output SerDe, the output delimiter of the Hive table, the output escaped by, and any other CREATE TABLE syntax allowed by Hive.

For SAS 9.4M2, in-database processing for Hadoop has been enhanced by the addition of the SAS In-Database Code Accelerator for Hadoop. The SAS In-Database Code Accelerator for Hadoop runs the DS2 data program as well as the thread program inside the database.

SAS 9.4M2 has the following enhancements and changes to the In-Database Code Accelerator:

- A new system option, DS2ACCEL controls whether the DS2 code is executed inside the database. The default value is NONE, which prevents DS2 code from executing inside the database.
- The PROC DS2 INDB option has changed its name to DS2ACCEL. The INDB option is still supported. However, the default value for this option has changed from YES to NO, which prevents DS2 code from executing in the database. This is a change in behavior from the 9.4 release.
- The SAS In-Database Code Accelerator for Teradata now runs the DS2 data program as well as the thread program inside the database.

DS2 Formats

In SAS 9.4M6, the new $UUID format writes character data to the universally unique identifier (UUID) format.

In SAS Viya 3.4, the new $UUID format writes character data in the universally unique identifier (UUID) format.

SAS 9.4M5 has the following changes and enhancements:

- The following formats have been added:

B8601DA B8601TM E8601DZ
B8601DN B8601TZ E8601LZ
B8601DT E8601DA E8601TM
B8601DZ E8601DN E8601TZ
B8601LZ E8601DT

- When using input-width aware formats such as $OCTAL and $HEX in the PUT function, the function now uses the width of the input field when no width is specified in the format.

DS2 Functions

SAS 9.4M7 and SAS Viya 3.5 have the following changes and enhancements:

- The following functions have been added:

  COMPLEV
  COMPCOST
  COMPGED

- The COMPRESS function now has an optional third argument, *modifier*, that specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COMPRESS function.

SAS 9.4M6 has the following changes and enhancements:

- The following functions have been added:

  CMISS SAVING SYSGET
  LOGISTIC SHA256

- The SCAN function now supports a modifier.
- To align the SUBSTR (right of =) function with DATA step behavior, a length of 0 is now considered invalid.

In SAS Viya 3.4, the new SYSGET function returns the value of the specified operating environment variable on the machine on which the program is running.

SAS Viya 3.3 has the following changes and enhancements:

- The following functions have been added:

  CMISS LOGISTIC SAVING

- The SCAN function now supports a modifier. The modifier is supported only on the CAS server.
- To align the SUBSTR (right of =) function with DATA step behavior, a length of 0 is now considered invalid.

SAS 9.4M5 has the following changes and enhancements:

- The following functions have been added:

AIRY	FNONCT	LOGSDF
CDF	IBESSEL	PDF
CNONCT	JBESSEL	QUANTILE
COT	LCOMB	SDF
CSC	LFACT	SQUANTILE
DAIRY	LOGCDF	TNONCT
FINANCE	LOGPDF	

- The following functions are no longer supported. Use the associated RAND function instead.

RANBIN	RANNOR	RANTRI
RANCAU	RANPOI	RANUNI
RANGAM	RANTBL	UNIFORM

- The following functions have been deprecated. It is recommended that you use the new PCRXFIND and PCRXREPLACE packages for regular expression matching and substitution.

PRXCHANGE	PRXPARSE
PRXMATCH	PRXPOSEN

- The LENGTHM function no longer accepts expressions or literal arguments. If an expression or literal argument is used, a null byte length is returned and a warning issued that the argument is invalid. The LENGTHM function accepts only variable arguments.

SAS 9.4M4 has the following changes and enhancements:

- Two new functions, DIF and LAG, enable you to access previous values of a variable or expression. These functions are useful for computing lags and differences of series.
- The new INTNEST function calculates the number of whole periods of the smaller interval that fits into the period of the larger interval.

SAS 9.4M3 has the following changes and enhancements:

- Two new functions, CMP and CMPT, enable you to compare two character strings including and excluding trailing blanks, respectively.
- The FMTINFO function returns information about a SAS format.
- The SHA256HEX and SHA256MACHEX functions convert a string to a 256-bit hash value based on the SHA256 algorithm and the Hash-based Message Authentication (HMAC) algorithm, respectively.

SAS 9.4M2 has the following changes and enhancements:

- Three new Perl regular expression (PRX) functions are available:
  - The PRXCHANGE function performs a pattern-matching replacement.
  - The PRXPARSE function compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.

  - The PRXPOSN function returns a character string that contains the value for a capture buffer.
- Five new DBCS functions are available:
  - The KCOUNT function returns the number of double-byte characters in an expression.
  - The KSTRCAT function concatenates two or more character expressions.
  - The KSTRIP function removes leading and trailing blanks from a character string.
  - The KUDPATE function inserts, deletes, and replaces character value contents.
  - The KUPDATES function inserts, deletes, and replaces the contents of the character value according to the byte position of the character value in the argument.

SAS 9.4M1 has the following changes and enhancements:

- The MISSING function also now supports all data types and package parameters.
- The new UUIDGEN function returns the short form of a Universally Unique Identifier (UUID).
- The MD5 function now supports Unicode character strings.

Predefined DS2 Packages

All Predefined DS2 Packages

In SAS Viya 3.5, the documentation about the scope of package instances has been updated. The THIS and *package-instance* arguments of the _NEW_ operator have been deprecated and removed from the _NEW_ operator for packages. The lifetime of a package instance now depends on whether a DS2 package variable references it. The lifetime of a package instance can extend beyond the scope in which the instance itself was created.

DS2 FCMP Package

In SAS Viya 3.5, the maximum number of threads you can specify on the SET FROM statement is 100 if you are using an FCMP package.

DS2 HTTP Package

In April 2021, the documentation that describes the HTTP package methods has been enhanced. The modifications clarify support for the new methods and summarize the features that are available with the newer methods, as noted below.

Starting with SAS 9.4M6 and SAS Viya 3.4, some new methods are added and some existing methods are modified to enable you to perform the following tasks:

- Set a URL or proxy URL as well as a user name and password for those URLs as the default URLs and credentials for one or more HTTP requests. The new methods are SETURL, SETUSERNAME, SETPASSWORD, SETPROXYURL, SETPROXYUSERNAME, and SETPROXYPASSWORD. The modified methods are CREATEGETMETHOD, CREATEHEADMETHOD, and CREATEPOSTMETHOD.
- Specify an Open Authorization (OAuth) token or search for one in the SAS environment (new SETOAUTHTOKEN and ADDOAUTHTOKEN methods).
- If the character set for the HTTP request and response body is not set, set the default character set using the new SETREQUESTBODYCHARACTERSET and SETRESPONSEBODYCHARACTERSET methods.
- If the content type is not set, the existing SETREQUESTBODYASSTRING method sets the default charset value to ISO-8859-1 (latin1) as specified by the 1.1 protocol.

In SAS Viya 3.3, if the content type is not set, the SETREQUESTBODYASSTRING method sets the default charset value to ISO-8859-1 (latin1) as specified by the 1.1 protocol.

SAS 9.4M2 has the following changes and enhancements:

- The DS2 HTTP predefined package is available and enables you to construct a client to access web services.
- A new logger, App.TableServices.d2pkg, is available that supports logging of traffic through the SAS logging facility.

DS2 JSON Package

In SAS 9.4M3, a new DS2 package, JSON, enables you to create and parse JSON text.

DS2 Loggers and Logger Packages

Starting in February 2021 in SAS Viya 3.5 (with a hot fix), a new logger is available that enables you to trace changes to variable values during DS2 program execution. The App.TableServices.DS2.Runtime.TraceVariables logger is used with a new DS_OPTIONS statement option: TRACEVARIABLES. For more information, see "App.TableServices.DS2.Runtime.TraceVariables" in *SAS DS2 Programmer's Guide* and "DS2_OPTIONS Statement" on page 1247.

SAS 9.4M1 has the following changes and enhancements:

- Five new loggers are available:

  App.TableServices.DS2.Config.Options
  shows the options that are supplied to the DS2 compiler.

  App.TableServices.DS2.Config.Source
  shows the DS2 source code that is processed by the DS2 compiler.

  App.TableServices.DS2.Config.Version
  shows version information for all threaded kernel extensions that are loaded by the DS2 compiler.

  App.TableServices.DS2.Runtime.Calls
  shows a trace of all method calls during execution.

  App.TableServices.DS2.Runtime.SQL
  shows all SQL statements that are either prepared by the DS2 compiler, executed by the DS2 compiler, or both.

- The following enhancements have been made to the LOG method:
  - The maximum length of a message is now 65535 characters.
  - Two new arguments have been added to specify formatted messages.

DS2 PCRXFIND and PCRXREPLACE Packages

In SAS 9.4M5, two new predefined packages are available, PCRXFIND and PCRXREPLACE, for regular expression matching and substitution. These packages are based on the PCRE 2 open-source regular expression library. The DS2 PRX functions have been deprecated because PCRXFIND and PCRXREPLACE provide superior performance to the existing PRX functions.

DS2 SQLSTMT Package

SAS 9.4M3 has the following changes and enhancements:

- Three new methods are available:
  - GETCOLUMNCOUNT returns the number of columns in the result set.
  - GETCOLUMNNAME returns the column name of the result set column with the designated index.
  - GETCOLUMNTYPENAME returns the type name of the result set column with the designated index.
- New constructor syntax and two new methods enable you to create an SQLSTMT package instance and prepare it at a later time.

In SAS 9.4M2, a connection string parameter is available when declaring and instantiating an SQLSTMT package.

DS2 TZ Package

In SAS 9.4M3, a new DS2 package, TZ, enables you to perform time zone processing on date and time data.

General Enhancements

SAS 9.4M9 has the following new features and enhancements:

- Support for the BULKLOAD= table option is available for Amazon Redshift. For more information, see "BULKLOAD= Table Option" on page 1400.
- Support for the BULKLOAD= table option is available for Microsoft SQL Server. Use this option to enable bulk loading to Microsoft SQL Server.For more information, see "BULKLOAD= Table Option" on page 1400.
- Support for the FETCH_NUMERIC_TYPE= table option for Google BigQuery was extended to SAS 9.4M8 and SAS9.4M9 with the SAS 9.4M9 release.
- A new table option, READMODE=, is available for use with Google BigQuery. Use this table option to control whether the Google Storage API is used to retrieve data into SAS.For more information, see "READMODE= Table Option" on page 1451.
- Support for the BULKLOAD= table option is available for Snowflake. For more information, see "DS2 Statement Table Options by Data Source" on page 1352.

The Google BigQuery NUMERIC and BIGNUMERIC data types are supported for reading by DS2, when appropriate SAS/ACCESS software is installed. Starting with the September 2024 update to SAS/ACCESS Interface to Google BigQuery on SAS 9.4M7, in order to improve read performance, these Google BigQuery data types are handled as DOUBLEs by default. The FETCH_NUMERIC_TYPE= table option enables you to revert to NUMERIC handling. For more information, see "Data Types

for Google BigQuery" on page 1881 and "FETCH_NUMERIC_TYPE= Table Option" on page 1424.

Starting in August 2021 in SAS Viya 3.5, most log messages that are generated for DS2 programs that run in CAS have log line numbers in them. Line numbers are now reported in execution-time messages. Program compilation messages that previously did not report line numbers also now include line numbers. The generation of the log line numbers can affect performance. A new option on the DS_OPTIONS statement, REPORTLINE=, is provided to suppress the generation of the log line numbers, if needed. For more information, see documentation for the REPORTLINE= option in the "DS2_OPTIONS Statement" on page 1247. This enhancement is provided as a hot fix.

Starting in April 2021 in SAS 9.4M7 and SAS Viya 3.5, the mapping of the TIMESTAMP(p) data type is modified for Google BigQuery. You must have appropriate SAS/ACCESS software.

SAS 9.4M7 has the following new features and enhancements:

- The DS2 language adds support for Spark and Yellowbrick when appropriate SAS/ACCESS software is installed. For data type support, see "Data Types for Spark" on page 1921 and "Data Types for Yellowbrick" on page 1928.
- There is a new table option for Teradata. The TD_UNICODE_PASSTHRU= table option allows pass-through characters (PTCs) to be imported into and exported from Teradata. For more information, see "TD_UNICODE_PASSTHRU= Table Option" on page 1485.
- The TD_DATA_ENCRYPTION= table option supports aliases. For more information, see "TD_DATA_ENCRYPTION= Table Option" on page 1464.

The SAS Viya 3.5 release has the following new features and enhancements:

- The ability to read and write to MongoDB and Salesforce nonrelational databases with DS2 is added when appropriate SAS/ACCESS software is installed. The functionality is available when accessing the databases through a SAS library and on the CAS server. For information about data type support when using a SAS library, see "Data Types for MongoDB", "Data Types for Reading to Salesforce", and "Data Types for Writing to Salesforce" in Appendix 1, "Data Type Reference," on page 1867. See "Data Types for CAS Tables" when accessing the databases from the CAS server. For general information about reading and writing to the nonrelational databases, see information about the databases in *SAS/ACCESS for Nonrelational Databases: Reference*.
- Support for the VARBINARY data type when reading and writing to CAS tables has been added.
- There are improvements to data type support for various third-party relational databases when appropriate SAS/ACCESS software is installed. These features are available when the databases are accessed through a SAS library. This functionality is available in SAS Viya and in SAS 9.4M6.
  - DS2 now creates VARCHAR columns containing more than 65,535 characters as type STRING in Hive.
  - DS2 now reads and writes Teradata NUMBER columns.
  - DS2 now reads JSON columns in MySQL.

In the November 2019 release of SAS 9.4M6, Write support is added for the MongoDB and Salesforce databases when appropriate SAS/ACCESS software is installed. For information about data type support, see "Data Types for MongoDB" and "Data Types for Writing to Salesforce" in Appendix 1, "Data Type Reference," on page 1867. For information about writing to a nonrelational database through a SAS library, see information about the databases in *SAS/ACCESS for Nonrelational Databases: Reference*.

In the August 2019 release of SAS 9.4M6 and SAS Viya 3.4, the Google BigQuery and Snowflake databases are supported as data sources when appropriate SAS/ACCESS software is installed. Access is Read and Write, and in SAS Viya, through a SAS library or on the CAS server. See *SAS/ACCESS for Relational Databases: Reference* for information about reading and writing data in the databases.

In the April 2019 release of SAS 9.4M6, the MongoDB and Salesforce non-relational databases are supported as data sources when appropriate SAS/ACCESS software is installed. Access is Read-only. See "Data Types for MongoDB" on page 1898 and "Data Types for Writing to Salesforce" on page 1912 for data type support with DS2. See *SAS/ACCESS for Nonrelational Databases: Reference* for information about reading data from the databases.

SAS 9.4M6 has the following changes and enhancements:

- Inline declarations can be specified for DO loop counters.
- A RETAIN option has been added to the MERGE statement that produces a many-to-many match-merge that is similar to a DATA step merge.
- The DS2 language supports databases (such as PostgreSQL) that are compliant with JDBC as data sources when appropriate SAS/ACCESS software is installed. See Appendix 1, "Data Type Reference," on page 1867 for information about data type support.

SAS Viya 3.4 has the following changes and enhancements:

- Inline declarations can be specified for DO loop counters.
- A RETAIN option has been added to the MERGE statement that produces a many-to-many match-merge that is similar to a DATA step merge.
- The DS2 language supports databases (such as PostgreSQL) that are compliant with JDBC as data sources when appropriate SAS/ACCESS software is installed. Access to the databases is available through a SAS library, or on the CAS server. See Appendix 1, "Data Type Reference," on page 1867 for information about data type support.

SAS Viya 3.3 has the following changes and enhancements:

- The DS2 language supports SAS library access to all SAS 9.4 third-party data sources in SAS Viya when appropriate SAS/ACCESS software is installed. It supports CAS library access to all data sources for which SAS Data Connector software is installed.
- DS2 supports BIGINT (INT64) and INTEGER (INT32) as well as CHAR, DOUBLE, and VARCHAR data types in the CAS server. Columns that are defined as SMALLINT and TINYINT data types in CAS are now created as INTEGER instead of DOUBLE.

- If you run the DS2 program with the runDS2 action, SQL text can be passed in the SET statement.

SAS 9.4M5 has the following changes and enhancements:

- To provide a more comprehensive user experience, information for using DS2 with the CAS server has been incorporated into the *SAS 9.4 DS2 Language Reference*.
- The OF operator can now be used with variable lists and variable arrays.
- A new table option, INLINE, specifies that the package or thread source code is not saved to a table for reuse and is validated and compiled only when loaded by a data program.
- A new method, REGISTEROUTPARAMETER, is available for the SQLSTMT package to map output parameters in the FedSQL statement to IN_OUT parameters in a DS2 package METHOD statement.
- When a variable is used but not declared, a warning is sent to the SAS log. The warning now indicates the data type, length, and, in some cases, precision, that is assigned to the undeclared variable.
- Methods that have in_out parameters can have return values.
- A variable value is reset to missing or null when it is used before the SET or MERGE statement in the RUN method.
- When used with a subsetting IF statement, a variable value is reset to missing or null if it is used before the SET or MERGE statement in the RUN method.
- The DS2 language supports three new data sources: Amazon Redshift, Microsoft SQL Server, and Vertica, when appropriate SAS/ACCESS software is installed. See Appendix 1, "Data Type Reference," on page 1867 for information about data type support.

SAS 9.4M4 has the following changes and enhancements:

- Three new special automatic variables enable you to subset a problem across a DS2 thread.
  - _HOSTNAME_ returns the name of the worker nodes on which the DS2 program is running.
  - _NTHREADS_ returns the total number of threads running in the program.
  - _THREADID_ can be used in a thread program to identify individual threads.
- The DS2 language supports SAS Scalable Performance Data (SPD) Server tables as a data source. See Appendix 1, "Data Type Reference," on page 1867 for information about data type support.
- The private access modifier is now supported for attributes or methods that are intended for internal use within the package.
- The DO statement now enables you to use multiple index variable clauses.
- The TIME and TIMESTAMP precision is now preserved across a THREAD and DATA boundary.

SAS 9.4M3 has the following changes and enhancements:

- The DS2 language now supports the MERGE statement, which enables you to match-merge table data.
- A new statement, DS2_OPTIONS, can specify or change the default behavior of a DS2 program:
  - how DS2 processes a division by zero operation
  - write a note instead of an error message to the SAS log when an invalid function argument generates a missing value
  - non-existent values are processed as ANSI SQL null values
  - create a trace of what statements are executed
- The SELECT statement in embedded SQL text now supports the PARTITION BY, ORDER BY, INDSNUM, and WHERE clauses.
- A new format, BESTDOTX, produces a US-locale-based value regardless of current locale.
- Partitioned tables using the DBCREATE_TABLE_OPTS table option are now supported.
- The SPD Engine data sets now support AES encryption on the ENCRYPT and ENCRYPTKEY table options.
- The DS2 language supports two new data sources: HAWQ and Impala, when appropriate SAS/ACCESS software is installed. See Appendix 1, "Data Type Reference," on page 1867 for information about data type support.

SAS 9.4M2 has the following changes and enhancements:

- The Getting Started section has been rewritten and contains new examples.
- The DS2 language supports three new data sources: Hive, SAS HDMD, and PostgreSQL, when appropriate SAS/ACCESS software is installed. See Appendix 1, "Data Type Reference," on page 1867 for information about data type support.

SAS 9.4M1 has the following changes and enhancements:

- If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2_SASMISSING environment variable is set to TRUE.
- You can access any FCMP library as long as the connection string defines the catalog in which the FCMP library is located.
- The data type and character set encoding for an undeclared variable on the left side of an assignment statement is determined by the data type and character set encoding of the value on the right side of the assignment statement.
- The MDYAMPM format is now supported.
- The DS2 language supports SAP HANA as a data source, when appropriate SAS/ACCESS software is installed. See Appendix 1, "Data Type Reference," on page 1867 for information about data type support.

PART 1

Introduction

Chapter 1
Introduction to the DS2 Language 3

1

Introduction to the DS2 Language

Introduction to the DS2 Language . . . 3
Running DS2 Programs . . . 4
Supported Data Sources . . . 5
Intended Audience . . . 6
When to Use DS2 . . . 7
Converting DATA Step Programs to DS2 Programs . . . 7
Syntax Conventions for the DS2 Language . . . 8
Typographic Conventions . . . 8
Syntax Conventions . . . 8

Introduction to the DS2 Language

DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with Base SAS and SAS Viya and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages.

Several DS2 language elements accept embedded FedSQL syntax, and the runtime-generated queries can exchange data interactively between DS2 and any supported database. This allows SQL preprocessing of input tables, which effectively combines the power of the two languages.

In addition, DATA step logic can be transformed to run in environments where DS2 is supported and the DATA step is not. These environments include the following:

- SAS Federation Server
- SAS LASR Analytic Server
- SAS Embedded Process
- SAS Enterprise Miner

- SAS Decision Services

The DS2 procedure enables you to submit DS2 language statements from the SAS windowing environment or SAS Studio. For more information about PROC DS2, see *Base SAS Procedures Guide*.

Note: Because the DS2 language can be used with many data sources, the terms row, column, and table are used to describe the data elements. Comparing this to SAS DATA step terminology, a row corresponds to an observation, a column corresponds to a variable, and a table corresponds to a data set.

Running DS2 Programs

You can submit DS2 programs in one of the following ways.

- Through the SAS windowing environment or SAS Studio using the DS2 procedure. The DS2 procedure can be used to run DS2 code in Base SAS, SAS Viya, or on the CAS server. A single PROC DS2 step can contain several DS2 programs.

  For more information, see "DS2 Procedure" in *Base SAS Procedures Guide*.

- In SAS Studio using the runDS2 action on the CAS server. The runDS2 action is used in conjunction with the CAS procedure.

  Note: Unless you are using Python or Lua, it is recommended that you use PROC DS2 to submit DS2 code to the CAS server.

  For more information, see "DS2 Action Set" in *SAS Viya: System Programming Guide* and "DS2 in CAS: Concepts" in *SAS DS2 Programmer's Guide*.

- Directly to a data source using the SAS In-Database Code Accelerator.

  For more information about using the SAS In-Database Code Accelerator, see *SAS In-Database Products: User's Guide*.

- Directly to the SAS Federation Server using the SAS LIBNAME engine for SAS Federation Server.

Note: Some of these execution methods might require additional software licenses. For example, accessing any relational database management system (RDBMS) from Base SAS requires the appropriate SAS/ACCESS software license.

Supported Data Sources

DS2 can access the following data sources:

- Amazon Redshift[1]
- Aster
- CAS tables[1]
- DB2 for UNIX* and Windows operating environments
- Greenplum
- Hadoop (Hive[1] and HDMD[1])
- Google BigQuery on Linux x64[1]
- Impala[1]
- databases[1] that are compliant with JDBC (such as PostgreSQL)
- Memory Data Store (MDS)
- Microsoft SQL Server[1]
- MongoDB on Linux for x64[1] [2]
- MySQL (Beginning with SAS Viya 3.5, includes MySQL 8 Server)
- Netezza
- databases[1] that are compliant with ODBC (such as Microsoft SQL Server)
- Oracle[1]
- PostgreSQL[1]
- Salesforce on Linux for x64 [1]
- SAP (Read-only)
- SAP HANA[1]
- SAP IQ
- SAS data sets[1]
- SAS Scalable Performance Data Engine (SPD Engine) data set[1]
- SAS Scalable Performance Data Server (SPD Server) tables in UNIX and Windows operating environments
- Snowflake on Linux x64[1]
- Spark
- Teradata for UNIX[1] and Windows operating environments
- Vertica

- Yellowbrick

Note: The following data sources are not supported:

- Informix
- OLEDB
- SAP ASE

[1]These data sources are supported on the CAS Server.

[2]The MongoDB data source has special requirements for creating tables. Depending on your data, you might want to create these tables: a root table, a parent table, and a child table. You can create only root tables with DS2. To create parent and child tables, you must use FedSQL. See the information about MongoDB in *SAS/ACCESS for Relational Databases: Reference*.

The DS2 procedure and SAS Federation Server support different data sources. See *Base SAS Procedures Guide* and *SAS Federation Server: Administrator's Guide* for information about the data sources that each one supports.

For information about accessing the data sources, see *SAS/ACCESS for Relational Databases: Reference* and *SAS/ACCESS for Nonrelational Databases: Reference*, as appropriate.

Intended Audience

The information in this document is intended for the following users who perform in these roles:

- **Application developers** who write the client applications. They write applications that create tables, bulk load tables, manipulate tables, and query data.
- **Database administrators** who design and implement the client/server environment. They administer the data by designing the databases and setting up the data source metadata. That is, database administrators build the data model.
- **SAS programmers** who want or need to take advantage of the advanced features of the DS2 language such as increased numeric precision, parallel computation for CPU-bound processes, using explicit pass-through SQL queries as direct input to a DATA step process, or in-database processing in big data environments.
- **Data analysts** who want to push SAS processing into big-data domains using the scoring and code accelerators.

When to Use DS2

Typically, DS2 programs are written for applications that carry out the following actions:

- require the precision that results from using the new supported data types
- benefit from using the new expressions or write methods or packages available in the DS2 syntax
- need to execute SAS FedSQL from within the DS2 program
- execute outside a SAS session, for example, in-database processing on Hadoop or Teradata, in SAS Viya, or the SAS Federation Server
- take advantage of threaded processing in products such as the SAS In-Database Code Accelerator and SAS Enterprise Miner

Converting DATA Step Programs to DS2 Programs

In SAS 9.4M5, you can use the DSTODS2 procedure to translate the DATA step code into DS2. Not all DATA step code is supported for translation, and some manual translation might be required. Code lines that cannot be translated are placed in comments. For more information, see the "DSTODS2 Procedure" in *Base SAS Procedures Guide*.

After you have converted the DATA step code to DS2 and your program is syntactically complete, you can use the DS2 procedure to run your program from within SAS or SAS Viya, or you can use the HPDS2 procedure to run your program on the High-Performance Analytic Server distributed computing environment.

Syntax Conventions for the DS2 Language

Typographic Conventions

Type styles have special meanings when used in the documentation of the DS2 language syntax.

UPPERCASE BOLD
identifies DS2 keywords such the names of statements and functions (for example, PUT).

UPPERCASE ROMAN
identifies arguments and values that are literals (for example, FROM).

italic
identifies arguments or values that you supply. Items in italic can represent user-supplied values that are either one of the following.

- nonliteral values assigned to an argument (for example, ALTER=*alter-password*).
- nonliteral arguments (for example, KEEP=(*column-list*).

If more than one of an item in italics can be used, the items are expressed as *item* [, ...*item*].

`monospace`
identifies examples of SAS code.

Syntax Conventions

This documentation uses the Backus-Naur Form (BNF), specifically the same syntax notation used by Jim Melton in *SQL:1999 Understanding Relational Language Components*.

The main difference between traditional SAS syntax and the syntax that is used in the DS2 language reference documentation is in how optional syntax arguments are displayed. In traditional SAS syntax, angle brackets (< >) are used to denote optional syntax. In DS2 language syntax, square brackets ([ ]) are used to denote optional syntax and angle brackets are used to denote non-terminal components.

The following symbols are used in the DS2 language syntax.

::=
: This symbol can be interpreted as “consists of” or “is defined as”.

<>
: Angle brackets identify a non-terminal component (that is, a syntax component that can be further resolved into lower level syntax grammar).

[ ]
: Square brackets identify optional arguments. Any argument that is not enclosed in square brackets is a required argument. Do not enter square brackets unless they are preceded by a backward slash (\), which denotes that they are literal.

{ }
: Braces provide a method to distinguish required multi-word arguments. Do not enter braces unless they are preceded by a backward slash (\), which denotes that they are literal.

|
: A vertical bar indicates that you can choose one value from a group. Values that are separated by bars are mutually exclusive.

...
: An ellipsis indicates that the argument or group of arguments that follow the ellipsis can be repeated any number of times. If the ellipsis and the following arguments are enclosed in square brackets, they are optional.

\
: A backward slash indicates that the next character is a literal.

The following examples illustrate the syntax conventions that are described in this section. These examples contain selected syntax elements, not the complete syntax.

```
1 SET 2 <table-reference> [… [<table-reference>] [INDSNAME=variable];
   [3 BY [DESCENDING] 4 column [5 ...[DESCENDING] column];
  6 <table-reference>::=
  {table (table-options)} 7 | 8 \{sql-text 8 \}
```

1 SET is in uppercase bold because it is the name of the statement.

2 <table-reference> is in angle brackets because it is a non-terminal argument that is further resolved into lower level syntax grammar. You must supply at least one <table-reference>.

3 BY and DESCENDING are in uppercase roman because they are literal arguments. DESCENDING is in square brackets because it is an optional argument.

4 *column* is in italics because it is an argument that you can supply.

5 The square brackets and ellipsis around the second instance of *column* indicate that you can repeat this argument any number of times as long as the arguments are separated by commas.

6 The <table-reference>::= non-terminal argument syntax is read as follows: A <table-reference> consists of a table name and table options or embedded SQL text.

7 The vertical bar (|) indicates you can supply either *table* [*table-options*] or *sql-text*, but not both.

8 The backslash (\) before the braces around *sql-text* indicate that those braces are literals and must be entered.

PART 2

Getting Started

Chapter 2
DS2 and the DATA Step **13**

Chapter 3
Learning by Example: Using the Sample Programs **19**

Chapter 4
Building Blocks of DS2 Programs **33**

Chapter 5
Understanding DS2 Methods and Packages **45**

2

DS2 and the DATA Step

What Is DS2? 13
Similarities between DS2 and the DATA Step 13
Differences between DS2 and the DATA Step 14

What Is DS2?

DS2 is a SAS proprietary programming language that is used for data manipulation and data modeling applications. The DS2 language shares core features with the DATA step. However, DS2 capabilities extend far beyond those of the DATA step.

DS2 is a procedural language that has variables and scope, methods, packages, control flow statements, table I/O statements, and parallel programming statements. Methods and packages give DS2 modularity and data encapsulation. DS2 enables you to insert SQL directly into the SET statement, thus blending the power of two powerful data manipulation languages.

Similarities between DS2 and the DATA Step

DS2 and the DATA step share many language elements, and those elements behave in the same way:

- SAS formats.
- SAS functions.
- SAS statements such as DATA, SET, KEEP, DROP, RUN, BY, RETAIN, PUT, OUTPUT, DO, IF-THEN/ELSE, Sum, and others.

- DATA step keywords are included in the list of DS2 keywords.

You can perform most DATA step tasks using DS2:

- process variable arrays, multi-dimensional arrays, and hash tables
- convert between data types
- work with expressions
- calculate date and time values
- process missing values

Note: All supported DS2 syntax is covered in this document. Any syntax appearing in other SAS documentation is not part of the supported DS2 syntax unless it is also documented here.

Differences between DS2 and the DATA Step

Table 2.1 *Comparison by Topic*

Topic	DATA Step	DS2
Programming paradigm	Executable code resides in the DATA step and PROC step.	Executable code resides in methods.
Scope	No concept of scope. All variables are global.	Variables that are declared in a method have local scope. All other identifiers have global scope. There are three types of global scope: - data program - thread program - package
Declaring variables	Variables are not explicitly declared. Variables are created by assignment. The data type of a variable is determined by the context of how it is first used. All variables have global scope. Variables are also defined when the SET statement is used.	Variables are declared using the DECLARE statement, which also determines the data type and scope attributes of the variable. Variables can be declared by assignment, but a best practice is to enforce variable declaration strict mode by setting the system option DS2SCOND=ERROR or the PROC DS2 option SCOND=ERROR.

Topic	DATA Step	DS2
		Global variables are also defined when the SET or SET FROM statement is used.
Keywords and reserved words	No reserved keywords.	Keywords are reserved words.

Quotation marks	Single or double quotation marks can delimit a character constant. Here are two examples that are equivalent:
	`"Tom"`
	`'Tom'`
		ANSI SQL quoting standards are followed:
		■ Single quotation marks delimit a character constant.
		■ Double quotation marks delimit an identifier.
		For example, `"Tom"` is a delimited identifier, and `'Tom'` is a character constant.

PUT statement	Supports column and line parameters.	Column and line parameters are not supported. Dot notation parameters are not supported.
Variable attributes	Establishing the attributes of a variable requires the use of the LENGTH, FORMAT, INFORMAT, LABEL, and ATTRIB statements.	Establishing the attributes of a variable requires only the DECLARE statement and its HAVING clause.
Text that is resolved from macro variable references	Double quotation marks are required to reference the resolved value of a macro variable. Here is an example: `my_host = "&syshostname";`	Because ANSI SQL quoting standards are followed, double quotation marks denote delimited identifiers. To reference a macro variable in a literal string, use the %TSLIT macro function. Here is an example: `my_host = %tslit(&syshostname);`
Data types	Two data types are supported: numeric and character. Numeric data is signed, fractional, limited to 8 bytes, and has approximate precision. Character data is fixed length.	Most ANSI SQL data types are supported. Numeric types of varying sizes and precision. Character data types can be fixed length and variable length. DS2 supports ANSI date, time, and timestamp data types, but can also process SAS date, time, and datetime values using conversion functions.
Missing and null values	Supports only missing values. No concept of a null value.	Supports both missing and null values. Nulls, from a database, can be processed in ANSI mode or in SAS mode.
Automatic data type conversion	SAS tries to convert between character and numeric data types when one data type is assigned to a variable of the other data type.	Has many more rules because of many more data types. Most data types are coercible. DATE, TIME, TIMESTAMP, BINARY, and VARBINARY data types are not coercible.

Topic	DATA Step	DS2
SQL language statements	Available in PROC SQL, not in the DATA step. Operates only on SAS data sets as tables.	SQL SELECT statements can be written directly in and used as input for a DS2 SET statement. In addition, the SQLSTMT predefined package provides a way to pass SQL statements to a DBMS for execution and to access the result set returned by the DBMS.
SAS Macro	The DATA step can interact with a macro at run time (for example, CALL EXECUTE, SYMGET, and CALL SYMPUT).	When DS2 runs inside a Base SAS session (for example, in PROC DS2), SAS macros are available. SAS macro support is not available when DS2 runs in the SAS Federation Server and in grid computing environments such as the in-database SAS Embedded Process, the High-Performance Analytics grid, and the SAS In-Database Code Accelerator. **Note:** In-database processing, the High-Performance Analytics grid, and the SAS In-Database Code Accelerator are not supported in SAS Viya.

Overwriting data sets	The DATA step, like SAS procedures, overwrites an existing data set.
	`data one;`
	`  set two;`
	`run;`
	`data one;`
	`  set three;`
	`run;`
		In keeping with SQL and database standards, DS2 does not automatically overwrite existing data sets. You must use the **`overwrite`** option. Here is an example.
		`proc ds2;`
		`data one;`
		`  method run();`
		`  set two;`
		`  end;`
		`enddata;`
		`run;`
		`data one / overwrite=yes;`
		`  method run();`
		`  set three;`
		`  end;`
		`enddata;`
		`run;`
		`quit;`

Reading from and writing to the same data set	Permits reading from and writing to the same data set name in a DATA or PROC step.
	`data one;`
	`  set one;`
	`  s = s + 1;`
	`run;`
		DS2 does not allow reading from and writing to the same table in a single data program. In database fashion, the user must create a temporary table, drop the original table, and then rename the temporary table. This example is equivalent to what SAS does to implement the appearance of reading from and writing to the same data set at the successful conclusion of a DATA or PROC step.
		`proc ds2;`
		`data temp001;`
		`  method run();`
		`  set one;`

Topic	DATA Step	DS2

		`s = s * 1;`
		`end;`
		`enddata;`
		`run;`
		`quit;`
		
		`proc fedsql;`
		`drop table one;`
		`alter table temp001 rename to one;`
		`run;`
		`quit;`

3

Learning by Example: Using the Sample Programs

About the Getting Started Sample Programs 19
How to Run the Sample Programs 19
Recommended Options 20
Using the DS2 Procedure 21
Verifying Access to DS2 21

Your First Sample Programs 21
Overview of the Sample Programs 21
Example 1: "Hello World!" Program – In a System Method 22
Example 2: "Hello World!" Program – In a User-defined Method 23
Example 3: "Hello World!" Program – In a User-defined Package 24
Example 4: "Hello World!" Program – In the Implicit Loop 25
Example 5: "Hello World!" Program – In Multiple Package Instances 27
Example 6: "Hello World!" Program – Using a Thread 28

See Also 30

About the Getting Started Sample Programs

How to Run the Sample Programs

You can run all of the getting started sample programs in this chapter from a SAS session using the DS2 procedure. Each is a complete program; simply copy a program into your SAS session and submit it.

The getting started sample programs do not rely on a pre-existing data source, nor do they require a connection to a database. When data is needed, the program creates it.

Note: To submit the getting started sample programs, you must have access to SAS 9.4 or later. Some features might not be available if you do not have the latest release.

Recommended Options

All of the getting started sample programs run correctly with the following SAS system option global statement:

```
options DS2SCOND=ERROR;
```

You can also override the default DS2SCOND option setting, WARNING, by specifying the following DS2 procedure option:

```
proc ds2 SCOND=ERROR;
```

The ERROR setting enforces variable declaration strict mode. In strict mode, a compilation error occurs if you do not explicitly declare a variable.

TIP Variable declaration strict mode is the recommended best practice when writing DS2 programs.

Unlike Base SAS, DS2 protects existing tables from being overwritten. However, if you are developing or changing a table, package, or thread, you need the ability to overwrite these tables.

The OVERWRITE=YES table option enables you to overwrite the table. Here are some examples:

```
package foo /overwrite=yes;
thread work.foo(int x) /overwrite=yes;
data foo (overwrite=yes);
data my_data /overwrite=yes;
```

TIP The OVERWRITE= option requires the forward slash (/) syntax with the PACKAGE and THREAD statements. You can use either the / syntax or the parentheses syntax with the DATA statement.

By default, the value of the OVERWRITE= table option is NO.

Using the DS2 Procedure

When you use the DS2 procedure to write a program, place your code within the following framework:

```
options ds2scond=error;
proc ds2;
... DS2 statements ...
run;
quit;
```

For more information, see "DS2 Procedure" in *Base SAS Procedures Guide*.

Verifying Access to DS2

In a SAS session, run the following program:

```
proc ds2;
quit;
```

The following is written to the log:

```
3317  proc ds2;
3318  quit;

NOTE: PROCEDURE DS2 used (Total process time):
      real time           0.08 seconds
      cpu time            0.04 seconds
```

If you see an error message, such as `ERROR: Procedure DS2 not found`, see your system administrator.

Your First Sample Programs

Overview of the Sample Programs

Because many programmers prefer to learn by reading code, this chapter presents several sample programs before explaining the language constructs that the programs use. The sample programs help you quickly learn DS2 syntax and concepts so that you avoid common pitfalls.

The first sample program is the “Hello World!” program. Subsequent sample programs are variations on this program. Some variations might seem needlessly complex. The point is to demonstrate common structural programming elements of the DS2 language, not the best way to write the “Hello World!” program.

Note: Although the sample programs introduce syntax and concepts in order of increasing complexity, they do not rely on a particular order to be fully understood.

Example 1: “Hello World!” Program – In a System Method

Here is one way to code the "Hello World!" program. This program writes "Hello World!" to the SAS log from the INIT( ) system method.

What to Notice

- The variable MESSAGE has local scope because it is declared in the INIT( ) method.
- The INIT( ) system method automatically runs first in a DS2 data program.
- Single quotation marks delimit the character constant.
- The libs=work option limits the connection string to use only the SAS Work library. For more information, see the “PROC DS2 Statement” in *Base SAS Procedures Guide*.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2 libs=work;
data _null_;

  /* init() - system method */
  method init();
    declare varchar(16) message; /* method (local) scope */
    message = 'Hello World!';
    put message;
  end;
enddata;
run;
quit;
```

The following is written to the log:

```
Hello World!
```

Example 2: “Hello World!” Program – In a User-defined Method

This variation of the program writes "Hello World!" to the SAS log from a user-defined method.

What to Notice

- Because the variable MESSAGE has global scope in the data program, all methods can access it.
- The INIT( ) system method calls the user-defined GREET( ) method to write the message to the log.

  Note: The GREET( ) method is defined before the method that references it. Otherwise, a compilation error would occur.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2 libs=work;
data _null_;
  dcl varchar(16) message; /* data program (global) scope */

  /* greet() - user-defined method */
  method greet();
    put message;
  end;

  /* init() - automatically runs first in the data program.*/
  method init();
    message = 'Hello World';
    message = cat(message, '!');
    greet();
  end;
enddata;
run;
quit;
```

The following is written to the log:

```
Hello World!
```

Example 3: “Hello World!” Program – In a User-defined Package

This variation of the program writes “Hello World!” and other messages to the SAS log through a user-defined package.

What to Notice

- The PACKAGE statement uses the OVERWRITE=YES table option so that you can run the program more than once without error. By default, DS2 protects existing packages from being overwritten.
- The variable MESSAGE is declared inside the package, not in the data program.
- The package contains a constructor and two package methods to manipulate the greeting string.

  The FORWARD statement enables the SETMESSAGE( ) method to be defined after methods that reference it. Otherwise, a compilation error would occur.

  The SETMESSAGE( ) method uses the THIS operator to distinguish the global variable MESSAGE from the parameter that is named MESSAGE.
- In the data program, the DECLARE PACKAGE statement simultaneously declares a package variable and constructs an instance of the package using the package constructor.
- Dot notation provides access to package methods from the data program.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2 libs=work;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
  dcl varchar(100) message;    /* package (global) scope */
  FORWARD setMessage;

  /* greeting(MESSAGE) - constructor */
  method greeting(varchar(100) message);
    setMessage(message);
  end;

  method greet();
    put message;
  end;

  method setMessage(varchar(100) message);
    /* Must use THIS. to distinguish global */
    /* variable MESSAGE from parameter named MESSAGE. */
    this.message = message;
  end;
```

```
endpackage;
run;

/* data program */
data _null_;
  /* declares and instantiates an instance of the GREETING package */
  dcl package greeting g('Hello World!');  /* data program (global) scope */

  /* init() - automatically runs first in the data program.*/
  method init();
    g.greet();
    g.setMessage('What''s new?'); /* change greeting */
    g.greet();
end;
enddata;
run;
quit;
```

The following is written to the log:

```
Hello World!
What's new?
```

Example 4: “Hello World!” Program – In the Implicit Loop

This variation contains two data programs: one to create a table of greetings and one to process the table.

What to Notice

- In the first data program, the DATA statement uses the OVERWRITE=YES table option so that you can run the program more than once without error. By default, DS2 protects existing packages from being overwritten.
- The GREETING package has two constructors: a default constructor and one that accepts an argument.
- The second data program uses the two-step method for instantiating a package:

  1. The DECLARE PACKAGE statement declares a global package variable.

  2. The INIT( ) method uses the _NEW_ operator to create the package instance.
- In the second data program, the RUN( ) method uses the implicit loop of the SET statement to read and process the MESSAGE variable from each row in the table.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2 libs=work;
/* data program # 1 - Creates a table of greetings */
```

```
data work.greetings /overwrite=yes;
  dcl char(100) message; /* data program (global) scope */

  method init();
    message = 'Hello World!'; output;
    message = 'What''s new?'; output;
    message = 'Good-bye World!'; output;
  end;
enddata;
run;
quit;

proc ds2;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
  dcl varchar(100) message;    /* package (global) scope */
  forward setMessage;

  /* greeting() - default constructor */
  method greeting();
    setMessage('This is the default greeting.');
  end;

  /* greeting(MESSAGE) - constructor */
  method greeting(varchar(100) message);
    setMessage(message);
  end;

  method greet();
    put message;
  end;

  method setMessage(varchar(100) message);
    /* Must use THIS. to distinguish global */
    /* variable MESSAGE from parameter named MESSAGE. */
    this.message = message;
  end;
endpackage;
run;

/* data program #2 */
data _null_;
  dcl package greeting g; /* package (global) scope */

  /* init() - automatically runs first in the data program. */
  method init();
     g = _NEW_ greeting();
    g.greet();
  end;

  /* run() - automatically runs after INIT() completes. */
  method run();
    /* Implicit loop reads each row from the table */
    set work.greetings;
    g.setMessage(message); /* MESSAGE is read from row by SET statement */
    g.greet();
```

```
  end;
enddata;
run;
quit;
```

The following is written to the log:

```
This is the default greeting.
Hello World!

What's new?

Good-bye World!
```

Example 5: “Hello World!” Program – In Multiple Package Instances

This version of the program uses multiple instantiations of packages to obtain the same results as Example 4, without creating a table.

What to Notice

- The GREETING package is identical to the GREETING package in the previous example. Because the package already exists, the package block could have been omitted from this program.
- You can use different constructors in the same DECLARE PACKAGE statement.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2 libs=work;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
  dcl varchar(100) message;    /* package (global) scope */
  FORWARD setMessage;

  /* greeting() - default constructor */
  method greeting();
    setMessage('This is the default greeting.');
  end;

  /* greeting(MESSAGE) - constructor */
  method greeting(varchar(100) message);
    setMessage(message);
  end;

  method greet();
    put message;
  end;
```

```
  method setMessage(varchar(100) message);
    /* Must use THIS. to distinguish global */
    /* variable MESSAGE from parameter named MESSAGE. */
    this.message = message;
  end;
endpackage;
run;

/* data program */
data _null_;
  dcl package greeting
  g0() g1('Hello World!') g2('What''s new?') g3('Good-bye World!');

  /* init() - automatically runs first in the data program.*/
  method init();
    g0.greet();
    g1.greet();
    g2.greet();
    g3.greet();
  end;
enddata;
run;
quit;
```

The following is written to the log:

```
This is the default greeting.
Hello World!
What's new?
Good-bye World!
```

Example 6: “Hello World!” Program – Using a Thread

This version of the program uses a thread to read a table and pass variables to the data program.

What to Notice

- This single DS2 program includes two data programs, a package, and a thread program.
- The data program specifies two threads in the SET FROM statement.
- In the thread program, the RUN( ) method uses the implicit loop of the SET statement to read and process each MESSAGE variable from the table.
- In the data program, the RUN( ) method uses the implicit loop of the SET FROM statement to read and process each MESSAGE variable from the thread program.

What to Notice

- In the log, the thread program's TERM( ) output shows that the thread program ran twice, once per thread. In addition, the value of _N_ indicates the number of times that the RUN( ) method executed on behalf of each thread.

In a SAS session, copy or enter the following program. Then, submit it.

```
proc ds2 libs=work;
/* data program #1 - Creates a table of greetings */
data work.greetings /overwrite=yes;
  dcl char(100) message; /* data program (global) scope */

  method init();
    message = 'Hello World!'; output;
    message = 'What''s new?'; output;
    message = 'Good-bye World!'; output;
  end;
enddata;
run;

/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
  dcl varchar(100) message;    /* package (global) scope */
  forward setMessage;

  /* greeting() - default constructor */
  method greeting();
    setMessage('This is the default greeting.');
  end;

  /* greeting(MESSAGE) - constructor */
  method greeting(varchar(100) message);
    setMessage(message);
  end;

  method greet();
    put message;
  end;

  method setMessage(varchar(100) message);
    /* Must use THIS. to distinguish global */
    /* variable MESSAGE from parameter named MESSAGE. */
    this.message = message;
  end;
endpackage;
run;

/* thread program - Read the table */
thread work.t /overwrite=yes;

  /* run() - system method */
  method run();
    set work.greetings;
    output; /* output variables to calling program */
```

```
  end;

  /* term() - system method */
  method term();
    put _all_;
  end;
endthread;
run;

/* data program #2 */
data _null_;
   dcl package greeting g; /* data program (global) scope */
   dcl thread work.t t; /* data program (global) scope */

  /* init() - automatically runs first in the data program. */
   method init();
      g = _NEW_ greeting();
      g.greet();
   end;

  /* run() - automatically runs after INIT() completes. */
  method run();
      /* Implicit loop reads each row from the table */
      set from t threads=2;
      g.setMessage(message); /* MESSAGE read from row by SET FROM stmt */
      g.greet();
   end;
enddata;
run;
quit;
```

The following is written to the log:

```
This is the default greeting.
_N_=4 message=Good-bye World!

_N_=1 message=

Hello World!

What's new?

Good-bye World!
```

See Also

- For a high-level overview of DS2 concepts, see Chapter 4, “Building Blocks of DS2 Programs,” on page 33.
- For more information about DS2 methods and packages, see Chapter 5, “Understanding DS2 Methods and Packages,” on page 45.

- To look at advanced, real-world examples of DS2 programs, see Appendix 3, "DS2 Example Programs," on page 1955.

4

Building Blocks of DS2 Programs

Basic DS2 Language Concepts 33
Introducing DS2 Data Types 33
Automatic Conversions of Data Types 35
DS2 Programming Blocks and Scope 36
Variable Declaration in DS2 38
DS2 Methods and Packages 39
Parallel Processing in DS2 39
What Is a DS2 Program? 40
Example: Block Scope 40
Example: A Simple Thread Program 42

Basic DS2 Language Concepts

Introducing DS2 Data Types

Unlike Base SAS, DS2 supports many of the ANSI SQL data types that are native to the data sources that SAS supports. Thus, you can declare DS2 variables that do not require data type conversions to access data that is stored in a data source. The ability to avoid data type conversions enables you to move data efficiently to and from a database or other data source.

Note: The types of data that you can store depend on the native types that your data source supports.

For more information, see "DS2 Data Types" in *SAS DS2 Programmer's Guide*.

The following table summarizes factors to consider when choosing DS2 data types.

Table 4.1 *DS2 Data Types: Quick Reference*

DS2 Data Type Category	Considerations
Character	CHAR(n) and VARCHAR(n) use one byte per character. NCHAR(n) and NVARCHAR(n), which handle Unicode national language character sets, use two or four bytes per multibyte character. When you specify the length of a variable, *n*, the length determines the size of a fixed-length variable or the maximum size of a varying-length variable. **Note:** Fixed-length CHAR(n) is the equivalent of a DATA step character variable, where *n* is the number of characters. It is also the default type for an undeclared DS2 character variable.
Fractional numeric	DECIMAL(*p,s*) (alias: NUMERIC) has exact precision. Other fractional numeric types include DOUBLE, FLOAT(*p*), and REAL (single-precision floating point) and are considered approximate. **Note:** DOUBLE is the equivalent of a DATA step numeric variable. It is also the default type for an undeclared DS2 numeric variable.
Integer numeric	Signed, exact whole numbers with varying storage sizes: TINYINT (-128 to 127) – 1 byte SMALLINT (-32,768 to 32,767) – 2 bytes INTEGER (-2,147,483,648 to 2,147,483,647) – 4 bytes BIGINT (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807) – 8 bytes
Binary	BINARY(*n*) is fixed length. VARBINARY(*n*) is variable length.
Date and time	DS2 supports ANSI date, time, and timestamp data types, but can also process SAS date, time, and datetime values using these conversion functions: ■ TO_DATE casts a SAS numeric date to a DS2 DATE ■ TO_TIME casts a SAS numeric time to a DS2 TIME ■ TO_TIMESTAMP casts a SAS numeric datetime to a DS2 TIMESTAMP ■ TO_DOUBLE

DS2 Data Type Category	Considerations
	casts a DS2 DATE, TIME, or TIMESTAMP to a SAS numeric date, time, or datetime

Automatic Conversions of Data Types

About Automatic Conversions

To avoid unintended results, you must understand the DS2 rules for automatic data type conversion.

CAUTION
A type conversion can lead to the loss of data or precision, or both. Data type conversions are especially critical if you save DS2 data types in SAS data sets, because SAS data sets support only two data types. That is, DS2 variables might be automatically converted to either fixed-length character or numeric double.

An automatic type conversion occurs under the following circumstances:

- A character type is used in a numeric expression.
- A numeric type is used in a character expression.
- A call to a method supplies an argument value that does not exactly match the signature of the method.
- The types of the operands differ in a logical, arithmetic, relational, or concatenation expression.
- A DS2 data type is saved to a data source that does not support the type.

Coercion and Precedence

DS2 uses type coercion and precedence rules to determine the resulting data type for a conversion:

- Coercible data types can automatically convert to multiple data types.
- Non-coercible data types automatically convert to only character data types.
- Precedence determines the conversion type when an expression contains more than one data type.

For more information, see "DS2 Type Conversions" in *SAS DS2 Programmer's Guide*.

Conversion of Nulls and Missing Values

The mode that you use to process null and missing values can affect your data. The default mode for processing nulls and missing values, either SAS mode or ANSI mode, depends on the environment in which you submit your DS2 program and the options that you choose. For example, by default, the DS2 procedure processes data in SAS mode. The SAS Federation Server processes data in ANSI mode.

CAUTION
During multiple conversions, it is possible to lose the original meaning of data. This is particularly true in the values of SAS special missing values (._, .A-.Z).

For more information, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.

DS2 Programming Blocks and Scope

A programming block defines a section of a DS2 program that encapsulates variables and code. Programming blocks encourage the creation of modular, reusable code. In addition, a programming block defines the scope of identifiers within that block. In DS2, it is possible for variables to have the same name and data type, as long as they have different scope.

The following table summarizes the characteristics of DS2 programming blocks and scope.

Table 4.2 *DS2 Programming Blocks and Scope: Quick Reference*

Block	Delimiters	Scope
Data program	DATA...ENDDATA	Variables that are declared at the top of this programming block have global scope within the data program. In addition, variables that the SET statement references have global scope. Unless explicitly dropped, global variables in a data program are included in the program data vector (PDV). **Note:** Global variables exist for the duration of the data program.
Package	PACKAGE...ENDPACKAGE	Variables that are declared at the top of this programming block have global scope within the package. Package-scope variables are not included in the PDV of a data program that is using an instance of the package.

Block	Delimiters	Scope
		Note: Package-scope global variables exist for the duration of the package instance.
Thread program	THREAD...ENDTHREAD	Variables that are declared at the top of this programming block have global scope within the thread program. In addition, variables that the SET statement references have global scope. Unless explicitly dropped, global variables in a thread program are included in the thread output set. **Note:** Thread-scope global variables exist for the duration of the thread program instance, but they can be passed to the SET FROM statement in the data program.
Method	METHOD...END	A method is a subblock of a data program, package, or thread program. Method names have global scope within the enclosing block. Variables that are declared at the top of this programming block have local scope. Local variables are not included in the PDV. **Note:** Local variables exist for the duration of the method call.
DO loop	DO...END	Not applicable.

TIP Although method names have global scope, to forward reference a method, use the FORWARD statement at the top of the outer programming block. For more information, see "FORWARD Statement" on page 1252.

As the preceding table shows, there are three types of global scope:

- Data program
- Package
- Thread program

Variable Declaration in DS2

Implicit Declaration of Variables

As in Base SAS, DS2 enables you to implicitly create global variables by assignment. However, this is not recommended for these reasons:

- Subtle errors can occur (for example, if a variable name is misspelled).
- The data types of such variables are limited to DOUBLE and CHAR(*n*).
- Undeclared variables might make your program difficult for others to read and understand.

Variable Declaration with the DECLARE Statement

Unlike Base SAS, DS2 enables you to explicitly declare variables using the DECLARE statement. You can control all attributes of a variable in a single DECLARE statement.

The DECLARE statement takes this form:

DECLARE [PRIVATE] data-type *variable-list* [HAVING *having-clause*];

Note: A DECLARE statement is allowed only at the top of the programming block in which it is used. Otherwise, a compilation error occurs.

For more information, see "DECLARE Statement" on page 1224.

Variable Declaration Strict Mode

A best practice is to always run your DS2 programs using variable declaration strict mode. This enforces the explicit declaration of all program variables.

For more information about controlling variable declaration strict mode, see "DS2SCOND= System Option" on page 1345.

DECLARE Statement and Scope

When you use a DECLARE statement to define a variable, the variable assumes the scope of the programming block in which the variable is declared. A method is a

subblock of another programming block. Therefore, a variable that is declared in a method has local scope and exists only when the method executes. A variable that is declared outside a method has global scope within that programming block.

For a sample program that demonstrates scope, see "Example: Block Scope" on page 40.

Declaring a Package Instance

Although a package instance is simply a type of variable, it is a special case that is worth mentioning because it has two parts:

package variable
: a variable whose data type is a reference to a type of package.

package instance
: an instance of a type of package. Ideally, a package instance is always referenced by at least one package variable.

> **TIP** The scope of the package variable is determined by the programming block in which it is declared. A package instance does not have an associated scope, and its lifetime depends on the package variables that reference it.

For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

DS2 Methods and Packages

Methods and packages are explained in greater detail in Chapter 5, "Understanding DS2 Methods and Packages," on page 45.

Parallel Processing in DS2

DS2 supports parallel execution of a single program that can operate on different parts of a table. This type of parallelism is classified as Single Program, Multiple Data (SPMD) parallelism. In DS2, it is the responsibility of the programmer to identify the program statements that can operate in parallel.

> **TIP** For programs that are CPU bound, using a thread program on Symmetric Multiprocessing (SMP) hardware can improve performance. For programs that are either CPU or I/O bound, Massively Parallel Processing (MPP) hardware can improve performance.

For more information, see "Threaded Processing" in *SAS DS2 Programmer's Guide*.

For a sample program that demonstrates a simple use of threads, see "Example: A Simple Thread Program" on page 42.

What Is a DS2 Program?

For the purposes of getting started with DS2, a DS2 program is a set of DS2 statements that runs in the DS2 procedure. The getting started sample programs demonstrate that a DS2 program can serve many purposes, including but not limited to the following:

- to define and store one or more packages or threads, in permanent or temporary locations.
- to create one or more data sets or tables, in permanent or temporary locations.
- to run one or more data programs, using any, all, or none of the above components.
- any combination of the above. That is, you can create data, packages, and threads, plus run one or more data programs within a single DS2 program.

The order and number of programming blocks in a DS2 program does not matter, as long as the program compiles and contains enough RUN statements to execute the program.

> **TIP** Always check the log for compilation errors.

In addition, in a SAS session, you can alternate between the DS2 procedure and Base SAS to test your code and to achieve your programming goals.

Example: Block Scope

The following program demonstrates how scope determines the visibility of program identifiers.

What to Notice

- This program has six INTEGER variables that have the name `i`.
- This program has three user-defined methods named SHOWME( ).

```
options ds2scond=error;
proc ds2;
/* INNERPKG */
package innerPkg /overwrite=yes;
  dcl int i;   /* i is global in this package */
  dcl varchar(100) str;

  /* init() - initializes package variables */
  method init();
    i = 5;     /* global i */
    str = 'I am INNERPKG!';
  end;

  /* showMe() - displays values that INNERPKG can "see" */
  method showMe() returns int;
    dcl int i; /* local i */
    i = 10;    /* local i */
    put str;
    put 'Local i=' i;
    put 'Global i=' this.i;
    return 1;
  end;
endpackage;
run;

/* OUTERPKG */
package outerPkg /overwrite=yes;
  dcl int i; /* i is global in this package */
  dcl package innerPkg ip();
  dcl varchar(100) str;

  /* init() - initializes package variables */
  method init();
    i = 15;     /* global i */
    str = 'I am OUTERPKG!';
    ip.init();    /* tell INNERPKG to initialize itself */
  end;

  /* showMe() - displays values that OUTERPKG can "see" */
  method showMe();
    dcl int i; /* local i */
    i = 20; /* local i */
    put str;
    put 'Local i=' i; /* local i */
    put 'Global i=' this.i; /* global i */
    ip.showMe(); /* tell INNERPKG to show what it can "see" */
  end;
endpackage;
run;

/* data program */
data _null_;
  dcl int i; /* i is global in this data program */
  dcl package outerPkg op(); /* instantiate with default constructor */
  dcl varchar(100) str;
  FORWARD showMe;
```

```
  /* init() - automatically runs as the first method in the program */
  method init();
    dcl int rc;
    i = 25;        /* global i */
    str = 'I am the data program!';
    op.init();     /* tell OUTERPKG to initialize itself */
    showMe();      /* show what the data program can "see" */
  end;

  /* showMe() - displays values that the data program can "see" */
  method showMe();
    dcl int i;     /* local i */
    i = 30;        /* local i */
    put str;
    put 'Local i=' i;       /* local i */
    put 'Global i=' this.i; /* global i */
    op.showMe();   /* tell OUTERPKG instance to show what it can "see" */
  end;
enddata;
run;
quit;
```

The following is written to the log:

```
I am the data program!
Local i= 30
Global i= 25
I am OUTERPKG!
Local i= 20
Global i= 15
I am INNERPKG!
Local i= 10
Global i= 5
```

Example: A Simple Thread Program

The following program demonstrates how a thread creates data and passes variables to the data program.

What to Notice

- The parameterized thread accepts a value that must be initialized by the data program using the SETPARMS( ) system method.
- The OVERWRITE=YES table option enables the thread program to be overwritten.

  Note: The THREAD statement syntax requires the '/' (slash character) syntax.
- Because the data program specifies two threads, the thread program runs in two separate threads in a single process.

What to Notice

Note: This thread program produces one set of output variables per thread.

- Because threads run asynchronously, the order of processing is unpredictable, as the log shows.
- In the data program, the global accumulator variable TOTAL is implicitly retained because of the `total+answer;` Sum statement syntax.

```
options DS2SCOND=ERROR;
proc ds2;
/* thread program - Creates data in a loop */
thread work.t (double d) /overwrite=yes;
  dcl int x;
  dcl double y;

  method init();
    dcl int i; /* local - not included in the output table */

    do i = 1 to 9;
      x = i;
      y = i * 2.5 + d;
      put 'THREAD: i=' i ' x= ' x ' y= ' y;
      output; /* output variables include X and Y */
    end;
  end;

  method term();
    put 'THREAD TERM (_ALL_):';
    put _all_;
  end;
endthread;
run;

/* data program - Reads data from a thread program */
data;
  dcl thread work.t t;
  dcl double answer total;

  method init();
    t.setparms(1.25); /* initialize parameter of thread */
    put 'INIT (_ALL_):';
    put _all_;
  end;

  method run();
    set from t threads=2; /* input variables include X and Y */
    answer = x + y;
    total+answer; /* Sum statement syntax implicitly retains TOTAL */
    put ' x= ' x ' y= ' y ' answer= ' answer ' total= ' total;
  end;

  method term();
    put 'TERM: (_ALL_)';
    put _all_;
```

```
  end;
enddata;
run;
quit;
```

The following is written to the log:

```
INIT (_ALL_):
total=0 answer=. x= y=. _N_=1
THREAD: i= 1  x=  1  y=  3.75
THREAD: i= 1  x=  1  y=  3.75
THREAD: i= 2  x=  2  y=  6.25
THREAD: i= 3  x=  3  y=  8.75
THREAD: i= 4  x=  4  y=  11.25
THREAD: i= 5  x=  5  y=  13.75
THREAD: i= 6  x=  6  y=  16.25
THREAD: i= 7  x=  7  y=  18.75
THREAD: i= 8  x=  8  y=  21.25
THREAD: i= 2  x=  2  y=  6.25
THREAD: i= 9  x=  9  y=  23.75
THREAD: i= 3  x=  3  y=  8.75
THREAD TERM (_ALL_):
THREAD: i= 4  x=  4  y=  11.25
d=1.25 x= y=. _N_=1
THREAD: i= 5  x=  5  y=  13.75
THREAD: i= 6  x=  6  y=  16.25
 x=  1  y=  3.75  answer=  4.75  total=  4.75
THREAD: i= 7  x=  7  y=  18.75
THREAD: i= 8  x=  8  y=  21.25
THREAD: i= 9  x=  9  y=  23.75
THREAD TERM (_ALL_):
d=1.25 x= y=. _N_=1
 x=  2  y=  6.25  answer=  8.25  total=  13
 x=  3  y=  8.75  answer=  11.75  total=  24.75
 x=  4  y=  11.25  answer=  15.25  total=  40
 x=  5  y=  13.75  answer=  18.75  total=  58.75
 x=  6  y=  16.25  answer=  22.25  total=  81
 x=  7  y=  18.75  answer=  25.75  total=  106.75
 x=  8  y=  21.25  answer=  29.25  total=  136
 x=  9  y=  23.75  answer=  32.75  total=  168.75
 x=  1  y=  3.75  answer=  4.75  total=  173.5
 x=  2  y=  6.25  answer=  8.25  total=  181.75
 x=  3  y=  8.75  answer=  11.75  total=  193.5
 x=  4  y=  11.25  answer=  15.25  total=  208.75
 x=  5  y=  13.75  answer=  18.75  total=  227.5
 x=  6  y=  16.25  answer=  22.25  total=  249.75
 x=  7  y=  18.75  answer=  25.75  total=  275.5
 x=  8  y=  21.25  answer=  29.25  total=  304.75
 x=  9  y=  23.75  answer=  32.75  total=  337.5
TERM: (_ALL_)
total=337.5 answer=. x=9 y=23.75 _N_=19
```

5

Understanding DS2 Methods and Packages

Modularity, Encapsulation, and Abstraction in DS2 45

Getting Started with DS2 Methods 46
What Are DS2 Methods? 46
What Are DS2 System Methods? 46
What Are DS2 User-defined Methods? 48

Getting Started with DS2 Packages 49
What Are DS2 Packages? 49
What Are DS2 Predefined Packages? 51

Example: Introduction to DS2 Methods 51

Example: Introduction to DS2 Packages 53

Modularity, Encapsulation, and Abstraction in DS2

When you use DS2 methods and packages, you approach writing SAS programs differently than you do when programming in Base SAS. These DS2 language constructs follow a more structured-programming and object-oriented approach.

In general, DS2 methods are like functions, procedures, subroutines, and the methods of object-oriented languages such as Java. A *method* can be thought of as a module that contains a sequence of instructions to perform a specific task. DS2 methods can exist only within a data program, thread program, or package.

Thus, methods enable you to break up a complex problem into smaller modules. Such modules are easier to design, implement, and test. Code reuse can shorten development time and help standardize often-repeated or business-specific programming tasks. Also, modular programming enhances readability and understandability by testers and other programmers.

DS2 packages enable encapsulation and abstraction of behavior. DS2 packages are similar to classes in object-oriented languages. However, a DS2 package can also be used as a bucket of useful but unrelated methods and variables, if that meets your needs.

A DS2 package bundles data and methods into a named object that can be stored and reused by other DS2 programs. Although DS2 packages do not hide their data or methods (there is no concept of public and private), packages can be designed to abstract behavioral details. In such a package, the methods define the object and enable controlled manipulation of package data.

Getting Started with DS2 Methods

What Are DS2 Methods?

Because executable code can reside only in methods, methods are the structural building blocks of DS2 programs.

There are two types of methods:

System Methods
: Also known as predefined, these methods provide the structural and functional framework for your program to execute.

User-defined Methods
: Similar to functions, procedures, and subroutines in other languages, these are the methods that you create or that someone else created for reuse.

All methods are subblocks of other programming blocks. Each method creates a method scope in which local variables can be defined.

A method programming block begins with the METHOD keyword and ends with the END keyword, as the following example shows.

```
method foo();
...DS2 variables and statements...
end;
```

For more information, see “Methods” in *SAS DS2 Programmer’s Guide*.

What Are DS2 System Methods?

System methods provide the structural framework that runs your code. That is, to create a program, simply add DS2 statements to one or more of these system methods: INIT( ), RUN( ), TERM( ).

System methods also provide special functionality that SAS programmers need and expect for their data and thread programs, such as implicit looping. In addition, system methods enable the semantic grouping of code, which helps make DS2 programs easier to organize, understand, and maintain.

Here are basic facts about system methods:

- Every DS2 program contains—and executes—all three of the main system methods. If you do not explicitly code a system method, the system executes a default version.
- You can explicitly code any, all, or none of the system methods.

  Note: A program that contains no system methods is syntactically correct, but performs no useful function.

- You cannot change the signature of a system method, and you cannot overload a system method.
- You cannot directly call a system method, with the exception of SETPARMS( ), which applies only to thread programs.

For more information, see "Overview of System Methods" on page 1333.

The following table summarizes the execution details of each DS2 system method.

Table 5.1 *DS2 System Methods: Execution Details*

System Method	Execution Details
INIT( )	Automatically executes one time, as the first method of a program. For more information, see "INIT Method" on page 1335.
RUN( )	Automatically executes after INIT( ) completes. The RUN( ) method is the functional equivalent of the DATA step, running as an implicit loop if the method contains a SET or SET FROM statement. For more information, see "RUN Method" on page 1336.
TERM( )	Automatically executes one time, as the last method of a program. For more information, see "TERM Method" on page 1340.
SETPARMS( )	Executes one time, when called from a data program, to initialize the values of a parameterized thread. For more information, see "SETPARMS Method" on page 1338.

What Are DS2 User-defined Methods?

Similar to functions, procedures, and subroutines in other languages, these are the methods that you create or that someone else created for reuse. Because methods are subblocks of other programming blocks, user-defined methods can exist in data programs, packages, and thread programs.

The following table summarizes basic concepts of user-defined methods.

Table 5.2 *DS2 User-defined Methods: Basic Concepts*

User-defined Method Topic	Description
Scope	The name of a user-defined method has global scope within the programming block in which it is defined. In addition, each method creates a method scope in which local variables can be defined. **Note:** You can call a user-defined method from wherever the method is in scope, and you can do it as many times as needed.
Method parameters	A user-defined method can accept arguments in the following ways: ■ by value. The argument value is copied to the method. ■ by reference (IN_OUT parameter). The method modifies the value of the argument variable. **TIP** DS2 methods can support up to 1000 arguments. A DS2 method that has more than 1000 arguments can generate a compilation error.
Return values	Like a function, a user-defined method can return a value.

Method overloading	User-defined methods that have the same name can exist in the same scope if their argument signatures are unique. That is, if only the return type differs, then the overloading is ambiguous.
	The following examples show an overloaded method with three unique argument signatures:
	`method squareIt(int value) returns int;`
	`  return value**2;`
	`end;`
	
	`method squareIt(decimal(6,2) value) returns decimal(8,4);`
	`  return value**2;`
	`end;`
	
	`method squareIt(int value, IN_OUT int square);`
	`  square = value**2;`
	`end;`

User-defined Method Topic	Description
Calling	Unlike system methods, which automatically run, user-defined methods must be called. You can call a user-defined method as many times as needed.

For more information, see "METHOD Statement" on page 1270.

Getting Started with DS2 Packages

What Are DS2 Packages?

DS2 packages are language constructs that bundle variables and methods into named objects that can be stored and reused by other DS2 programs. The main benefit of a package derives from the reusability of a set of useful methods. However, DS2 packages are capable of much more.

A DS2 package is similar to a class in object-oriented languages. Thus, you can write packages that approximate the encapsulation and abstraction of object-oriented classes.

Note: A DS2 package is not a program. A DS2 package is a template for instantiating an object that can be used in a program.

There are two types of packages:

Predefined Packages
: These packages encapsulate common functionality that is useful to many customer solutions (for example, manipulating hash and matrix data structures). Predefined packages are part of DS2.

User-defined Packages
: These are the packages that you create or that someone else created for reuse.

A package is defined by a package programming block. A package begins with the PACKAGE keyword and ends with the ENDPACKAGE keyword, as the following example shows.

```
package foo;
  ...DS2 variables, statements, and methods...
endpackage;
```

The following table summarizes basic concepts of both user-defined and predefined packages.

Table 5.3 *DS2 Packages: Basic Concepts*

Package Topic	Description
Scope	The names of package methods and variables have global scope within the package. **Note:** You can access a package's methods and variables from wherever a package variable that references the package instance is in scope.

Package methods	Because packages are not programs, they do not have system methods that run automatically. Thus, to execute a package method, your program must call it.
	After you instantiate a package, use dot notation to access a method of the package instance, as the following example shows.
	`method run();`
	`  dcl package matrix m(2, 3);`
	`  dcl double mr mc;`
	`  mr=m.rows();`
	`  mc=m.cols();`
	`  put mr=;`
	`  put mc=;`
	`end;`

Preparing packages for use	A package must be compiled and stored before it can be used in a program. You can use the DS2 procedure to define and store a user-defined package. For more information, see "PACKAGE Statement" on page 1286.

Overwriting packages	Unlike Base SAS, DS2 protects existing tables from being overwritten. However, if you are developing or changing a package, you need the ability to overwrite the package. To overwrite an existing package, use the OVERWRITE=YES table option, as the following example shows.
	`package greeting /overwrite=yes;`
	`  ...DS2 variables, statements, and methods...`
	`endpackage;`
	`run;`

Instantiating packages	To use a package, you create an instance of the package (a package instance) and a variable that references the instance (a package variable). You can instantiate a package in two ways:
	■ The DECLARE PACKAGE statement can simultaneously declare one or more package variables. For each declared package variable, the DECLARE PACKAGE statement can optionally construct a package instance. For example:
	`dcl package matrix m0 m1(3, 3) m2(5, 4);`
	The above declare statement declares three matrix package variables (m0, m1, and m2) and constructs two matrix package instances (one of which is assigned to m1 and one of which is assigned to m2).
	■ by first declaring the package variable and then assigning a package instance, using the _NEW_ operator:
	`data _null_;`

Package Topic	Description

	dcl package matrix m1 m2;
	
	method init();
	dcl package matrix m3;
	
	m1 = _NEW_ matrix(3, 2);
	m3 = _NEW_ matrix(5, 4);
	m2 = _NEW_ matrix();
	end;
	
	...DS2 methods...
	enddata;
	run;
	
	Note: If you declare a package variable without simultaneously declaring an instance of the package, the value of the package variable is NULL.

For more information, see "DS2 Packages" in *SAS DS2 Programmer's Guide*.

What Are DS2 Predefined Packages?

Predefined packages encapsulate common functionality that is useful to many customer solutions.

For a list of the predefined packages that are available with DS2, see "Predefined DS2 Packages" in *SAS DS2 Programmer's Guide*.

You can find many example programs that use predefined packages in Appendix 3, "DS2 Example Programs," on page 1955.

Example: Introduction to DS2 Methods

The following program demonstrates many characteristics of both system and user-defined DS2 methods.

What to Notice

- This program has three overloaded user-defined methods named SQUAREIT( ).
- This program contains three system methods.

```
options DS2SCOND=ERROR;
proc ds2;
/* data program */
```

```
data _null_;
  dcl int root1 result1;
  dcl decimal(6,2) root2;
  dcl decimal(8,4) result2;

  /* overloaded user-defined method */
  method squareIt(int value) returns int;
    return value**2;
  end;

  /* overloaded user-defined method */
  method squareIt(decimal(6,2) value) returns decimal(8,4);
    return value**2;
  end;

  /* overloaded user-defined method with IN_OUT parameter */
  method squareIt(int value, IN_OUT int square);
    square = value**2;
  end;

  /* system method */
  method init();
    root1 = 3.01;
    root2 = 3.01;
  end;

  /* system method */
  method run();
    result1 = squareIt(root1);
    put 'The square of INTEGER ' root1 ' is ' result1;

    result2 = squareIt(root2);
    put 'The square of DECIMAL ' root2 ' is ' result2;

    root1 = 4.99;
    squareIt(root1, result1);
    put 'The square of INTEGER ' root1 ' is ' result1;
  end;

  /* system method */
  method term();
    put _all_; /* final values of global variables */
  end;
enddata;
run; quit;
```

The following is written to the log:

```
The square of INTEGER  3  is  9
The square of DECIMAL  3.01  is  9.0601
The square of INTEGER  4  is  16
root1= root2= result1= result2= _N_=1
```

Example: Introduction to DS2 Packages

The following program demonstrates some basic concepts of packages. This is a version of the “Hello World!” program that uses a package to write messages to the SAS log.

What to Notice

- The PACKAGE statement uses the OVERWRITE=YES table option so that you can run the program more than once without error. By default, DS2 protects existing packages from being overwritten.
- In the data program, the DECLARE PACKAGE statement simultaneously declares package variables and constructs two separate instances of the package using the package constructor.
- Dot notation provides access to package methods and package-global variables from the data program.

  Note: Although this sample program shows that you can directly access package-global variables, the recommended best practice is to avoid this practice unless you have a valid reason to do so.

```
options DS2SCOND=ERROR;
proc ds2;
/* GREETING - User-defined package that writes a message to the SAS log */
package greeting /overwrite=yes;
  dcl varchar(100) message;    /* package (global) scope */

  /* greeting(MESSAGE) - constructor that accepts an argument */
  method greeting(varchar(100) message);
    THIS.message = message;
  end;

  /* greeting() - default constructor */
  method greeting();
  end;

  /* package method */
  method greet();
    if not missing(message) then
      put message;
    else put 'Message is null or missing.';
  end;
run;

/* data program */
data _null_;
  /* declares and instantiates two instances of the GREETING package */
```

```
  dcl package greeting g1('Hello World!') g2();  /* data program (global) scope */

  /* init() - system method */
  method init();
    g1.greet();
    g2.greet();
    g2.message = 'Good-bye World!';
    g2.greet();
  end;
enddata;
run;
quit;
```

The following is written to the log:

```
Hello World!
Message is null or missing.
Good-bye World!
```

PART 3

DS2 Language Reference

Chapter 6
DS2 Formats 57

Chapter 7
DS2 Functions 229

Chapter 8
DS2 Informats 1195

Chapter 9
DS2 Operators 1199

Chapter 10
DS2 Statements 1203

Chapter 11
DS2 System Methods 1333

Chapter 12
DS2 System Options 1343

Chapter 13
DS2 Table Options 1347

Chapter 14
DS2 FCMP Package Methods, Operators, and Statements 1495

Chapter 15
DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements 1503

Chapter 16
DS2 HTTP Package Methods, Operators, and Statements 1587

Chapter 17
DS2 JSON Package Methods, Operators, and Statements 1631

Chapter 18
DS2 Logger Package Methods, Operators, and Statements 1665

Chapter 19
DS2 Matrix Package Methods, Operators, and Statements **1675**

Chapter 20
DS2 PCRXFIND Package Methods, Operators, and Statements **1753**

Chapter 21
DS2 PCRXREPLACE Package Methods, Operators, and Statements . . . **1771**

Chapter 22
DS2 SQLSTMT Package Methods, Operators, and Statements **1779**

Chapter 23
DS2 TZ Package Methods, Operators, and Statements **1849**

6

DS2 Formats

Overview of Formats 59

General Format Syntax 59

Using Formats in DS2 60
How to Specify Formats 60
Write Formatted Data in DS2 60

Validation of DS2 Formats 62

DS2 Format Examples 62

Format Categories 62

Dictionary 75
$BASE64Xw. Format 75
$BINARYw. Format 76
$CHARw. Format 77
$HEXw. Format 78
$OCTALw. Format 80
$QUOTEw. Format 81
$REVERJw. Format 83
$REVERSw. Format 84
$UPCASEw. Format 85
$UUIDw. Format 86
$w. Format 88
B8601DAw. Format 89
B8601DNw. Format 90
B8601DTw.d Format 92
B8601DZw. Format 93
B8601LZw. Format 95
B8601TMw.d Format 97
B8601TZw. Format 99
BESTw. Format 101
BESTDOTXw. Format 103
BESTDw.p Format 104
BINARYw. Format 106
COMMAw.d Format 108
COMMAXw.d Format 109
Dw.p Format 111

DATEw. Format ... 113
DATEAMPMw.d Format ... 114
DATETIMEw.d Format ... 116
DAYw. Format ... 119
DDMMYYw. Format ... 120
DDMMYYxw. Format ... 121
DOLLARw.d Format ... 124
DOLLARXw.d Format ... 125
DOWNAMEw. Format ... 127
DTDATEw. Format ... 128
DTMONYYw. Format ... 130
DTWKDATXw. Format ... 132
DTYEARw. Format ... 133
DTYYQCw. Format ... 135
E8601DAw. Format ... 136
E8601DNw. Format ... 137
E8601DTw.d Format ... 139
E8601DZw. Format ... 141
E8601LZw. Format ... 143
E8601TMw.d Format ... 145
E8601TZw.d Format ... 147
Ew. Format ... 149
EUROw.d Format ... 150
EUROXw.d Format ... 152
FLOATw.d Format ... 153
FRACTw. Format ... 155
HEXw. Format ... 156
HHMMw.d Format ... 157
HOURw.d Format ... 159
IEEEw.d Format ... 161
JULIANw. Format ... 163
MDYAMPMw.d Format ... 164
MMDDYYw. Format ... 166
MMDDYYxw. Format ... 168
MMSSw.d Format ... 170
MMYYw. Format ... 172
MMYYxw. Format ... 173
MONNAMEw. Format ... 175
MONTHw. Format ... 177
MONYYw. Format ... 178
NEGPARENw.d Format ... 179
NENGOw. Format ... 181
OCTALw. Format ... 182
PERCENTw.d Format ... 184
PERCENTNw.d Format ... 185
QTRw. Format ... 187
QTRRw. Format ... 188
ROMANw. Format ... 189
TIMEw.d Format ... 190
TIMEAMPMw.d Format ... 192
TODw.d Format ... 194
VAXRBw.d Format ... 196
w.d Format ... 197
WEEKDATEw. Format ... 199

WEEKDATXw. Format 201
WEEKDAYw. Format 203
YEARw. Format 204
YENw.d Format 205
YYMMw. Format 207
YYMMxw. Format 208
YYMMDDw. Format 211
YYMMDDxw. Format 213
YYMONw. Format 215
YYQw. Format 216
YYQxw. Format 218
YYQRw. Format 220
YYQRxw. Format 222
YYQZw. Format 224
Zw.d Format 226

Overview of Formats

A **format** is an instruction that DS2 uses to write data values. You use formats to control the written appearance of data values, or, in some cases, to group data values together for analysis. For example, the ROMAN*w*. format, which converts numeric values to roman numerals, writes the numeric value 2006 as `MMVI`.

General Format Syntax

DS2 formats have the following form:

[$] *format* [*w*] . [*d*]

Here is an explanation of the syntax:

$
indicates a character format; its absence indicates a numeric format.

format
names the format. The format is a DS2 format or a user-defined format that was previously defined with the INVALUE statement in PROC FORMAT. For more information about user-defined formats, see the *Base SAS Procedures Guide*.

Restriction To create and access user-defined formats, a SAS session must be available in order to access the SAS catalog file that stores the SAS format definitions.

w
specifies the format width, which for most formats is the number of columns in the input data.

d
specifies an optional decimal scaling factor in the numeric formats. DS2 divides the input data by 10 to the power of *d*.

Tip When the value of *d* is greater than 15, the precision of the decimal value after the 15th decimal place might not be accurate.

Formats always contain a period (.) as a part of the name. If you omit the *w* and the *d* values from the format, DS2 uses default values. The *d* value that you specify with a format tells DS2 to display that many decimal places, regardless of how many decimal places are in the data. Formats never change or truncate the internally stored data values.

For example, in DOLLAR10.2, the *w* value of 10 specifies a maximum of 10 columns for the value. The *d* value of 2 specifies that two of these columns are for the decimal part of the value, which leaves eight columns for all the remaining characters in the value. This includes the decimal point, the remaining numeric value, a minus sign if the value is negative, the dollar sign, and commas, if any.

If the format width is too narrow to represent a value, DS2 tries to squeeze the value into the space available. Character formats truncate values on the right. Numeric formats sometimes revert to the BEST*w.d* format. DS2 prints asterisks if you do not specify an adequate width. In the following example, the result is x=**.

```
x=123;
put x= 2.;
```

If you use an incompatible format, such as using a numeric format to write character values, DS2 first attempts to use an analogous format of the other type. If this is not feasible, an error message that describes the problem appears in the SAS log.

Using Formats in DS2

How to Specify Formats

In DS2, specify formats as an attribute of the HAVING clause of the DECLARE statement. The HAVING clause provides equivalent functionality to the Base SAS FORMAT and LABEL statements, except that now the attribute must be specified in the declaration statement of the variable.

For example, in the following statement, the columns *profit* and *loss* are declared with the EURO13.2 format.

```
dcl double profit loss having format euro13.2;
```

Note: In DS2, a format for a column cannot be changed or removed. However, you can use formats as arguments in the PUT function to write data in a different format. For more information, see "Write Formatted Data in DS2" on page 61.

Write Formatted Data in DS2

You can use formats as arguments in the PUT function to write formatted data. If a value is not specified for the format width or decimal specification, DS2 uses the default values for that format. Note that when using input-width aware formats such as $OCTAL and $HEX, the width, decimal specification, or both is calculated from the input field width. For example, $HEX uses 2 times the width and $OCTAL uses 3 times the width.

For example, in this case, the PUT function returns the formatted value of 99 using the BEST12. format.

```
x=99;
y=put(x, best12.);
```

If the PUT function is used without a format, all data is written without formatting.

Any type conversions of the value that is formatted are done based on the format name. Any value that is passed to the PUT function with a numeric format is converted, if necessary, to DOUBLE. Any value that is passed to the PUT function with a character format is converted to NCHAR. For more information, see the "PUT Function" on page 992.

DS2 supports SAS formats as follows.

- Formats that are supplied by SAS and user-defined formats can be used. For information about how to create your own format in SAS, see PROC FORMAT in the *Base SAS Procedures Guide*.

  Note: To create and access user-defined formats, a SAS session must be available in order to access the SAS catalog file that stores the SAS format definitions.

- Only the SAS data set and SPD data sets support storing and retrieving a format with a column.
- Formats can be associated with all data types, but all data types are converted to either CHAR or DOUBLE.
- You associate SAS formats with a column by using the HAVING clause of the DS2 DECLARE statement. For more information, see "How to Specify Formats" on page 60.

Validation of DS2 Formats

Formats are not validated by a data source or applied to a column until execution time.

When metadata for a column is requested, the format name is returned without validation.

DS2 Format Examples

```
declare datetime shipdate having format dateampm22.2;
x = put (name, $char8.);
x = put (price, myformat. -c);
```

Format Categories

Formats can be categorized by the types of values that they operate on. Each DS2 format belongs to one of the following categories:

Table 6.1 *Format Category Descriptions*

Category	Description
CAS	formats that can be used on the CAS server.
Character	writes character data values from character variables.
Date and time	writes character data values from character variables.
Numeric	writes numeric data values from numeric variables.

The following table provides brief descriptions of DS2 formats. For more detailed information, see the individual formats.

Category	Language Elements	Description
CAS	$BASE64Xw. Format (p. 75)	Converts character data into ASCII text by using Base 64 encoding.
	$BINARYw. Format (p. 76)	Converts character data to binary representation.
	$CHARw. Format (p. 77)	Writes standard character data.
	$HEXw. Format (p. 78)	Converts character data to hexadecimal representation.
	$OCTALw. Format (p. 80)	Converts character data to octal representation.
	$QUOTEw. Format (p. 81)	Writes data values that are enclosed in double quotation marks.
	$REVERJw. Format (p. 83)	Writes character data in reverse order and preserves blanks.
	$REVERSw. Format (p. 84)	Writes character data in reverse order and left aligns.
	$UPCASEw. Format (p. 85)	Converts character data to uppercase.
	$UUIDw. Format (p. 86)	Writes character data in universally unique identifier (UUID) format.
	$w. Format (p. 88)	Writes standard character data.
	BESTw. Format (p. 101)	SAS chooses the best notation.
	BESTDOTXw. Format (p. 103)	Specifies that SAS choose the best notation and use a dot as a decimal separator.
	BESTDw.p Format (p. 104)	Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.
	BINARYw. Format (p. 106)	Converts numeric values to binary representation.
	COMMAw.d Format (p. 108)	Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.
	COMMAXw.d Format (p. 109)	Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.
	Dw.p Format (p. 111)	Prints numeric values, possibly with a great range of values, lining up decimal places for values of similar magnitude.
	DATEw. Format (p. 113)	Writes SAS date values in the form ddmmmyy, ddmmmyyyy, or dd-mmm-yyyy.
	DATEAMPMw.d Format (p. 114)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss with AM or PM.
	DATETIMEw.d Format (p. 116)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss.

Category	Language Elements	Description
	DAYw. Format (p. 119)	Writes SAS date values as the day of the month.
	DDMMYYw. Format (p. 120)	Writes SAS date values in the form ddmm<yy>yy or dd/mm/<yy>yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	DDMMYYxw. Format (p. 121)	Writes SAS date values in the form ddmm<yy>yy or dd-mm-yy<yy>, where x in the format name is a character that represents the special character that separates the day, month, and year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	DOLLARw.d Format (p. 124)	Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.
	DOLLARXw.d Format (p. 125)	Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.
	DOWNAMEw. Format (p. 127)	Writes SAS date values as the name of the day of the week.
	DTDATEw. Format (p. 128)	Expects a SAS datetime value as input and writes the SAS date values in the form ddmmmyy or ddmmmyyyy.
	DTMONYYw. Format (p. 130)	Writes the date part of a SAS datetime value as the month and year in the form mmmyy or mmmyyyy.
	DTWKDATXw. Format (p. 132)	Writes the date part of a SAS datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy).
	DTYEARw. Format (p. 133)	Writes the date part of a SAS datetime value as the year in the form yy or yyyy.
	DTYYQCw. Format (p. 135)	Writes the date part of a SAS datetime value as the year and the quarter and separates them with a colon (:).
	E8601DAw. Format (p. 136)	Writes date values by using the ISO 8601 extended notation yyyy-mm-dd.
	E8601DNw. Format (p. 137)	Writes dates from SAS datetime values by using the ISO 8601 extended notation yyyy-mm-dd.
	E8601DTw.d Format (p. 139)	Writes datetime values by using the ISO 8601 extended notation yyyy-mm-ddThh:mm:ss.ffffff.
	E8601DZw. Format (p. 141)	Writes datetime values for the zero meridian Coordinated Universal Time (UTC) time by using the ISO 8601 datetime and time zone extended notation yyyy-mm-ddThh:mm:ss +00:00.
	E8601LZw. Format (p. 143)	Writes time values as local time, appending the Coordinated Universal Time (UTC) offset for the local SAS

Category	Language Elements	Description
		session, using the ISO 8601 extended time notation hh:mm:ss+|−hh:mm.
	E8601TMw.d Format (p. 145)	Writes time values by using the ISO 8601 extended notation hh:mm:ss.ffffff.
	E8601TZw.d Format (p. 147)	Adjusts time values to the Coordinated Universal Time (UTC) and writes the time values by using the ISO 8601 extended notation hh:mm:ss.<fff>+|−hh:mm.
	Ew. Format (p. 149)	Writes numeric values in scientific notation.
	EUROw.d Format (p. 150)	Writes numeric values with a leading euro symbol (E), a comma that separates every three digits, and a period that separates the decimal fraction.
	EUROXw.d Format (p. 152)	Writes numeric values with a leading euro symbol (E), a period that separates every three digits, and a comma that separates the decimal fraction.
	FLOATw.d Format (p. 153)	Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the dth power.
	FRACTw. Format (p. 155)	Converts numeric values to fractions.
	HEXw. Format (p. 156)	Converts real binary (floating-point) values to hexadecimal representation.
	HHMMw.d Format (p. 157)	Writes SAS time values as hours and minutes in the form hh:mm.
	HOURw.d Format (p. 159)	Writes SAS time values as hours and decimal fractions of hours.
	IEEEw.d Format (p. 161)	Generates an IEEE floating-point value by multiplying a number by 10 raised to the dth power.
	JULIANw. Format (p. 163)	Writes SAS date values as Julian dates in the form yyddd or yyyyddd.
	MDYAMPMw.d Format (p. 164)	Writes datetime values in the form mm/dd/yy<yy> hh:mm AM|PM. The year can be either two or four digits.
	MMDDYYw. Format (p. 166)	Writes SAS date values in the form mmdd<yy>yy or mm/dd/<yy>yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	MMDDYYxw. Format (p. 168)	Writes SAS date values in the form mmdd<yy>yy or mm-dd-<yy>yy, where the x in the format name is a character that represents the special character, which separates the month, day, and year. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Category	Language Elements	Description
	MMSSw.d Format (p. 170)	Writes SAS time values as the number of minutes and seconds since midnight.
	MMYYw. Format (p. 172)	Writes SAS date values in the form mmM<yy>yy, where M is the separator and the year appears as either 2 or 4 digits.
	MMYYxw. Format (p. 173)	Writes SAS date values in the form mm<yy>yy or mm-<yy>yy, where the x in the format name is a character that represents the special character that separates the month and the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	MONNAMEw. Format (p. 175)	Writes SAS date values as the name of the month.
	MONTHw. Format (p. 177)	Writes SAS date values as the month of the year.
	MONYYw. Format (p. 178)	Writes SAS date values as the month and the year in the form mmmyy or mmmyyyy.
	NEGPARENw.d Format (p. 179)	Writes negative numeric values in parentheses.
	NENGOw. Format (p. 181)	Writes SAS date values as Japanese dates in the form e.yymmdd.
	OCTALw. Format (p. 182)	Converts numeric values to octal representation.
	PERCENTw.d Format (p. 184)	Writes numeric values as percentages.
	PERCENTNw.d Format (p. 185)	Produces percentages, using a minus sign for negative values.
	QTRw. Format (p. 187)	Writes SAS date values as the quarter of the year.
	QTRRw. Format (p. 188)	Writes SAS date values as the quarter of the year in Roman numerals.
	ROMANw. Format (p. 189)	Writes numeric values as roman numerals.
	TIMEw.d Format (p. 190)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss.
	TIMEAMPMw.d Format (p. 192)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM.
	TODw.d Format (p. 194)	Writes SAS time values and the time portion of SAS datetime values in the form hh:mm:ss.ss.
	VAXRBw.d Format (p. 196)	Writes real binary (floating-point) data in VMS format.
	w.d Format (p. 197)	Writes standard numeric data one digit per byte.

Category	Language Elements	Description
	WEEKDATEw. Format (p. 199)	Writes SAS date values as the day of the week and the date in the form day-of-week, month-name dd, yy (or yyyy).
	WEEKDATXw. Format (p. 201)	Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).
	WEEKDAYw. Format (p. 203)	Writes SAS date values as the day of the week.
	YEARw. Format (p. 204)	Writes SAS date values as the year.
	YENw.d Format (p. 205)	Writes numeric values with yen signs, commas, and decimal points.
	YYMMw. Format (p. 207)	Writes SAS date values in the form <yy>yyMmm, where M is a character separator to indicate that the month number follows the M and the year appears as either 2 or 4 digits.
	YYMMxw. Format (p. 208)	Writes SAS date values in the form <yy>yymm or <yy>yy-mm, where the x in the format name is a character that represents the special character that separates the year and the month, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	YYMMDDw. Format (p. 211)	Writes SAS date values in the form <yy>yymmdd or <yy>yy–mm–dd, where a hyphen is the separator and the year appears as either 2 or 4 digits.
	YYMMDDxw. Format (p. 213)	Writes SAS date values in the form <yy>yymmdd or <yy>yy-mm-dd, where the x in the format name is a character that represents the special character that separates the year, month, and day. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	YYMONw. Format (p. 215)	Writes SAS date values in the form yymmm or yyyymmm.
	YYQw. Format (p. 216)	Writes SAS date values in the form <yy>yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.
	YYQxw. Format (p. 218)	Writes SAS date values in the form <yy>yyq or <yy>yy-q, where the x in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	YYQRw. Format (p. 220)	Writes SAS date values in the form <yy>yyQqr, where Q is the separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in roman numerals.
	YYQRxw. Format (p. 222)	Writes SAS date values in the form <yy>yyqr or <yy>yy-qr, where the x in the format name is a character that represents the special character that separates the year

Category	Language Elements	Description
		and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits and qr is the quarter of the year expressed in roman numerals.
	YYQZw. Format (p. 224)	Writes SAS date values in the form <yy><qq>, the year appears as 2 or 4 digits, and qq is the quarter of the year.
	Zw.d Format (p. 226)	Writes standard numeric data with leading 0s.
Character	$BASE64Xw. Format (p. 75)	Converts character data into ASCII text by using Base 64 encoding.
	$BINARYw. Format (p. 76)	Converts character data to binary representation.
	$CHARw. Format (p. 77)	Writes standard character data.
	$HEXw. Format (p. 78)	Converts character data to hexadecimal representation.
	$OCTALw. Format (p. 80)	Converts character data to octal representation.
	$QUOTEw. Format (p. 81)	Writes data values that are enclosed in double quotation marks.
	$REVERJw. Format (p. 83)	Writes character data in reverse order and preserves blanks.
	$REVERSw. Format (p. 84)	Writes character data in reverse order and left aligns.
	$UPCASEw. Format (p. 85)	Converts character data to uppercase.
	$UUIDw. Format (p. 86)	Writes character data in universally unique identifier (UUID) format.
	$w. Format (p. 88)	Writes standard character data.
Date and Time	B8601DAw. Format (p. 89)	Writes date values by using the ISO 8601 basic notation yyyymmdd.
	B8601DNw. Format (p. 90)	Writes dates from datetime values by using the ISO 8601 basic notation yyyymmdd.
	B8601DTw.d Format (p. 92)	Writes datetime values by using the ISO 8601 basic notation yyyymmddThhmmss<ffffff>.
	B8601DZw. Format (p. 93)	Writes datetime values for the zero meridian Coordinated Universal Time (UTC) time by using the ISO 8601 datetime and time zone basic notation yyyymmddThhmmss+0000.
	B8601LZw. Format (p. 95)	Writes time values as local time by appending a time zone offset difference between the local time and UTC, using the ISO 8601 basic time notation hhmmss+|–hhmm.
	B8601TMw.d Format (p. 97)	Writes time values by using the ISO 8601 basic notation hhmmss<ffff>.

Category	Language Elements	Description
	B8601TZw. Format (p. 99)	Adjusts time values to the Coordinated Universal Time (UTC) and writes the time values by using the ISO 8601 basic time notation hhmmss+|−hhmm.
	DATEw. Format (p. 113)	Writes SAS date values in the form ddmmmyy, ddmmmyyyy, or dd-mmm-yyyy.
	DATEAMPMw.d Format (p. 114)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss with AM or PM.
	DATETIMEw.d Format (p. 116)	Writes SAS datetime values in the form ddmmmyy:hh:mm:ss.ss.
	DAYw. Format (p. 119)	Writes SAS date values as the day of the month.
	DDMMYYw. Format (p. 120)	Writes SAS date values in the form ddmm<yy>yy or dd/mm/<yy>yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	DDMMYYxw. Format (p. 121)	Writes SAS date values in the form ddmm<yy>yy or dd-mm-yy<yy>, where x in the format name is a character that represents the special character that separates the day, month, and year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	DOWNAMEw. Format (p. 127)	Writes SAS date values as the name of the day of the week.
	DTDATEw. Format (p. 128)	Expects a SAS datetime value as input and writes the SAS date values in the form ddmmmyy or ddmmmyyyy.
	DTMONYYw. Format (p. 130)	Writes the date part of a SAS datetime value as the month and year in the form mmmyy or mmmyyyy.
	DTWKDATXw. Format (p. 132)	Writes the date part of a SAS datetime value as the day of the week and the date in the form day-of-week, dd month-name yy (or yyyy).
	DTYEARw. Format (p. 133)	Writes the date part of a SAS datetime value as the year in the form yy or yyyy.
	DTYYQCw. Format (p. 135)	Writes the date part of a SAS datetime value as the year and the quarter and separates them with a colon (:).
	E8601DAw. Format (p. 136)	Writes date values by using the ISO 8601 extended notation yyyy-mm-dd.
	E8601DNw. Format (p. 137)	Writes dates from SAS datetime values by using the ISO 8601 extended notation yyyy-mm-dd.
	E8601DTw.d Format (p. 139)	Writes datetime values by using the ISO 8601 extended notation yyyy-mm-ddThh:mm:ss.ffffff.

Category	Language Elements	Description
	E8601DZw. Format (p. 141)	Writes datetime values for the zero meridian Coordinated Universal Time (UTC) time by using the ISO 8601 datetime and time zone extended notation yyyy-mm-ddThh:mm:ss +00:00.
	E8601LZw. Format (p. 143)	Writes time values as local time, appending the Coordinated Universal Time (UTC) offset for the local SAS session, using the ISO 8601 extended time notation hh:mm:ss+|–hh:mm.
	E8601TMw.d Format (p. 145)	Writes time values by using the ISO 8601 extended notation hh:mm:ss.ffffff.
	E8601TZw.d Format (p. 147)	Adjusts time values to the Coordinated Universal Time (UTC) and writes the time values by using the ISO 8601 extended notation hh:mm:ss.<fff>+|–hh:mm.
	HHMMw.d Format (p. 157)	Writes SAS time values as hours and minutes in the form hh:mm.
	HOURw.d Format (p. 159)	Writes SAS time values as hours and decimal fractions of hours.
	JULIANw. Format (p. 163)	Writes SAS date values as Julian dates in the form yyddd or yyyyddd.
	MDYAMPMw.d Format (p. 164)	Writes datetime values in the form mm/dd/yy<yy> hh:mm AM|PM. The year can be either two or four digits.
	MMDDYYw. Format (p. 166)	Writes SAS date values in the form mmdd<yy>yy or mm/dd/<yy>yy, where a forward slash is the separator and the year appears as either 2 or 4 digits.
	MMDDYYxw. Format (p. 168)	Writes SAS date values in the form mmdd<yy>yy or mm-dd-<yy>yy, where the x in the format name is a character that represents the special character, which separates the month, day, and year. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	MMSSw.d Format (p. 170)	Writes SAS time values as the number of minutes and seconds since midnight.
	MMYYw. Format (p. 172)	Writes SAS date values in the form mmM<yy>yy, where M is the separator and the year appears as either 2 or 4 digits.
	MMYYxw. Format (p. 173)	Writes SAS date values in the form mm<yy>yy or mm-<yy>yy, where the x in the format name is a character that represents the special character that separates the month and the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	MONNAMEw. Format (p. 175)	Writes SAS date values as the name of the month.

Category	Language Elements	Description
	MONTHw. Format (p. 177)	Writes SAS date values as the month of the year.
	MONYYw. Format (p. 178)	Writes SAS date values as the month and the year in the form mmmyy or mmmyyyy.
	NENGOw. Format (p. 181)	Writes SAS date values as Japanese dates in the form e.yymmdd.
	QTRw. Format (p. 187)	Writes SAS date values as the quarter of the year.
	QTRRw. Format (p. 188)	Writes SAS date values as the quarter of the year in Roman numerals.
	TIMEw.d Format (p. 190)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss.
	TIMEAMPMw.d Format (p. 192)	Writes SAS time values as hours, minutes, and seconds in the form hh:mm:ss.ss with AM or PM.
	TODw.d Format (p. 194)	Writes SAS time values and the time portion of SAS datetime values in the form hh:mm:ss.ss.
	WEEKDATEw. Format (p. 199)	Writes SAS date values as the day of the week and the date in the form day-of-week, month-name dd, yy (or yyyy).
	WEEKDATXw. Format (p. 201)	Writes SAS date values as the day of the week and date in the form day-of-week, dd month-name yy (or yyyy).
	WEEKDAYw. Format (p. 203)	Writes SAS date values as the day of the week.
	YEARw. Format (p. 204)	Writes SAS date values as the year.
	YYMMw. Format (p. 207)	Writes SAS date values in the form <yy>yyMmm, where M is a character separator to indicate that the month number follows the M and the year appears as either 2 or 4 digits.
	YYMMxw. Format (p. 208)	Writes SAS date values in the form <yy>yymm or <yy>yy-mm, where the x in the format name is a character that represents the special character that separates the year and the month, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	YYMMDDw. Format (p. 211)	Writes SAS date values in the form <yy>yymmdd or <yy>yy–mm–dd, where a hyphen is the separator and the year appears as either 2 or 4 digits.
	YYMMDDxw. Format (p. 213)	Writes SAS date values in the form <yy>yymmdd or <yy>yy-mm-dd, where the x in the format name is a character that represents the special character that separates the year, month, and day. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Category	Language Elements	Description
	YYMONw. Format (p. 215)	Writes SAS date values in the form yymmm or yyyymmm.
	YYQw. Format (p. 216)	Writes SAS date values in the form <yy>yyQq, where Q is the separator, the year appears as either 2 or 4 digits, and q is the quarter of the year.
	YYQxw. Format (p. 218)	Writes SAS date values in the form <yy>yyq or <yy>yy-q, where the x in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.
	YYQRw. Format (p. 220)	Writes SAS date values in the form <yy>yyQqr, where Q is the separator, the year appears as either 2 or 4 digits, and qr is the quarter of the year expressed in roman numerals.
	YYQRxw. Format (p. 222)	Writes SAS date values in the form <yy>yyqr or <yy>yy-qr, where the x in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits and qr is the quarter of the year expressed in roman numerals.
	YYQZw. Format (p. 224)	Writes SAS date values in the form <yy><qq>, the year appears as 2 or 4 digits, and qq is the quarter of the year.
ISO 8601	B8601DAw. Format (p. 89)	Writes date values by using the ISO 8601 basic notation yyyymmdd.
	B8601DNw. Format (p. 90)	Writes dates from datetime values by using the ISO 8601 basic notation yyyymmdd.
	B8601DTw.d Format (p. 92)	Writes datetime values by using the ISO 8601 basic notation yyyymmddThhmmss<ffffff>.
	B8601DZw. Format (p. 93)	Writes datetime values for the zero meridian Coordinated Universal Time (UTC) time by using the ISO 8601 datetime and time zone basic notation yyyymmddThhmmss+0000.
	B8601LZw. Format (p. 95)	Writes time values as local time by appending a time zone offset difference between the local time and UTC, using the ISO 8601 basic time notation hhmmss+|–hhmm.
	B8601TMw.d Format (p. 97)	Writes time values by using the ISO 8601 basic notation hhmmss<ffff>.
	B8601TZw. Format (p. 99)	Adjusts time values to the Coordinated Universal Time (UTC) and writes the time values by using the ISO 8601 basic time notation hhmmss+|–hhmm.
	E8601DAw. Format (p. 136)	Writes date values by using the ISO 8601 extended notation yyyy-mm-dd.

Category	Language Elements	Description
	E8601DNw. Format (p. 137)	Writes dates from SAS datetime values by using the ISO 8601 extended notation yyyy-mm-dd.
	E8601DTw.d Format (p. 139)	Writes datetime values by using the ISO 8601 extended notation yyyy-mm-ddThh:mm:ss.ffffff.
	E8601DZw. Format (p. 141)	Writes datetime values for the zero meridian Coordinated Universal Time (UTC) time by using the ISO 8601 datetime and time zone extended notation yyyy-mm-ddThh:mm:ss +00:00.
	E8601LZw. Format (p. 143)	Writes time values as local time, appending the Coordinated Universal Time (UTC) offset for the local SAS session, using the ISO 8601 extended time notation hh:mm:ss+|−hh:mm.
	E8601TMw.d Format (p. 145)	Writes time values by using the ISO 8601 extended notation hh:mm:ss.ffffff.
	E8601TZw.d Format (p. 147)	Adjusts time values to the Coordinated Universal Time (UTC) and writes the time values by using the ISO 8601 extended notation hh:mm:ss.<fff>+|−hh:mm.
Numeric	BESTw. Format (p. 101)	SAS chooses the best notation.
	BESTDOTXw. Format (p. 103)	Specifies that SAS choose the best notation and use a dot as a decimal separator.
	BESTDw.p Format (p. 104)	Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.
	BINARYw. Format (p. 106)	Converts numeric values to binary representation.
	COMMAw.d Format (p. 108)	Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.
	COMMAXw.d Format (p. 109)	Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.
	Dw.p Format (p. 111)	Prints numeric values, possibly with a great range of values, lining up decimal places for values of similar magnitude.
	DOLLARw.d Format (p. 124)	Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.
	DOLLARXw.d Format (p. 125)	Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.
	Ew. Format (p. 149)	Writes numeric values in scientific notation.

Category	Language Elements	Description
	EUROw.d Format (p. 150)	Writes numeric values with a leading euro symbol (E), a comma that separates every three digits, and a period that separates the decimal fraction.
	EUROXw.d Format (p. 152)	Writes numeric values with a leading euro symbol (E), a period that separates every three digits, and a comma that separates the decimal fraction.
	FLOATw.d Format (p. 153)	Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the dth power.
	FRACTw. Format (p. 155)	Converts numeric values to fractions.
	HEXw. Format (p. 156)	Converts real binary (floating-point) values to hexadecimal representation.
	IEEEw.d Format (p. 161)	Generates an IEEE floating-point value by multiplying a number by 10 raised to the dth power.
	NEGPARENw.d Format (p. 179)	Writes negative numeric values in parentheses.
	OCTALw. Format (p. 182)	Converts numeric values to octal representation.
	PERCENTw.d Format (p. 184)	Writes numeric values as percentages.
	PERCENTNw.d Format (p. 185)	Produces percentages, using a minus sign for negative values.
	ROMANw. Format (p. 189)	Writes numeric values as roman numerals.
	VAXRBw.d Format (p. 196)	Writes real binary (floating-point) data in VMS format.
	w.d Format (p. 197)	Writes standard numeric data one digit per byte.
	YENw.d Format (p. 205)	Writes numeric values with yen signs, commas, and decimal points.
	Zw.d Format (p. 226)	Writes standard numeric data with leading 0s.

Dictionary

$BASE64Xw. Format

Converts character data into ASCII text by using Base 64 encoding.

Categories: CAS
Character

Alignment: Left

Syntax

$BASE64X*w*.

Required Argument

w
specifies the width of the output field.

Default 1

Range 1–32767

Note If you specify the $BASE64X. format without a width, it uses a default width of 1, which is insufficient to produce output. No results are returned.

Details

Base 64 is an industry encoding method whose encoded characters are determined by using a positional scheme that uses only ASCII characters. Several Base 64 encoding schemes have been defined by the industry for specific uses, such as email or content masking. SAS maps positions 0 – 61 to the characters A – Z, a – z, and 0 – 9. Position 62 maps to the character +, and position 63 maps to the character /.

Here are some uses of Base 64 encoding:

- embed binary data in an XML file
- encode passwords

- encode URLs

The '=' character in the encoded results indicates that the results have been padded with zero bits. In order for the encoded characters to be decoded, the '=' must be included in the value to be decoded.

Example

The following program illustrates the $BASE64X*w*. format:

```
data _null_;
   declare varchar(30) a b c ;
   method run();
      a= put('FCA01A7993BC', $base64x64.);
      b= put('MyPassword', $base64x64.);
      c= put('www.mydomain.com/myhiddenURL', $base64x64.);
      put a= b= c=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=RkNBMDFBNzk5M0JD              b=TXlQYXNzd29yZA==              c=d3d3Lm15ZG9tYWluLmNvbS9teWhpZ
```

$BINARYw. Format

Converts character data to binary representation.

Categories: CAS
Character

Alignment: Left

Syntax

$BINARY*w*.

Required Argument

w
specifies the width of the output field.

Default The default width is calculated based on the length of the variable to be printed.

Range 1–32767

Tip Because each character value generates eight binary characters, increase the value of *w* by eight times the length of the character value.

Comparisons

The $BINARY*w*. format converts character values to binary representation. The BINARY*w*. format converts numeric values to binary representation.

Example

The following program illustrates the $BINARY*w*. format:

```
data _null_;
   declare varchar(30) a;
   method run();
      a = put ('name', $binary16.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=0110111001100001
```

See Also

Formats:

- “BINARYw. Format” on page 106

$CHARw. Format

Writes standard character data.

Categories: CAS
Character

Alignment: Left

Alias: $*w*. and $F*w*.

Syntax

$CHAR*w*.

Required Argument

w
specifies the width of the output field. You can specify a number or a column range.

Default 8 if the length of variable is undefined; otherwise, the length of the variable.

Range 1–32767

Comparisons

- The $CHAR*w*. format is identical to the $*w*. and $F*w*. formats.
- The $CHAR*w*., $F*w*., and $*w*. formats do not trim leading blanks. To trim leading blanks, use the LEFT function to left align character data before output.

Example

The following program illustrates the $CHAR*w*. format:

```
data _null_;
   declare varchar(30) a;
   method run();
      a= put ('XYZ',$char.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=XYZ
```

$HEXw. Format

Converts character data to hexadecimal representation.

Categories: CAS
Character

Alignment: Left

Syntax

$HEX*w*.

Required Argument

w
specifies the width of the output field.

Default The default width is calculated based on the length of the variable to be printed.

Range 1–32767

Note If *w* is greater than twice the length of the variable that you want to represent, $HEX*w*. pads it with blanks.

Tip Because each character value generates two hexadecimal characters, increase the value of *w* by two times the length of the character value.

Details

The $HEX*w*. format converts each character into two hexadecimal characters. Each blank counts as one character, including trailing blanks.

Comparisons

The HEX*w*. format converts real binary numbers to their hexadecimal equivalent.

Example

The following program illustrates the $HEX*w*. format:

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put ('bank', $hex.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=62616E6B
```

See Also

Formats:

- “HEXw. Format” on page 156

$OCTALw. Format

Converts character data to octal representation.

Categories: CAS
Character

Alignment: Left

Syntax

$OCTAL*w*.

Required Argument

w
specifies the width of the output field.

Default The default width is calculated based on the length of the variable to be printed.

Range 1–32767

Tip Because each character value generates three octal characters, increase the value of *w* by three times the length of the character value.

Comparisons

The $OCTAL*w*. format converts character values to the octal representation of their character codes. The OCTAL*w*. format converts numeric values to octal representation.

Example

The following program illustrates the $OCTAL*w*. format:

```
data _null_;
   declare varchar(30) a b c;
   method run();
      a=put('art', $octal9.);
      b=put('rice', $octal12.);
      c=put('bank', $octal12.);
      put a= b= c=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=141162164 b=162151143145 c=142141156153
```

See Also

Formats:

- "OCTALw. Format" on page 182

$QUOTEw. Format

Writes data values that are enclosed in double quotation marks.

Categories: CAS
Character

Alignment: Left

Syntax

$QUOTE*w*.

Required Argument

w
specifies the width of the output field.

Default 2 if the length of the variable is undefined; otherwise, the length of the variable + 2

Range 2–32767

Tip Make *w* wide enough to include the left and right quotation marks.

Details

The following list describes the output that SAS produces when you use the $QUOTE*w*. format.

- When your data value is not enclosed in quotation marks, SAS encloses the output in double quotation marks.
- When your data value is not enclosed in quotation marks, but the value contains a single quotation mark, SAS takes the following action:
  - encloses the data value in double quotation marks
  - does not change the single quotation mark.
- When your data value begins and ends with single quotation marks, and the value contains double quotation marks, SAS takes the following action:
  - encloses the data value in double quotation marks
  - duplicates the double quotation marks that are found in the data value
  - does not change the single quotation marks.
- When your data value begins and ends with single quotation marks, and the value contains two single contiguous quotation marks, SAS takes the following action:
  - encloses the value in double quotation marks
  - does not change the single quotation marks.
- When your data value begins and ends with single quotation marks, and contains both double quotation marks and single, contiguous quotation marks, SAS takes the following action:
  - encloses the value in double quotation marks
  - duplicates the double quotation marks that are found in the data value
  - does not change the single quotation marks.
- When the length of the target field is not large enough to contain the string and its quotation marks, SAS returns as much of the quoted string that fits into the field. For example, if the specified value is ABCDE and you specify $QUOTE5., then DE in the value is truncated and the result is "ABC".

  Note: An embedded quotation mark must be enclosed within additional quotation marks. If you specify a value of A"B, then $QUOTE5. truncates the B in the value and writes "A""".

Example

The following program illustrates the $QUOTE*w*. format:

```
data _null_;
   declare varchar(30) a b c d e f;
   method run();
      a=put('SAS',$quote.);
      b=put('SAS''s',$quote.);
      c=put('ad''verb',$quote16.);
      d=put('"ad" "verb"',$quote16.);
      e=put('"ad"''"verb"',$quote20.);
      f=put('deoxyribonucleotide',$quote20.);
      put a= b= c= d= e= f=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a="SAS" b="SAS's" c="ad'verb"        d="""ad"" ""verb"  e="""ad""'""verb"""
f="deoxyribonucleotid"
```

$REVERJw. Format

Writes character data in reverse order and preserves blanks.

Categories: CAS
Character

Alignment: Right

Syntax

$REVERJ*w*.

Required Argument

w
specifies the width of the output field.

Default 1 if *w* is not specified

Range 1–32767

Comparisons

The $REVERJ*w*. format is similar to the $REVERS*w*. format except that $REVERS*w*. left aligns the result by trimming all leading blanks.

Example

The following program illustrates the $REVERJ*w*. format:

```
data _null_;
   declare varchar(30) a b;
   method run();
      a=put('ABCD###',$reverj7.);
      b=put('###ABCD',$reverj7.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=###DCBA b=DCBA###
```

See Also

Formats:

- “$REVERSw. Format” on page 84

$REVERSw. Format

Writes character data in reverse order and left aligns.

Categories: CAS
Character

Alignment: Left

Syntax

$REVERS*w*.

Required Argument

w

specifies the width of the output field.

Default 1 if *w* is not specified

Range 1–32767

Comparisons

The $REVERS*w*. format is similar to the $REVERJ*w*. format except that $REVERJ*w*. does not left align the result.

Example

The following program illustrates the $REVERS*w*. format:

```
data _null_;
   declare varchar(30) a b;
   method run();
      a=put('ABCD   ',$revers7.);
      b=put('   ABCD',$revers7.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=DCBA    b=DCBA
```

See Also

Formats:

- “$REVERJw. Format” on page 83

$UPCASEw. Format

Converts character data to uppercase.

Categories: CAS
Character

Alignment: Left

Syntax

$UPCASE*w*.

Required Argument

w
specifies the width of the output field.

Default 8 if the length of the variable is undefined; otherwise, the length of the variable.

Range 1–32767

Details

Special characters, such as hyphens and other symbols, are not altered.

Example

The following program illustrates the $UPCASE*w*. format:

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put('coxe-ryan',$upcase9.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=COXE-RYAN
```

$UUIDw. Format

Writes character data in universally unique identifier (UUID) format.

Categories: CAS
Character

Alignment: Left

Syntax

$UUID*w*.

Required Argument

w
specifies the width of the output field.

Range 1–200

Details

You must provide the value that you want in hexadecimal format. Otherwise, the hexadecimal representation of the original characters is returned as the UUID.

Note: If you provide the value as a hexadecimal literal with an 'x' at the beginning, the value can be enclosed only in single quotation marks. You cannot use double quotation marks.

Example

The following program illustrates the $UUID*w*. format:

```
data _null_;
   declare char(36) x y;
   method run();
      x=put (x'1548611fb8cb6a47a721a6fd284e74f2', $uuid36.);
      put x=;
      x = inputc('1548611f-b8cb-6a47-a721-a6fd284e74f2','$uuid36.');
      y=put (x, $uuid36.);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x=1548611f-b8cb-6a47-a721-a6fd284e74f2
y=1548611f-b8cb-6a47-a721-a6fd284e74f2
```

$w. Format

Writes standard character data.

Categories: CAS
Character

Alignment: Left

Alias: $F*w*. and $CHAR*w*.

Syntax

$*w*.

Required Argument

w

specifies the width of the output field. You can specify a number or a column range.

Default 1 if the length of the variable is undefined; otherwise, the length of the variable

Range 1–32767

Comparisons

The $*w*., $F*w*., and the $CHAR*w*. formats are identical, and they do not trim leading blanks. To trim leading blanks, use the LEFT function to left align character data before output.

Example

The following program illustrates the $*w*. format:

```
data _null_;
   declare char(15) a;
   method run();
      a=put(' Cary',$5.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= Cary
```

See Also

Functions:

- "LEFT Function" on page 781

B8601DAw. Format

Writes date values by using the ISO 8601 basic notation *yyyymmdd*.

Categories: Date and Time
ISO 8601

Alignment: Left

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.2.1.1, complete representation

Syntax

B8601DA*w*.

Required Argument

w
specifies the width of the output field.

Default 10

Range 8–10

Details

The B8601DA format writes the date value by using the ISO 8601 basic date notation *yyyymmdd*:

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 0 and 31.

Example

The following example uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(20999,b8601da.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=20170629
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

B8601DNw. Format

Writes dates from datetime values by using the ISO 8601 basic notation *yyyymmdd*.

Categories: Date and Time
ISO 8601

Alignment: Left

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.2.1.1, complete representation

Syntax

B8601DN*w*.

Required Argument

w
specifies the width of the output field.

Default 10

Range 8–10

Details

The B8601DN format writes the date from a datetime value by using the ISO 8601 basic date notation *yyyymmdd*:

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 01 and 31.

Example

The following example uses the input value of 1793308532. 1793308532 is the SAS date value that corresponds to October 28, 2016.

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(1793308532,b8601dn.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=  20161028
```

See Also

“Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations” in *SAS Formats and Informats: Reference*

B8601DTw.d Format

Writes datetime values by using the ISO 8601 basic notation *yyyymmdd*T*hhmmss<ffffff>*.

Categories: Date and Time
ISO 8601

Alignment: Left

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.4.1, complete representation

Syntax

B8601DT*w.d*

Required Arguments

w
specifies the width of the output field.

Default 19

Range 15–26

d
specifies the number of digits to the right of the seconds value that represents a fraction of a second. This argument is optional.

Default 0

Range 0–6

Details

The B8601DT format writes the datetime value by using the ISO 8601 basic datetime notation *yyyymmdd*T*hhmmss<ffffff>*:

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 01 and 31.

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

ffffff
are optional fractional seconds, with a precision of up to six digits, where each digit is between 0 and 9.

Example

The following example uses the input value of 1793308532. 1793308532 is the SAS datetime value that corresponds to 9:15:32 p.m. on October 28, 2016.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(1793308532, b8601dt.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 20161028T211532
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

B8601DZw. Format

Writes datetime values for the zero meridian Coordinated Universal Time (UTC) time by using the ISO 8601 datetime and time zone basic notation *yyyymmdd*T*hhmmss*+0000.

Categories: Date and Time
ISO 8601

Alignment: Left

Supports: ISO 8601 Element 5.4.1, complete representation

Syntax

B8601DZ*w*.

Required Argument

w
specifies the width of the output field.

Default 26

Range 20–35

Details

UTC values specify a time and a time zone based on the zero meridian in Greenwich, England. The B8601DZ format writes SAS datetime values for the zero meridian date and time by using one of the following ISO 8601 basic datetime notations:

- *yyyymmdd*T*hhmmss*+0000

  Note: Use this form when *w* is large enough to support this time zone notation.

- *yyyymmdd*T*hhmmss*Z

  Note: Use this form when *w* is not large enough to support the +0000 time zone notation.

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 01 and 31.

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

+0000
indicates the UTC time for the zero meridian (Greenwich, England).

An ISO 8601 time or datetime value that specifies a time zone offset is adjusted by the number of hours and minutes that is specified in the offset. Then, the

time zone offset is processed as the time or datetime for the zero meridian (Greenwich, England). The B8601DZ format always writes the datetime value by using the zero meridian offset value of +0000. To write a datetime that uses a time zone offset other than +0000, see "B8601LZw. Format" on page 95.

Restriction: The shorter form +00 is not supported.

Z

indicates that the time is for the zero meridian (Greenwich, England) or +0000 UTC time. Z is used when the width of the format does not support the +0000 notation.

Example

The following example uses the input value of 1793308532. 1793308532 is the SAS date value that corresponds to 9:15:32 p.m. on October 28, 2016.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(1793308532,b8601dz.);
     put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=      20161028T211532+0000
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

B8601LZw. Format

Writes time values as local time by appending a time zone offset difference between the local time and UTC, using the ISO 8601 basic time notation *hhmmss*+|–*hhmm*.

Categories: Date and Time
ISO 8601

Alignment: Left

Supports: ISO 8601 Elements 5.3.3 and 5.3.4.2

Syntax

B8601LZ*w*.

Required Argument

w
specifies the width of the output field.

Default 14

Range 9–20

Details

The B8601LZ format writes time values without making any adjustments, and appends the UTC time zone offset for the local SAS session by using the ISO 8601 basic notation *hhmmss*+|–*hhmm*:

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

+|–*hhmm*
is an hour and minute signed offset from zero meridian time. Note that the offset must be +|–*hhmm* (that is, + or – and four characters).

Use + for time zones east of the zero meridian, and use – for time zones west of the zero meridian. For example, +0200 indicates a two-hour time difference to the east of the zero meridian, and –0600 indicates a six-hour time difference to the west of the zero meridian.

Restriction: The shorter form +|–*hh* is not supported.

When SAS reads a UTC time by using the B8601TZ informat, and the adjusted time is greater than 24 hours or less than 00 hours, SAS adjusts the value so that the time is between 000000 and 235959. If the B8601LZ format attempts to format a time outside this time range, the time is formatted with asterisks to indicate that the value is out of range.

Example

The following example uses the input value of 20999, which is a local time value of 054959-0400.

```
data _null_;
```

```
   declare varchar(30) a;
   method run();
      a=put(20999,b8601lz.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=   054959-0400
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

B8601TMw.d Format

Writes time values by using the ISO 8601 basic notation *hhmmss<ffff>*.

Categories: Date and Time
ISO 8601

Alignment: Left

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.3.1.1, complete representation

Syntax

B8601TM*w.d*

Required Arguments

w
specifies the width of the output field.

Default 8

Range 6–15

d
specifies the number of digits to the right of the seconds value that represents a fraction of a second. This argument is optional.

Default 0

Range 0–6

Details

The B8601TM format writes SAS time values by using the ISO 8601 basic time notation *hhmmss<ffffff>*:

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

ffffff
are optional fractional seconds, with a precision of up to six digits, where each digit is between 0 and 9.

Example

The following example uses the input value of 20999, which corresponds to a time value of 054959.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(20999,b8601tm.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=054959
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

B8601TZw. Format

Adjusts time values to the Coordinated Universal Time (UTC) and writes the time values by using the ISO 8601 basic time notation *hhmmss*+|–*hhmm*.

Categories: Date and Time
ISO 8601

Alignment: Left

Supports: ISO 8601 Elements 5.3.3 and 5.3.4

Syntax

B8601TZ*w*.

Required Argument

w
specifies the width of the output field.

Default 14

Range 9–20

Details

UTC time values specify a time and a time zone based on the zero meridian in Greenwich, England. The B8601TZ format adjusts the time value to be the time at the zero meridian and writes the time value in one of the following ISO 8601 basic time notations:

- *hhmmss*+|–*hhmm*

  Note: Use this form when *w* is large enough to support this time notation.

- *hhmmss*Z

  Note: Use this form when *w* is not large enough to support the +|–*hhmm* time zone notation.

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

+|–*hh:mm*
is an hour and minute signed offset from zero meridian time. Note that the offset must be +|–*hhmm* (that is, + or – and four characters).

Use + for time zones east of the zero meridian, and use – for time zones west of the zero meridian. For example, +0200 indicates a two-hour time difference to the east of the zero meridian, and –0600 indicates a six-hour time difference to the west of the zero meridian.

Restriction: The shorter form +|–*hh* is not supported.

Z
indicates that the time is for zero meridian (Greenwich, England) or +0000 UTC time. Z is used when the width of the format does not support the +0000 notation.

When SAS reads a UTC time by using the B8601TZ informat, and the adjusted time is greater than 24 hours or less than 00 hours, SAS adjusts the value so that the time is between 000000 and 240000. If the B8601TZ format attempts to format a time outside this time range, the time is formatted with asterisks to indicate that the value is out of range.

Comparisons

- For time values between 000000 and 240000, the B8601TZ format adjusts the time value to be the time at the zero meridian and writes the time value in the international standard extended time notation.
- The B8601LZ format makes no adjustment to the time and writes time values in the international standard extended time notation, using a UTC time zone offset for the local SAS session.

Example

The following example uses the input value of 20999, which corresponds to a time value of 054959+0000.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(20999,b8601tz.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=   054959+000
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

BESTw. Format

SAS chooses the best notation.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

BEST*w*.

Required Argument

w
specifies the width of the output field.

Default 12

Range 1–32

Tip If you print numbers between 0 and .01 exclusively, then use a field width of at least 7 to avoid excessive rounding. If you print numbers between 0 and –.01 exclusively, use a field width of at least 8.

Details

The BEST*w*. format is the default format for writing numeric values. When there is no format specification, SAS chooses the format that provides the most information about the value according to the available field width. BEST*w*. rounds the value, and if SAS can display at least one significant digit in the decimal portion,

within the width specified, BEST*w*. produces the result in decimal. Otherwise, it produces the result in scientific notation. SAS always stores the complete value regardless of the format that you use to represent it.

Comparisons

- The BEST*w*. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.
- The BESTDOTX*w*. format writes as many significant digits as possible in the output field. Integers are printed with a decimal.
- The D*w.p* format writes numbers with the desired precision and more alignment than the BEST*w*. format.
- The *w.d* format aligns decimal points, if possible, but does not necessarily show the same precision for all numbers.

Example

The following program illustrates the BEST*w*. format:"

```
data _null_;
   declare varchar(30) a b;
   method run();
      a=put(1257000,best6.);
      b=put(1257000,best3.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=1.26E6 b=1E6
```

See Also

Formats:

- "BESTDw.p Format" on page 104
- "BESTDOTXw. Format" on page 103
- "Dw.p Format" on page 111
- "w.d Format" on page 197

BESTDOTXw. Format

Specifies that SAS choose the best notation and use a dot as a decimal separator.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

BESTDOTX*w*.

Required Argument

w
specifies the width of the output field.

Default 12

Range 1–32

Tip If you print numbers between 0 and .01 exclusively, then use a field width of at least 7 to avoid excessive rounding. If you print numbers between 0 and –.01 exclusively, use a field width of at least 8.

Details

If the NLDECSEPARATOR system option is disabled, the BEST*w*. and BESTDOTX*w*. formats process data the same way. If the NLDECSEPARATOR system option is enabled, then the results from the BEST*w*. and BESTDOTX*w*. formats are different. See the following table to understand the differences:

LOCALE option	Default decimal separator character for the locale	NLDECSEPARATOR option	Separator character used by BEST*w*.	Separator character used by BESTDOTX*w*.
en_US	Dot	Disabled (default)	Dot	Dot

LOCALE option	Default decimal separator character for the locale	NLDECSEPARATOR option	Separator character used by BEST*w.*	Separator character used by BESTDOTX*w.*
		Enabled	Dot	Dot
fr_FR	Comma	Disabled (default)	Dot	Dot
		Enabled	Comma	Dot

For more information about the NLDECSEPARATOR system option, see *SAS National Language Support (NLS): Reference Guide*.

Comparisons

- The BESTDOTX*w* format writes as many significant digits as possible in the output field. Integers are printed with a decimal.
- The BEST*w*. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.

See Also

Formats:

- "BESTw. Format" on page 101

BESTDw.p Format

Prints numeric values, lining up decimal places for values of similar magnitude, and prints integers without decimals.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

BESTD*w*.[*p*]

Required Argument

w
specifies the width of the output field.

Default 12

Range 1–32

Optional Argument

p
specifies the precision.

Default 3

Range 0 to *w*–1

Requirement must be less than *w*

Tip If *p* is omitted or is specified as 0, then *p* is set to 3.

Details

The BESTD*w.p* format writes numbers so that the decimal point aligns in groups of values with similar magnitude. Integers are printed without a decimal point. Larger values of *p* print the data values with more precision and potentially more shifts in the decimal point alignment. Smaller values of *p* print the data values with less precision and a greater chance of decimal point alignment.

The format chooses the number of decimal places to be printed for ranges of values, even when the underlying values can be represented with fewer decimal places.

Comparisons

- The BEST*w*. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up. Integers are printed without a decimal.
- The BESTD*w.p* format is a combination of the BEST*w*. format and the D*w.p* format in that it formats all numeric data, and it does a better job of aligning decimals than the BEST*w*. format.

- The D*w.p* format writes numbers with the desired precision and more alignment than the BEST*w* format.
- The *w.d* format aligns decimal points, if possible, but it does not necessarily show the same precision for all numbers.

Example

The following program illustrates the BESTD*w.p* format:

```
data _null_;
   declare varchar(30) a b c d e;
   method run();
      a=put(12345, bestd14.);
      b=put(123.45, bestd14.);
      c=put(1.2345, bestd14.);
      d=put(.12345, bestd14.);
      e=put(1.23456789, bestd14.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=         12345 b=   123.4500000 c=     1.2345000 d=     0.1234500 e=     1.2345679
```

See Also

Formats:

- “BESTw. Format” on page 101
- “Dw.p Format” on page 111
- “w.d Format” on page 197

BINARYw. Format

Converts numeric values to binary representation.

Categories: CAS
Numeric

Alignment: Left

Syntax

BINARY*w*.

Required Argument

w
specifies the width of the output field.

Default 8

Range 1–64

Comparisons

BINARY*w*. converts numeric values to binary representation. The $BINARY*w*. format converts character values to binary representation.

Example

The following program illustrates the BINARY*w*. format:

```
data _null_;
   declare varchar(30) a b c;
   method run();
      a = put (123.45, binary8.);
      b = put (123, binary8.);
      c = put (-123, binary8.);
      put a= b= c=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=01111011 b=01111011 c=10000101
```

See Also

Formats:

- “$BINARYw. Format” on page 76

COMMAw.d Format

Writes numeric values with a comma that separates every three digits and a period that separates the decimal fraction.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

COMMA*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 6

Range 1–32

Tip Make *w* wide enough to write the numeric values, the commas, and the optional decimal point.

Optional Argument

d
specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Comparisons

- The COMMA*w.d* format is similar to the COMMAX*w.d* format, but the COMMAX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The COMMA*w.d* format is similar to the DOLLAR*w.d* format except that the COMMA*w.d* format does not print a leading dollar sign.

Example

The following program illustrates the COMMA*w*. format:

```
data _null_;
   declare varchar(30) a b;
   method run();
      a=put (23451.23,comma10.2);
      b=put (123451.234,comma10.2);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 23,451.23 b=123,451.23
```

See Also

Formats:

- “COMMAXw.d Format” on page 109
- “DOLLARw.d Format” on page 124

COMMAXw.d Format

Writes numeric values with a period that separates every three digits and a comma that separates the decimal fraction.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more

information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

COMMAX[*w*.*d*]

Required Argument

w

specifies the width of the output field.

Default 6

Range 1–32

Tip Make *w* wide enough to write the numeric values, the commas, and the optional decimal point.

Optional Argument

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Comparisons

The COMMA*w.d* format is similar to the COMMAX*w.d* format, but the COMMAX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.

Example

The following program illustrates the COMMAX*w*. format:

```
data _null_;
   declare varchar(15) a b;
   method run();
      a=put (23451.23,commax10.2);
      b=put (123451.234,commax10.2);
      put a= b=;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log.

```
a= 23.451,23 b=123.451,23
```

See Also

Formats:

- “COMMAw.d Format” on page 108

Dw.p Format

Prints numeric values, possibly with a great range of values, lining up decimal places for values of similar magnitude.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see “DECIMALCONV= System Option” in *SAS System Options: Reference*.

Syntax

D[*w*.*p*]

Required Arguments

w
specifies the width of the output field.

Default 12

Range 1–32

p
specifies the significant digits.

Default 3

Range 0–16

Requirement must be less than *w*

Details

The D*w.p* format writes numbers so that the decimal point aligns in groups of values with similar magnitude. Larger values of *p* print the data values with more precision and potentially more shifts in the decimal point alignment. Smaller values of *p* print the data values with less precision and a greater chance of decimal point alignment.

Comparisons

- The BEST*w*. format writes as many significant digits as possible in the output field, but if the numbers vary in magnitude, the decimal points do not line up.
- D*w.p* writes numbers with the desired precision and more alignment than BEST*w*. format

Example

The following program illustrates the D*w.p* format:

```
data _null_;
   declare char(15) a b c d e f;
   method run();
      a=put (12345,d10.4);
      b=put (1234.5,d10.4);
      c=put (123.45,d10.4);
      d=put (12.345,d10.4);
      e=put (1.2345,d10.4);
      f=put (.12345,d10.4);
      put a= b= c= d= e= f=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=   12345.0      b=    1234.5      c= 123.45000
d=  12.34500      e=   1.23450      f=0.12345
```

See Also

Formats:

- “BESTw. Format” on page 101

DATEw. Format

Writes SAS date values in the form *ddmmmyy*, *ddmmmyyyy*, or *dd-mmm-yyyy*.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DATE*w*.

Required Argument

w
specifies the width of the output field.

Default 7

Range 5–11

Tip Use a width of 9 to print a 4-digit year without a separator between the day, month, and year. Use a width of 11 to print a 4-digit year using a hyphen as a separator between the day, month, and year.

Details

The DATE*w*. format writes SAS date values in the form *ddmmmyy*, *ddmmmyyyy*, or *dd-mmm-yyyy*. Here is an explanation of the syntax:

dd
is an integer that represents the day of the month.

mmm
is the first three letters of the month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(15) a b c d e;
   method run();
      a=put (20999,date5.);
      b=put (20999,date6.);
      c=put (20999,date7.);
      d=put (20999,date8.);
      e=put (20999,date9.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=29JUN b= 29JUN c=29JUN17 d= 29JUN17 e=29JUN2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "DATE Function" on page 500

DATEAMPMw.d Format

Writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss* with AM or PM.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

DATEAMPM*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 19

Range 7–40

Tip SAS requires a minimum *w* value of 13 to write AM or PM. For widths between 10 and 12, SAS writes a 24-hour clock time.

Optional Argument

d

specifies the number of digits to the right of the decimal point in the seconds value.

Range 0–39

Requirement must be less than *w*

Note If *w*–*d*< 17, SAS truncates the decimal values.

Details

The DATEAMPM*w.d* format writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss*. Here is an explanation of the syntax:

dd
is an integer that represents the day of the month.

mmm
is the first three letters of the month name.

yy
is a two-digit integer that represents the year.

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Comparisons

The DATEAMPM*w.d* format is similar to the DATETIME*w.d* format except that DATEAMPM*w.d* prints AM or PM at the end of the time.

Example

The example table uses the input value of 1823167525.18123167525 is the SAS datetime value that corresponds to 11:25:25 AM on October 9, 2017.

```
data _null_;
   declare varchar(15) a b c d e;
   method run();
      a=put (1823167525,dateampm.);
      b=put (1823167525,dateampm7.);
      c=put (1823167525,dateampm10.);
      d=put (1823167525,dateampm13.);
      e=put (1823167525,dateampm22.2);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=09OCT17:11:25:2 b=09OCT17 c=09OCT17:11 d=09OCT17:11 AM e=09OCT17:11:25:2
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATETIMEw.d Format" on page 116

DATETIMEw.d Format

Writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss*.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

DATETIME*w*.[*d*]

Required Argument

w

specifies the width of the output field.

Default 16

Range 7–40

Tip SAS requires a minimum *w* value of 16 to write a SAS datetime value with the date, hour, and seconds. Add two places to *w* and a value to *d* to return values with optional decimal fractions of seconds.

Optional Argument

d

specifies the number of digits to the right of the decimal point in the seconds value.

Range 0–39

Requirement must be less than *w*

Details

The DATETIME*w.d* format writes SAS datetime values in the form *ddmmmyy:hh:mm:ss.ss*. Here is an explanation of the syntax:

dd

is an integer that represents the day of the month.

mmm

is the first three letters of the month name.

yy

is a two-digit integer that represents the year.

hh

is an integer that represents the hour in 24-hour clock time.

mm

is an integer that represents the minutes.

ss.ss

is the number of seconds to two decimal places.

Note: If *w*–*d*<17, SAS truncates the decimal values.

Comparisons

The DATEAMPM*w.d* format is similar to the DATETIME*w.d* format except that DATEAMPM*w.d* prints AM or PM at the end of the time.

Example

The example table uses the input value of 1817623985.1817623985 is the SAS datetime value that corresponds to 7:33:05 AM on August 6, 2017.

```
data _null_;
   declare varchar(15) a b c d e f g h;
   method run();
      a=put (1817623985,datetime.);
      b=put (1817623985,datetime7.);
      c=put (1817623985,datetime12.);
      d=put (1817623985,datetime18.);
      e=put (1817623985,datetime18.1);
      f=put (1817623985,datetime19.);
      g=put (1817623985,datetime20.1);
      h=put (1817623985,datetime21.2);
      put a= b= c= d= e= f= g= h=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=06AUG17:07:33:0 b=06AUG17 c=  06AUG17:07 d=  06AUG17:07:33 e=06AUG17:07:33:0
f=06AUG2017:07:3 g=06AUG2017:07:33 h=06AUG2017:07:33
```

See Also

- “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Formats:

- “DATEw. Format” on page 113
- “DATEAMPMw.d Format” on page 114
- “TIMEw.d Format” on page 190

Functions:

- “DATETIME Function” on page 504

DAYw. Format

Writes SAS date values as the day of the month.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DAY*w*.

Required Argument

w
specifies the width of the output field.

Default 2

Range 2–32

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 28, 2017.

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(20999,day2.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=29
```

See Also

"DS2 Expressions" in *SAS DS2 Programmer's Guide*

DDMMYYw. Format

Writes SAS date values in the form *ddmm<yy>yy* or *dd/mm/<yy>yy*, where a forward slash is the separator and the year appears as either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DDMMYY*w*.

Required Argument

w
specifies the width of the output field.

Default 8

Range 2–10

Interaction When *w* has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When *w* is 7, the date appears as a two-digit year without slashes.

Details

The DDMMYY*w*. format writes SAS date values in the form *ddmm<yy>yy* or *dd/mm/<yy>yy*. Here is an explanation of the syntax:

dd
is an integer that represents the day of the month.

/
is the separator.

mm
is an integer that represents the month.

<yy>yy
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(15) a b c d e f;
   method run();
      a=put(20999,ddmmyy.);
      b=put(20999,ddmmyy5.);
      c=put(20999,ddmmyy6.);
      d=put(20999,ddmmyy7.);
      e=put(20999,ddmmyy8.);
      f=put(20999,ddmmyy10.);
      put a= b= c= d= e= f=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=29/06/17 b=29/06 c=290617 d= 290617 e=29/06/17 f=29/06/2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATEw. Format" on page 113
- "DDMMYYxw. Format" on page 121
- "MMDDYYw. Format" on page 166
- "YYMMDDw. Format" on page 211

Functions:

- "MDY Function" on page 818

DDMMYYxw. Format

Writes SAS date values in the form *ddmm<yy>yy* or *dd-mm-yy<yy>*, where *x* in the format name is a character that represents the special character that separates the day, month, and year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DDMMYY*xw*.

Required Arguments

x

identifies a separator or specifies that no separator appear between the day, the month, and the year. Here are the valid values:

B
separates with a blank

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a slash.

w

specifies the width of the output field.

Default 8

Range 2–10

Interactions When *w* has a value of from 2 to 5, the date appears with as much of the day and the month as possible. When *w* is 7, the date appears as a two-digit year without separators.

When *x* has a value of N, the width range changes to 2–8.

Details

The DDMMYY*xw*. format writes SAS date values in the form *ddmm*<*yy*>*yy* or *dd*X*mm*X<*yy*>*yy*. Here is an explanation of the syntax:

dd
is an integer that represents the day of the month.

X
is a specified separator.

mm
is an integer that represents the month.

<yy>yy
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(15) a b c d;
   method run();
      a=put(20999,ddmmyyc5.);
      b=put(20999,ddmmyyd8.);
      c=put(20999,ddmmyyn8.);
      d=put(20999,ddmmyyp10.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=29:06 b=29-06-17 c=29062017 d=29.06.2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATEw. Format" on page 113
- "DDMMYYw. Format" on page 120
- "MMDDYYxw. Format" on page 168
- "YYMMDDxw. Format" on page 213

Functions:

- "DAY Function" on page 505
- "MDY Function" on page 818
- "MONTH Function" on page 834
- "YEAR Function" on page 1185

DOLLARw.d Format

Writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

DOLLAR*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 6

Range 2–32

Optional Argument

d
specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Details

The DOLLAR*w.d* format writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character ($) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these

codes represent might be different in other countries, but DOLLAR*w.d* always produces one of these codes.

Comparisons

- The DOLLAR*w.d* format is similar to the DOLLARX*w.d* format, but the DOLLARX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The DOLLAR*w.d* format is the same as the COMMA*w.d* format except that the COMMA*w.d* format does not write a leading dollar sign.

Example

The following program illustrates the DOLLAR*w.d* format:

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(1254.71,dollar10.2);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= $1,254.71
```

See Also

Formats:

- “COMMAw.d Format” on page 108
- “DOLLARXw.d Format” on page 125

DOLLARXw.d Format

Writes numeric values with a leading dollar sign, a period that separates every three digits, and a comma that separates the decimal fraction.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

DOLLARX*w*.[*d*]

Required Argument

w

specifies the width of the output field.

Default 6

Range 2–32

Optional Argument

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Requirement must be less than *w*

Details

The DOLLARX*w.d* format writes numeric values with a leading dollar sign, a comma that separates every three digits, and a period that separates the decimal fraction.

The hexadecimal representation of the code for the dollar sign character ($) is 5B on EBCDIC systems and 24 on ASCII systems. The monetary character that these codes represent might be different in other countries, but DOLLARX*w.d* always produces one of these codes.

Comparisons

- The DOLLARX*w.d* format is similar to the DOLLAR*w.d* format, but the DOLLARX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.

- The DOLLARX*w.d* format is the same as the COMMAX*w.d* format except that the COMMA*w.d* format does not write a leading dollar sign.

Example

The following program illustrates the DOLLARX*w.d* format:

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(1254.71,dollarx10.2);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= $1.254,71
```

See Also

Formats:

- “COMMAXw.d Format” on page 109
- “DOLLARw.d Format” on page 124

DOWNAMEw. Format

Writes SAS date values as the name of the day of the week.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DOWNAME*w*.

Required Argument

w
specifies the width of the output field.

Default 9

Range 1–32

Tip If you omit *w*, SAS prints the entire name of the day.

Details

If necessary, SAS truncates the name of the day to fit the format width. For example, the format DOWNAME2. prints the first two letters of the day name.

Example

The following program illustrates the DOWNAME*w*. format:

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(20999,downame.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= Thursday
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "WEEKDAYw. Format" on page 203

DTDATEw. Format

Expects a SAS datetime value as input and writes the SAS date values in the form *ddmmmyy* or *ddmmmyyyy*.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DTDATE*w*.

Required Argument

w
specifies the width of the output field.

Default 7

Range 5–9

Tip Use a width of 9 to print a 4-digit year.

Details

The DTDATE*w*. format writes SAS date values in the form *ddmmmyy* or *ddmmmyyyy*. Here is an explanation of the syntax:

dd
is an integer that represents the day of the month.

mmm
are the first three letters of the month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Comparisons

The DTDATE*w*. format produces the same type of output that the DATE*w*. format produces. The difference is that the DTDATE*w*. format requires a SAS datetime value.

Example

The example table uses the input value of 1823167525 and prints both a two-digit and a four-digit year for the DTDATE*w*. format. 1823167525 is the SAS datetime value that corresponds to 11:25:25 AM on October 9, 2017.

```
data _null_;
   declare varchar(15) a b;
   method run();
```

```
        a=put(1823167525,dtdate.);
        b=put(1823167525,dtdate9.);
        put a= b=;
    end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=09OCT17 b=09OCT2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATEw. Format" on page 113

DTMONYYw. Format

Writes the date part of a SAS datetime value as the month and year in the form *mmmyy* or *mmmyyyy*.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DTMONYY*w*.

Required Argument

w
specifies the width of the output field.

Default 5

Range 5–7

Details

The DTMONYY*w*. format writes SAS datetime values in the form *mmmyy* or *mmmyyyy*. Here is an explanation of the syntax:

mmm
is the first three letters of the month name.

yy or *yyyy*
is a two–digit or four-digit integer that represents the year.

Comparisons

The DTMONYY*w*. format and the MONYY*w*. format are similar in that they both write date values. The difference is that DTMONYY*w*. expects a SAS datetime value as input, and MONYY*w*. expects a SAS date value.

Example

The example table uses as input the value 1823167525. 1823167525 is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 AM.

```
data _null_;
   declare varchar(15) a b c d;
   method run();
      a=put(1823167525,dtmonyy.);
      b=put(1823167525,dtmonyy5.);
      c=put(1823167525,dtmonyy6.);
      d=put(1823167525,dtmonyy7.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=OCT17 b=OCT17 c= OCT17 d=OCT2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATETIMEw.d Format" on page 116
- "MONYYw. Format" on page 178

DTWKDATXw. Format

Writes the date part of a SAS datetime value as the day of the week and the date in the form *day-of-week*, *dd month-name yy* (or *yyyy*).

Categories: CAS
Date and Time

Alignment: Right

Syntax

DTWKDATX*w*.

Required Argument

w
specifies the width of the output field.

Default 29

Range 3–37

Details

The DTWKDATX*w*. format writes SAS date values in the form *day-of-week*, *dd*, *month-name*, *yy*, or *yyyy*. Here is an explanation of the syntax:

day-of-week
is either the first three letters of the day name or the entire day name.

dd
is an integer that represents the day of the month.

month-name
is either the first three letters of the month name or the entire month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Comparisons

The DTWKDATX*w*. format is similar to the WEEKDATX*w*. format in that they both write date values. The difference is that DTWKDATX*w*. expects a SAS datetime value as input, and WEEKDATX*w*. expects a SAS date value.

Example

The example table uses as input the value 1823167525. 1823167525 is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 a.m.

```
data _null_;
   declare varchar(30) a b c d;
   method run();
      a=put(1823167525,dtwkdatx.);
      b=put(1823167525,dtwkdatx3.);
      c=put(1823167525,dtwkdatx8.);
      d=put(1823167525,dtwkdatx25.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=       Monday, 9 October 2017 b=Mon c=     Mon d=       Monday, 9 Oct 2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATETIMEw.d Format" on page 116
- "WEEKDATXw. Format" on page 201

DTYEARw. Format

Writes the date part of a SAS datetime value as the year in the form *yy* or *yyyy*.

Categories: CAS
Date and Time

Alignment: Right

Syntax

DTYEAR*w*.

Required Argument

w

specifies the width of the output field.

Default 4

Range 2–4

Comparisons

The DTYEAR*w*. format is similar to the YEAR*w*. format in that they both write date values. The difference is that DTYEAR*w*. expects a SAS datetime value as input, and YEAR*w*. expects a SAS date value.

Example

The example table uses as input the value 1823167525.1823167525 is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 a.m.

```
data _null_;
   declare varchar(15) a b c d;
   method run();
      a=put(1823167525,dtyear.);
      b=put(1823167525,dtyear2.);
      c=put(1823167525,dtyear3.);
      d=put(1823167525,dtyear4.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2017 b=17 c= 17 d=2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATETIMEw.d Format" on page 116
- "YEARw. Format" on page 204

DTYYQCw. Format

Writes the date part of a SAS datetime value as the year and the quarter and separates them with a colon (:).

Categories: CAS
Date and Time

Alignment: Right

Syntax

DTYYQC*w*.

Required Argument

w
specifies the width of the output field.

Default 4

Range 4–6

Details

The DTYYQC*w*. format writes SAS datetime values in the form *yy* or *yyyy*, followed by a colon (:) and the numeric value for the quarter of the year.

Example

The example table uses as input the value 1823167525. 1823167525 is the SAS datetime value that corresponds to October 9, 2017, at 11:25:25 p.m.

```
data _null_;
   declare varchar(15) a b c d;
   method run();
      a=put(1823167525,dtyyqc.);
      b=put(1823167525,dtyyqc4.);
      c=put(1823167525,dtyyqc5.);
      d=put(1823167525,dtyyqc6.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17:4 b=17:4 c= 17:4 d=2017:4
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATETIMEw.d Format" on page 116

E8601DAw. Format

Writes date values by using the ISO 8601 extended notation *yyyy-mm-dd*.

Categories: CAS
Date and Time
ISO 8601

Alignment: Left

Alias: IS8601DA*w*.

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.2.1.1, complete representation

Syntax

E8601DA*w*.

Required Argument

w
specifies the width of the output field.

Default 10

Requirement The width of the output field must be 10.

Details

The E8601DA format writes a date by using the ISO 8601 extended notation *yyyy-mm-dd*:

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 01 and 31.

Example

The following example uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(20999,e8601da.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2017-06-29
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

E8601DNw. Format

Writes dates from SAS datetime values by using the ISO 8601 extended notation *yyyy-mm-dd*.

Categories: CAS
Date and Time
ISO 8601

Alignment: Left

Alias: IS8601DN

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.2.1.1, complete representation

Syntax

E8601DN*w*.

Required Argument

w
specifies the width of the input field.

Default 10

Requirement The width of the input field must be 10.

Details

The E8601DN format writes the date by using the ISO 8601 extended date notation *yyyy-mm-dd*:

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 01 and 31.

Example

The following example uses the input value of 1793308532. 1793308532 is the SAS date value that corresponds to October 28, 2016.

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(1793308532,e8601dn.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2016-10-28
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

E8601DTw.d Format

Writes datetime values by using the ISO 8601 extended notation *yyyy-mm-dd*T*hh:mm:ss.ffffff*.

Categories: CAS
Date and Time
ISO 8601

Alignment: Left

Alias: IS8601DT*w.d*

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.4.1, complete representation

Syntax

E8601DT*w.d*

Required Arguments

w
specifies the width of the input field.

Default 19

Range 16–26

d
specifies the number of digits to the right of the decimal point in the seconds value. This argument is optional.

Default 0

Range 0–6

Details

The E8601DT format writes datetime values by using the ISO 8601 extended datetime notation *yyyy-mm-dd*T*hh:mm:ss.ffffff*:

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 01 and 31.

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

ffffff
are optional fractional seconds, with a precision of up to six digits, where each digit is between 0 and 9.

Note: If you specify a width of 16, SAS assumes that the value for seconds is 0 and omits them from the output.

Example

The following example uses the input value of 1793308532. 1793308532 is the SAS datetime value that corresponds to 9:15:32pm on October 28, 2016.

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(1793308532,e8601dt.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2016-10-28T21:1
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

E8601DZw. Format

Writes datetime values for the zero meridian Coordinated Universal Time (UTC) time by using the ISO 8601 datetime and time zone extended notation *yyyy-mm-dd*T*hh:mm:ss*+00:00.

Categories: CAS
Date and Time
ISO 8601

Alignment: Left

Alias: IS8601DZ*w*.

Supports: ISO 8601 Element 5.4.1, complete representation

Syntax

E8601DZ*w*.

Required Argument

w
specifies the width of the output field.

Default 26

Range 20–35

Details

UTC values specify a time and a time zone based on the zero meridian in Greenwich, England. The E8601DZ format writes SAS datetime values by using one of the following ISO 8601 extended datetime notations:

- *yyyy-mm-dd*T*hh:mm:ss*+00:00

  Note: Use this form when *w* is large enough to support this time zone notation.

- *yyyy-mm-dd*T*hh:mm:ss*Z

  Note: Use this form when *w* is not large enough to support the +00:00 time zone notation.

yyyy
is a four-digit year.

mm
is a two-digit month (zero padded) between 01 and 12.

dd
is a two-digit day of the month (zero padded) between 01 and 31.

hh
is a two-digit hour (zero padded) between 00 and 24.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

+00:00
indicates that the time is for zero meridian (Greenwich, England) time.

An ISO 8601 time or datetime value that specifies a time zone offset is adjusted by the number of hours and minutes that is specified in the offset and processed as the time or datetime for the zero meridian (Greenwich, England). The E8601DZ format always writes the datetime value by using the zero meridian offset value of +00:00. To write a datetime that uses the UTC offset other than +00:00, see "E8601LZw. Format" in *SAS Formats and Informats: Reference*.

Restriction: The shorter form +00 is not supported.

Z
indicates that the time is for zero meridian (Greenwich, England) or +00:00 UTC time. Z is used when the width of the format does not support the +00:00 notation.

Example

The following example uses the input value of 1793308532. 1793308532 is the SAS date value that corresponds to 2016-10-28T21:15:32+00:00.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(1793308532,e8601dz.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 2016-10-28T21:15:32+00:00
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

E8601LZw. Format

Writes time values as local time, appending the Coordinated Universal Time (UTC) offset for the local SAS session, using the ISO 8601 extended time notation *hh:mm:ss+|–hh:mm*.

Categories: CAS
Date and Time
ISO 8601

Alignment: Left

Alias: IS8601LZ*w*.

Supports: ISO 8601 Element 5.3.1.1, complete representation

Syntax

E8601LZ*w*.

Required Argument

w
specifies the width of the output field.

Default 14

Range 9–20

Details

The E8601LZ format writes time values without making any adjustments, and appends the UTC time zone offset for the local SAS session by using one of the following ISO 8601 extended time notations:

- *hh:mm:ss+|–hh:mm*

  Note: Use this form when *w* is large enough to support this time notation.

- *hh:mm:ss*Z

Note: Use this form when *w* is not large enough to support the +|– *hh:mm* time zone notation.

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

+|–*hh:mm*
is an hour and minute signed offset from zero meridian time. Note that the offset must be +|–*hh:mm* (that is, + or – and five characters).

Use + for time zones east of the zero meridian, and use – for time zones west of the zero meridian. For example, +02:00 indicates a two-hour time difference to the east of the zero meridian, and –06:00 indicates a six-hour time difference to the west of the zero meridian.

Restriction: The shorter form +|–*hh* is not supported.

Z
indicates zero meridian (Greenwich, England) or +00:00 UTC time.

SAS writes the time value by using the form *hh:mm.ffffff*, and appends the time zone indicator +|–*hh:mm* based on the time zone offset from the zero meridian for the local SAS session, or Z. The Z time zone indicator is used for format lengths that are less than 14.

If the same time is written using both zone indicators, they indicate two different times based on the UTC. For example, if the local SAS session uses Eastern Time in the U.S., and the time value is 45824, SAS would write 12:43:44–04:00 or 12:43:44Z. The time 12:43:44–04:00 is the time 16:43:44+00:00 at the zero meridian. The Z indicates that the time is the time at the zero meridian, or 12:43:44+00:00.

When SAS reads a UTC time by using the E8601TZ informat, and the adjusted time is greater than 24 hours or less than 00 hours, SAS adjusts the value so that the time is between 00:00:00 and 24:00:00. If the E8601LZ format attempts to format a time outside of this time range, the time is formatted with asterisks to indicate that the value is out of range.

Example

The following example uses the input value of 20999, which is a local time value of 05:49:59-04:00.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(20999,e8601lz.);
      put a=;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log.

```
a=05:49:59-04:00
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

E8601TMw.d Format

Writes time values by using the ISO 8601 extended notation *hh:mm:ss.ffffff*.

Categories: CAS
Date and Time
ISO 8601

Alignment: Left

Alias: IS8601TM*w.d*

Restriction: UTC time zone offset values are not supported.

Supports: ISO 8601 Element 5.3.1.1, complete representation, and 5.3.1.3, representation of decimal fractions

Syntax

E8601TM*w.d*

Required Arguments

w

specifies the width of the output field.

Default 8

Range 8–15

d

specifies the number of digits to the right of the decimal point in the seconds value. This argument is optional.

Default 0

Range 0–6

Details

The E8601TM format writes SAS time values by using the ISO 8601 extended time notation *hh:mm:ss.ffffff*:

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

.ffffff
are optional fractional seconds, with a precision of up to six digits, where each digit is between 0 and 9.

Example

The following example uses the input value of 20999, which corresponds to a time value of 05:49:49.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(20999,e8601tm.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=05:49:59
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

E8601TZw.d Format

Adjusts time values to the Coordinated Universal Time (UTC) and writes the time values by using the ISO 8601 extended notation *hh*:*mm*:*ss.<fff>*+|–*hh*:*mm*.

Categories: CAS
Date and Time
ISO 8601

Alignment: Left

Alias: IS8601TZ*w.d*

Supports: ISO 8601 Element 5.3.1.1, complete representation

Syntax

E8601TZ*w.d*

Required Arguments

w

specifies the width of the output field.

Default 14

Range 9–20

d

specifies the number of digits to the right of the decimal point in the seconds value. This argument is optional.

Default 0

Range 0–6

Details

UTC time values specify a time and a time zone based on the zero meridian in Greenwich, England. The E8601TZ format writes time values in one of the following ISO 8601 extended time notations:

- *hh:mm:ss<.fff>*+|–*hh:mm*

  Note: Use this form when *w* is large enough to support this time zone notation.

- *hh:mm:ss*Z

  Note: Use this form when *w* is not large enough to support the +|–*hh:mm* time zone notation.

hh
is a two-digit hour (zero padded) between 00 and 23.

mm
is a two-digit minute (zero padded) between 00 and 59.

ss
is a two-digit second (zero padded) between 00 and 59.

fff
are optional fractional seconds.

+|–*hh:mm*
is an hour and minute signed offset from zero meridian time. The offset must be +|–*hh:mm* (that is, + or – and five characters).

Restriction: The shorter form +|–*hh* is not supported.

Use + for time zones east of the zero meridian, and use – for time zones west of the zero meridian. For example, +02:00 indicates a two-hour time difference to the east of the zero meridian, and –06:00 indicates a six- hour time difference to the west of the zero meridian.

Z
indicates zero meridian (Greenwich, England) or +00:00 UTC time.

When SAS reads a UTC time by using the E8601TZ informat, and the adjusted time is greater than 24 hours or less than 00 hours, SAS adjusts the value so that the time is between 00:00:00 and 24:00:00. If the E8601TZ format attempts to format a time outside of this time range, the time is formatted with asterisks to indicate that the value is out of range.

Comparisons

For time values between 00:00:00 and 24:00:00, the time value E8601TZ format adjusts the time value to be the time at the zero meridian and writes the time value in the international standard extended time notation. The E8601LZ format makes no adjustment to the time and writes time values in the international standard extended time notation, using a UTC time zone offset for the local SAS session.

Example

The following example uses the input value of 20999, which corresponds to a time value of 05:49:59+00:00.

```
data _null_;
   declare varchar(30) a;
```

```
   method run();
      a=put(20999,e8601tz.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=05:49:59+00:00
```

See Also

"Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats: Reference*

Ew. Format

Writes numeric values in scientific notation.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

E*w*.

Required Argument

w
specifies the width of the output field.

Default 12

Range 7–32

Details

SAS reserves the first column of the result for a minus sign.

Example

The example uses the input value of 1257.

```
data _null_;
   declare char(15) a b;
   method run();
      a=put(1257,e10.);
      b=put(-1257,e10.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 1.257E+03      b=-1.257E+03
```

EUROw.d Format

Writes numeric values with a leading euro symbol (E), a comma that separates every three digits, and a period that separates the decimal fraction.

Categories: CAS
Numeric

Alignment: Right

Syntax

EURO*w.d*

Required Arguments

w
specifies the width of the output field.

Default 6

Range 1-32

Tip If you want the euro symbol to be part of the output, be sure to choose an adequate width.

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0-31

Requirement must be less than *w*

Comparisons

- The EURO*w.d* format is similar to the EUROX*w.d* format, but EUROX*w.d* format reverses the roles of the decimal point and the comma. This convention is common in European countries.
- The EURO*w.d* format is similar to the DOLLAR*w.d* format, except that DOLLAR*w.d* format writes a leading dollar sign instead of the euro symbol.

Note: The EUROX*w.d* format uses the euro character (U+20AC). If you use the DBCS version of SAS and an encoding that does not support the euro character, an error occurs. To prevent this error, change your session encoding to an encoding that supports the euro character.

Example

The following program illustrates the EURO*w.d* format:

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(1254.71,euro10.2);
      b=put(1254.71,euro5.);
      c=put(1254.71,euro9.2);
      d=put(1254.71,euro15.3);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= €1,254.71      b=€1255          c=€1,254.71      d=     €1,254.710
```

See Also

Formats:

- “EUROXw.d Format” on page 152

EUROXw.d Format

Writes numeric values with a leading euro symbol (E), a period that separates every three digits, and a comma that separates the decimal fraction.

Categories: CAS
Numeric

Alignment: Right

Syntax

EUROX*w.d*

Required Arguments

w

specifies the width of the output field.

Default 6

Range 1 – 32

Tip If you want the euro symbol to be part of the output, be sure to choose an adequate width.

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0 – 31

Requirement must be less than *w*

Comparisons

- The EUROX*w.d* format is similar to the EURO*w.d* format, but EURO*w.d* format reverses the roles of the comma and the decimal point. This convention is common in English-speaking countries.
- The EUROX*w.d* format is similar to the DOLLARX*w.d* format, except that DOLLARX*w.d* format writes a leading dollar sign instead of the euro symbol.

Note: The EUROX*w.d* format uses the euro character (U+20AC). If you use the DBCS version of SAS and an encoding that does not support the euro character, an error occurs. To prevent this error, change your session encoding to an encoding that supports the euro character.

Example

The following program illustrates the EUROX*w.d* format:

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(1254.71,eurox10.2);
      b=put(1254.71,eurox5.);
      c=put(1254.71,eurox9.2);
      d=put(1254.71,eurox15.3);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= €1.254,71      b=€1255           c=€1.254,71       d=     €1.254,710
```

See Also

Formats:

- “DOLLARXw.d Format” on page 125
- “EUROw.d Format” on page 150

FLOATw.d Format

Generates a native single-precision, floating-point value by multiplying a number by 10 raised to the *d*th power.

Categories: CAS
Numeric

Alignment: Left

Syntax

FLOAT*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Requirement width must be 4

Optional Argument

d
specifies the power of 10 by which to multiply the value.

Default 0

Range 0 – 31

Details

Values that are written by FLOAT4. typically are those meant to be read by some other external program that runs in your operating environment and that expects these single-precision values. If the value that is to be formatted is a missing value, or if it is out-of-range for a native single-precision, floating-point value, a single-precision value of zero is generated.

Example

In the example below, you use the VARBINARY data type in the DECLARE statement to get a hexadecimal representation of a binary number.

Statements	Results [1]
`a=put(1,float4.);`	`0000803F`

1 The result is a hexadecimal representation of a binary number that is stored in IEEE form.

FRACTw. Format

Converts numeric values to fractions.

Categories: CAS
Numeric

Alignment: Right

Syntax

FRACT*w*.

Required Argument

w
specifies the width of the output field.

Default 10

Range 4–32

Details

Dividing the number 1 by 3 produces the value 0.33333333. To write this value as 1/3, use the FRACT*w*. format. FRACT*w*. writes fractions in reduced form, that is, 1/2 instead of 50/100.

Example

The following program illustrates the FRACT*w*. format:

```
data _null_;
   declare char(15) a b;
   method run();
      a=put(0.6666666667,fract8.);
      b=put(0.2784,fract8.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=      2/3         b= 174/625
```

HEXw. Format

Converts real binary (floating-point) values to hexadecimal representation.

Categories: CAS
Numeric

Alignment: Left

Syntax

HEX*w*.

Required Argument

w
specifies the width of the output field.

Default 8

Range 1–16

Tip If *w* < 16, the HEX*w*. format converts real binary numbers to fixed-point integers before writing them as hexadecimal characters. It also writes negative numbers in two's complement notation, and right aligns digits. If *w* is 16, HEX*w*. displays floating-point values in their hexadecimal form.

Details

In any operating environment, the least significant byte written by HEX*w*. is the rightmost byte. Some operating environments store integers with the least significant digit as the first byte. The HEX*w*. format produces consistent results in any operating environment regardless of the order of significance by byte.

Note: Different operating environments store floating-point values in different ways. However, the HEX16. format writes hexadecimal representations of floating-point values with consistent results in the same way that your operating environment stores them.

Comparisons

The HEX*w*. numeric format and the $HEX*w*. character format both generate the hexadecimal equivalent of values.

Example

The following program illustrates the HEX*w*. format:

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(35.4, hex8.);
      b=put(88, hex8.);
      c=put(2.33, hex8.);
      d=put(-150, hex8.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=00000023        b=00000058        c=00000002        d=FFFFFF6A
```

See Also

Formats:

- "$HEXw. Format" on page 78

HHMMw.d Format

Writes SAS time values as hours and minutes in the form *hh:mm*.

Categories: CAS
Date and Time

Alignment: Right

Syntax

HHMM*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 5

Range 2–20

Optional Argument

d
specifies the number of digits to the right of the decimal point in the minutes value. The digits to the right of the decimal point specify a fraction of a minute.

Default 0

Range 0–19

Requirement must be less than *w*

Details

The HHMM*w.d* format writes SAS datetime values in the form *hh:mm*. Here is an explanation of the syntax:

hh
is an integer.

mm
is the number of minutes that range from 00 through 59.

SAS rounds hours and minutes that are based on the value of seconds in a SAS time value.

Comparisons

The HHMM*w.d* format is similar to the TIME*w.d* format except that the HHMM*w.d* format does not print seconds.

The HHMM*w.d* format and the TIME*w.d* format write a leading blank for the single-hour digit. The TOD*w.d* format writes a leading zero for a single-hour digit.

Example

The example table uses the input value of 46796. 46796 is the SAS time value that corresponds to 12:59:56 p.m..

In the first example, SAS rounds up the time value four seconds based on the value of seconds in the SAS time value. In the second example, by adding a decimal specification of 2 to the format shows that fifty-six seconds is 93% of a minute.

```
data _null_;
   declare char(15) a b;
   method run();
      a=put(46796,hhmm.);
      b=put(46796,hhmm8.2);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=13:00          b=12:59.93
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "HOURw.d Format" on page 159
- "MMSSw.d Format" on page 170
- "TIMEw.d Format" on page 190
- "TODw.d Format" on page 194

Functions:

- "HMS Function" on page 676
- "HOUR Function" on page 682
- "MINUTE Function" on page 824
- "SECOND Function" on page 1071
- "TIME Function" on page 1124

HOURw.d Format

Writes SAS time values as hours and decimal fractions of hours.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

HOUR*w*.[*d*]

Required Argument

w

specifies the width of the output field.

Default 2

Range 2–20

Optional Argument

d

specifies the number of digits to the right of the decimal point in the hour value. Therefore, SAS prints decimal fractions of the hour.

Range 0-19

Requirement must be less than *w*

Details

SAS rounds hours based on the value of minutes in the SAS time value.

Example

The example table uses the input value of 41400. 41400 is the SAS time value that corresponds to 11:30 AM.

```
data _null_;
   declare char(15) a;
   method run();
      a=put(41400,hour4.1);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=11.5
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "HHMMw.d Format" on page 157
- "MMSSw.d Format" on page 170
- "TIMEw.d Format" on page 190
- "TODw.d Format" on page 194

Functions:

- "HMS Function" on page 676
- "HOUR Function" on page 682
- "MINUTE Function" on page 824
- "SECOND Function" on page 1071
- "TIME Function" on page 1124

IEEEw.d Format

Generates an IEEE floating-point value by multiplying a number by 10 raised to the *d*th power.

Categories: CAS
Numeric

Alignment: Left

CAUTION: **Large floating-point values and floating-point values that require precision might not be identical to the original SAS value when they are written to an IBM mainframe by using the IEEE format and read back into SAS using the IEEE informat.**

Syntax

IEEE*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 8

Range 3–8

Tip If *w* is 8, an IEEE double-precision, floating-point number is written. If *w* is 5, 6, or 7, an IEEE double-precision, floating-point number is written, which assumes truncation of the appropriate number of bytes. If *w* is 4, an IEEE single-precision floating-point number is written. If *w* is 3, an IEEE single-precision, floating-point number is written, which assumes truncation of one byte.

Optional Argument

d

specifies to multiply the number by 10^d.

Default 0

Range 0–10

Details

This format is useful in operating environments where IEEE*w.d* is the floating-point representation that is used. In addition, you can use the IEEE*w.d* format to create files that are used by programs in operating environments that use the IEEE floating-point representation.

Typically, programs generate IEEE values in single-precision (4 bytes) or double-precision (8 bytes). Programs perform truncation solely to save space on output files. Machine instructions require that the floating-point number be one of the two lengths. The IEEE*w.d* format allows other lengths, which enables you to write data to files that contain space-saving truncated data.

Example

In the following example, you use the VARBINARY data type in the DECLARE statement to produce results that are hexadecimal representations of binary numbers stored in IEEE form.

```
data _null_;
   declare varbinary(8) a;
   declare varchar(15) b;
   method run();
      a=put(1,ieee4.);
      b= put(a,$hex.);
      put b=;
      a=put(1,ieee5.);
      b= put(a,$hex.);
      put b=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
b=3F800000
b=3FF0000000
```

JULIANw. Format

Writes SAS date values as Julian dates in the form *yyddd* or *yyyyddd*.

Categories: CAS
Date and Time

Alignment: Left

Syntax

JULIAN*w*.

Required Argument

w
specifies the width of the output field.

Default 5

Range 5–7

Tip If *w* is 5, the JULIAN*w*. format writes the date with a two-digit year. If *w* is 7, the JULIAN*w*. format writes the date with a four-digit year.

Details

The JULIAN*w*. format writes SAS date values in the form *yyddd* or *yyyyddd*. Here is an explanation of the syntax:

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

ddd
is the number of the day, 1–365 (or 1–366 for leap years), in that year.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017, (the 180th day of the year).

```
data _null_;
   declare char(30) a b;
   method run();
      a=put(20999,julian5.);
      b=put(20999,julian7.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17180                          b=2017180
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "DATEJUL Function" on page 501

MDYAMPMw.d Format

Writes datetime values in the form *mm/dd/yy<yy> hh:mm* AM|PM. The year can be either two or four digits.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Note: The default time period is AM.

Syntax

MDYAMPM*w*.

Required Argument

w
specifies the width of the output field.

Default 19

Range 8–40

Details

The MDYAMPM*w.d* format writes SAS datetime values in the following form:

mm/*dd*/*yy*[yy] *hh*:*mm*[AM | PM]:

mm
is an integer between 1 and 12 that represents the month.

dd
is an integer between 1 and 31 that represents the day of the month.

yy or *yyyy*
specifies a two-digit or four-digit integer that represents the year.

hh
is an integer between 00 and 23 that represents hours.

mm
is an integer between 00 and 59 that represents minutes.

AM | PM
specifies either the time period 00:01–12:00 noon (PM) or the time period 12:01–12:00 midnight (AM). The default is AM.

date and time separator characters
is one of several special characters, such as the slash (/), colon (:), or a blank character that SAS uses to separate date and time components.

Comparisons

The MDYAMPM*w.* format writes datetime values with separators in the form *mm/dd/yy<yy> hh:mm* AM | PM, and requires a space between the date and the time.

The DATETIME*w.d* format writes datetime values with separators in the form *ddmmmyy<yy>: hh:mm:ss.ss*.

Example

This example uses the input value of 1823167525. 1823167525 is the SAS datetime value that corresponds to 11:25:25 AM on October 9, 2107.

```
data _null_;
   declare char(30) a;
   method run();
      a=put(1823167525,mdyampm18.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=  10/9/17 11:25 AM
```

See Also

Formats:

- “DATETIMEw.d Format” on page 116

MMDDYYw. Format

Writes SAS date values in the form *mmdd<yy>yy* or *mm/dd/<yy>yy*, where a forward slash is the separator and the year appears as either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

MMDDYY*w*.

Required Argument

w
specifies the width of the output field.

Default 8

Range 2–10

Interaction When *w* has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When *w* is 7, the date appears as a two-digit year without slashes.

Details

The MMDDYY*w*. format writes SAS date values in the form *mmdd<yy>yy* or *mm/dd/<yy>yy*. Here is an explanation of the syntax:

mm
is an integer that represents the month.

/
is the separator.

dd
is an integer that represents the day of the month.

<yy>yy
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(20999,mmddyy5.);
      b=put(20999,mmddyy8.);
      c=put(20999,mmddyy8.);
      d=put(20999,mmddyy10.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=06/29          b=06/29/17       c=06/29/17       d=06/29/2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATEw. Format" on page 113
- "DDMMYYw. Format" on page 120
- "MMDDYYxw. Format" on page 168
- "YYMMDDw. Format" on page 211

Functions:

- "DAY Function" on page 505
- "MDY Function" on page 818
- "MONTH Function" on page 834
- "YEAR Function" on page 1185

MMDDYYxw. Format

Writes SAS date values in the form *mmdd<yy>yy* or *mm-dd-<yy>yy*, where the *x* in the format name is a character that represents the special character, which separates the month, day, and year. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

MMDDYY*xw*.

Required Arguments

x

identifies a separator or specifies that no separator appear between the month, the day, and the year. Here are the valid values:

B
separates with a blank

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a slash.

w

specifies the width of the output field.

Default 8

Range 2–10

Interactions When *w* has a value of from 2 to 5, the date appears with as much of the month and the day as possible. When *w* is 7, the date appears as a two-digit year without separators.

When *x* has a value of N, the width range changes to 2–8.

Details

The MMDDYY*xw*. format writes SAS date values in the form *mmdd*<*yy*>*yy* or *mm*X*dd*X<*yy*>*yy*. Here is an explanation of the syntax:

mm
is an integer that represents the month.

X
is a specified separator.

dd
is an integer that represents the day of the month.

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(20999,mmddyyc5.);
      b=put(20999,mmddyyd8.);
      c=put(20999,mmddyyn8.);
      d=put(20999,mmddyyp10.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=06:29          b=06-29-17        c=06292017        d=06.29.2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATEw. Format" on page 113
- "DDMMYYxw. Format" on page 121
- "MMDDYYxw. Format" on page 168
- "YYMMDDxw. Format" on page 213

Functions:

- "DAY Function" on page 505
- "MDY Function" on page 818
- "MONTH Function" on page 834
- "YEAR Function" on page 1185

MMSSw.d Format

Writes SAS time values as the number of minutes and seconds since midnight.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

MMSS*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 5

Range 2–20

Tip Set *w* to a minimum of 5 to write a value that represents minutes and seconds.

Optional Argument

d

specifies the number of digits to the right of the decimal point in the seconds value. Therefore, the SAS time value includes fractional seconds.

Range 0–19

Restriction must be less than *w*

Example

The following program illustrates the MMSS*w*. format:

```
data _null_;
   declare char(15) a;
   method run();
      a=put(4530,mmss.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=75:30
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "HHMMw.d Format" on page 157
- "TIMEw.d Format" on page 190

Functions:

- "HMS Function" on page 676
- "MINUTE Function" on page 824
- "SECOND Function" on page 1071

MMYYw. Format

Writes SAS date values in the form *mm*M<*yy*>*yy*, where M is the separator and the year appears as either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

MMYY*w*.

Required Argument

w
specifies the width of the output field.

Default 7

Range 5–32

Interaction When *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

Details

The MMYY*w*. format writes SAS date values in the form *mm*M<*yy*>*yy*. Here is an explanation of the syntax:

mm
is an integer that represents the month.

M
is the character separator.

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a b c d e;
   method run();
      a=put(20999,mmyy5.);
      b=put(20999,mmyy6.);
      c=put(20999,mmyy.);
      d=put(20999,mmyy7.);
      e=put(20999,mmyy10.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=06M17          b= 06M17          c=06M2017         d=06M2017         e=   06M2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "MMYYxw. Format" on page 173
- "YYMMw. Format" on page 207

MMYYxw. Format

Writes SAS date values in the form *mm*<*yy*>*yy* or *mm*-<*yy*>*yy*, where the *x* in the format name is a character that represents the special character that separates the month and the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

MMYY*xw*.

Required Arguments

x

identifies a separator or specifies that no separator appear between the month and the year. Here are the valid values:

C

separates with a colon

D

separates with a hyphen

N

indicates no separator

P

separates with a period

S

separates with a forward slash.

w

specifies the width of the output field.

Default 7

Range 5–32

Interactions When *x* is set to N, no separator is specified. The width range is then 4–32, and the default changes to 6.

When *x* has a value of C, D, P, or S and *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

When *x* has a value of *N* and *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *x* has a value of *N* and *w* is 6 or more, the date appears with a four-digit year.

Details

The MMYY*xw*. format writes SAS date values in the form *mm*<*yy*>*yy* or *mm*X<*yy*>*yy*. Here is an explanation of the syntax:

mm

is an integer that represents the month.

X

is a specified separator.

<*yy*>*yy*

is a two-digit or four-digit integer that represents the year.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a b c d e;
   method run();
      a=put(20999,mmyyc5.);
      b=put(20999,mmyyd7.);
      c=put(20999,mmyyn4.);
      d=put(20999,mmyyp8.);
      e=put(20999,mmyys10.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=06:17          b=06-2017        c=0617           d= 06.2017       e=   06/2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "MMYYw. Format" on page 172
- "YYMMw. Format" on page 207

MONNAMEw. Format

Writes SAS date values as the name of the month.

Categories: CAS
Date and Time

Alignment: Right

Syntax

MONNAME*w*.

Required Argument

w

specifies the width of the output field.

Default 9

Range 1–32

Tip Use MONNAME3. to print the first three letters of the month name.

Details

If necessary, SAS truncates the name of the month to fit the format width.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a b c;
   method run();
      a=put(20999,monname1.);
      b=put(20999,monname3.);
      c=put(20999,monname5.);
      put a= b= c=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=J              b=Jun            c= June
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "MONTHw. Format" on page 177

MONTHw. Format

Writes SAS date values as the month of the year.

Categories: CAS
Date and Time

Alignment: Right

Syntax

MONTH*w*.

Required Argument

w
specifies the width of the output field.

Default 2

Range 1–32

Details

The MONTH*w*. format writes the month (1 through 12) of the year from a SAS date value.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a;
   method run();
      a=put(20999,month.);
      put a= ;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 6
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "MONNAMEw. Format" on page 175

MONYYw. Format

Writes SAS date values as the month and the year in the form *mmmyy* or *mmmyyyy*.

Categories: CAS
Date and Time

Alignment: Right

Syntax

MONYY*w*.

Required Argument

w
specifies the width of the output field.

Default 5

Range 5–7

Details

The MONYY*w*. format writes SAS date values in the form *mmmyy* or *mmmyyyy*. Here is an explanation of the syntax:

mmm
is the first three letters of the month name.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

Comparisons

The MONYY*w*. format and the DTMONYY*w*. format are similar in that they both write date values. The difference is that MONYY*w*. expects a SAS date value as input, and DTMONYY*w*. expects a datetime value.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to December 24, 2011.

```
data _null_;
   declare char(15) a b;
   method run();
      a=put(20999,monyy5.);
      b=put(20999,monyy7.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=JUN17          b=JUN2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DTMONYYw. Format" on page 130
- "DDMMYYw. Format" on page 120
- "MMDDYYw. Format" on page 166
- "YYMMDDw. Format" on page 211

Functions:

- "MONTH Function" on page 834
- "YEAR Function" on page 1185

NEGPARENw.d Format

Writes negative numeric values in parentheses.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

NEGPAREN*w*.[*d*]

Required Argument

w

specifies the width of the output field.

Default 6

Range 1–32

Optional Argument

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Details

The NEGPAREN*w.d* format attempts to right align output values. If the input value is negative, NEGPAREN*w.d* displays the output by enclosing the value in parentheses, if the field that you specify is wide enough. Otherwise, it uses a minus sign to represent the negative value. If the input value is nonnegative, NEGPAREN*w.d* displays the value with a leading and trailing blank to ensure proper column alignment. It reserves the last column for a close parenthesis even when the value is positive.

Comparisons

The NEGPARENw.d format is similar to the COMMAw.d format in that it separates every three digits of the value with a comma.

Example

The following program illustrates the NEGPARENw.d format:

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(100,negparen6.);
      b=put(1000,negparen6.);
      c=put(-200,negparen6.);
      d=put(-2000,negparen8.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=  100          b= 1,000          c= (200)          d= (2,000)
```

NENGOw. Format

Writes SAS date values as Japanese dates in the form *e.yymmdd*.

Categories: CAS
Date and Time

Alignment: Left

Syntax

NENGO*w*.

Required Argument

w
specifies the width of the output field.

Default 10

Range 2–10

Details

The NENGO*w*. format writes SAS date values in the form *e.yymmdd*. Here is an explanation of the syntax:

e
is the first letter of the name of the imperial era (Meiji, Taisho, Showa, Heisei, or Reiwa).

yy
is an integer that represents the year.

mm
is an integer that represents the month.

dd
is an integer that represents the day of the month.

If the width is too small, SAS omits the period.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2107.

```
data _null_;
   declare char(15) x1 x2 x3 x4 x5;
   method run();
      x1=put(20999,nengo3.);
      x2=put(20999,nengo6.);
      x3=put(20999,nengo8.);
      x4=put(20999,nengo9.);
      x5=put(20999,nengo10.);
      put x1= x2= x3= x4= x5=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x1=H29             x2=H29/06          x3=H.290629        x4=H29/06/29
x5=H.29/06/29
```

OCTALw. Format

Converts numeric values to octal representation.

Categories: CAS
Numeric

Alignment: Left

Syntax

OCTAL*w*.

Required Argument

w
specifies the width of the output field.

Default 3

Range 1–24

Details

If necessary, the OCTAL*w*. format converts numeric values to integers before displaying them in octal representation.

Comparisons

OCTAL*w*. converts numeric values to octal representation. The $OCTAL*w*. format converts character values to octal representation.

Example

The following program illustrates the OCTAL*w*. format:

```
data _null_;
   declare char(15) a;
   method run();
      a=put(3592, octal6.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=007010
```

See Also

Formats:

■ "$OCTALw. Format" on page 80

PERCENTw.d Format

Writes numeric values as percentages.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

PERCENT*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 6

Range 4–32

Optional Argument

d
specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Details

The PERCENT*w.d* format multiplies values by 100, formats them the same as the BEST*w.d* format, adds a percent sign (%) to the end of the formatted value, and encloses negative values in parentheses. The PERCENT*w.d* format allows room for a percent sign and parentheses, even if the value is not negative.

Example

The following program illustrates the PERCENT*w.d* format:

```
data _null_;
   declare char(15) a b c;
   method run();
      a=put(0.1,percent10.);
      b=put(1.2,percent10.);
      c=put(-.05,percent10.);
      put a= b= c=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=      10%       b=     120%       c=(      5%)
```

See Also

Formats:

- "PERCENTNw.d Format" on page 185

PERCENTNw.d Format

Produces percentages, using a minus sign for negative values.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

PERCENTN*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 6

Range 4–32

Optional Argument

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Details

The PERCENTN*w.d* format multiplies negative values by 100, formats them the same as the BEST*w.d* format, adds a minus sign to the beginning of the value, and adds a percent sign (%) to the end of the formatted value. The PERCENTN*w.d* format allows room for a percent sign and a minus sign, even if the value is not negative.

Comparisons

The PERCENTN*w.d* format produces percents by using a minus sign instead of parentheses for negative values. The PERCENT*w.d* format produces percents by using parentheses for negative values.

Example

The following program illustrates the PERCENTN*w.d* format:

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(0.1,percentn.);
      b=put(1.2,percentn.);
      c=put(-.05,percentn.);
      d=put(-6.3,percentn.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=  10%          b= 120%          c=  -5%          d=-630%
```

See Also

Formats:

- “PERCENTw.d Format” on page 184

QTRw. Format

Writes SAS date values as the quarter of the year.

Categories: CAS
Date and Time

Alignment: Right

Syntax

QTR*w*.

Required Argument

w
specifies the width of the output field.

Default 1

Range 1–32

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a;
   method run();
      a=put(20999,qtr.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "QTRRw. Format" on page 188

QTRRw. Format

Writes SAS date values as the quarter of the year in Roman numerals.

Categories: CAS
Date and Time

Alignment: Right

Syntax

QTRR*w*.

Required Argument

w
specifies the width of the output field.

Default 3

Range 3–32

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(15) a;
   method run();
      a=put(20999,qtrr.);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= II
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "QTRw. Format" on page 187

ROMANw. Format

Writes numeric values as roman numerals.

Categories: CAS
Numeric

Alignment: Left

Syntax

ROMAN*w*.

Required Argument

w
specifies the width of the output field.

Default 6

Range 2–32

Details

The ROMAN*w*. format truncates a floating-point value to its integer component before the value is written.

Example

```
data _null_;
   declare char(15) a;
```

```
      method run();
         a=put(2017,roman.);
         put a=;
      end;
   enddata;
   run;
```

SAS writes the following output to the log.

```
a=MMXVII
```

TIMEw.d Format

Writes SAS time values as hours, minutes, and seconds in the form *hh:mm:ss.ss*.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

TIME*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 8

Range 2–20

Tip Make *w* large enough to produce the desired results. To obtain a complete time value with three decimal places, you must allow at least 12 spaces: Eight spaces to the left of the decimal point, one space for the decimal point itself, and three spaces for the decimal fraction of seconds.

Optional Argument

d

specifies the number of digits to the right of the decimal point in the seconds value.

Default	0
Range	0–19
Requirement	must be less than *w*

Details

The TIME*w.d* format writes SAS time values in the form *hh:mm:ss.ss*. Here is an explanation of the syntax:

hh

is an integer.

Note: If *hh* is a single digit, TIME*w.d* places a leading blank before the digit. For example, the TIME*w.d.* format writes 9:00 instead of 09:00.

mm

is the number of minutes, ranging from 00 through 59.

ss.ss

is the number of seconds, ranging from 00 through 59, with the fraction of a second following the decimal point.

Comparisons

The TIME*w.d* format is similar to the HHMM*w.d* format except that TIME*w.d* includes seconds.

The TIME*w.d* format and the HHMM*w*write a leading blank for a single-hour digit. The TOD*w.d* format writes a leading zero for a single-hour digit.

Example

The example table uses the input value of 59083. 59083 is the SAS time value that corresponds to 4:24:43 PM.

```
data _null_;
   declare varchar(15) a;
   method run();
      a=put(59083, time.);
      put a=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=16:24:43
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "HHMMw.d Format" on page 157
- "HOURw.d Format" on page 159
- "MMSSw.d Format" on page 170
- "TODw.d Format" on page 194

Functions:

- "HOUR Function" on page 682
- "MINUTE Function" on page 824
- "SECOND Function" on page 1071
- "TIME Function" on page 1124

TIMEAMPMw.d Format

Writes SAS time values as hours, minutes, and seconds in the form *hh:mm:ss.ss* with AM or PM.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

TIMEAMPM*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 11

Range 2–20

Optional Argument

d
specifies the number of digits to the right of the decimal point in the seconds value.

Default 0

Range 0–19

Requirement must be less than *w*

Details

The TIMEAMPM*w.d* format writes SAS time values in the form *hh:mm:ss.ss* with AM or PM. Here is an explanation of the syntax:

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Times greater than 23:59:59 PM appear as the next day.

Make *w* large enough to produce the desired results. To obtain a complete time value with three decimal places and AM or PM, you must allow at least 11 spaces (*hh:mm:ss* PM). If *w* is less than 5, SAS writes AM or PM only.

Comparisons

- The TIMEAMPMM*w.d* format is similar to the TIMEM*w.d* format except, that TIMEAMPMM*w.d* prints AM or PM at the end of the time.
- TIME*w.d* writes hours greater than 23:59:59 PM, and TIMEAMPM*w.d* does not.

Example

The example table uses the input value of 59083. 59083 is the SAS time value that corresponds to 4:24:43 PM.

```
data _null_;
   declare char(15) a b c d;
   method run();
      a=put(59083,timeampm3.);
      b=put(59083,timeampm5.);
      c=put(59083,timeampm7.);
      d=put(59083,timeampm11.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= PM            b= 4 PM          c=4:24 PM        d= 4:24:43 PM
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "TIMEw.d Format" on page 190

TODw.d Format

Writes SAS time values and the time portion of SAS datetime values in the form *hh:mm:ss.ss*.

Categories: CAS
Date and Time

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

TOD*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 8

Range 2–20

Tip SAS writes a zero for a zero hour if the specified width is sufficient (for example, 02:30 or 00:30).

Optional Argument

d
specifies the number of digits to the right of the decimal point in the seconds value.

Default 0

Range 0–19

Requirement must be less than *w*

Details

The TOD*w.d* format writes SAS datetime values in the form *hh:mm:ss.ss*. Here is an explanation of the syntax:

hh
is an integer that represents the hour.

mm
is an integer that represents the minutes.

ss.ss
is the number of seconds to two decimal places.

Comparisons

The TOD*w.d* format writes a leading zero for a single-hour digit. The TIME*w.d* format and the HHMM*w.d* format write a leading blank for a single-hour digit.

Example

The following program illustrates the TOD*w.d* format:

```
data _null_;
   declare varchar(15) a b;
```

```
      method run();
         a=put(1850739623,tod9.);
         put a= ;
         b=put(1472049623, time.);
         put b=;
      end;
   enddata;
   run;
```

SAS writes the following output to the log.

```
a= 14:20:23
b=  408902
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "TIMEw.d Format" on page 190
- "TIMEAMPMw.d Format" on page 192

Functions:

- "TIMEPART Function" on page 1125

VAXRBw.d Format

Writes real binary (floating-point) data in VMS format.

Categories: CAS
Numeric

Alignment: Right

Syntax

VAXRB*w*.[*d*]

Required Argument

w
specifies the width of the output field.

Default 8

Range 2–8

Optional Argument

d

specifies the power of 10 by which to divide the value.

Default 0

Range 0–31

Details

Use the VAXRB*w.d* format to write data in native VAX/VMS floating-point notation.

Example

In the following example, you use the VARBINARY data type so that the result is the hexadecimal representation for the integer.

```
data _null_;
   declare varbinary(20) a;
   declare varchar(64) b;
   method run();
      a=put(1,vaxrb8.);
      put a=;
      b= put(a, $hex.);
      put b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=
b=8040000000000000
```

w.d Format

Writes standard numeric data one digit per byte.

Categories: CAS
Numeric

Alignment: Right

Alias: F*w.d*

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

w.[*d*]

Required Argument

w

specifies the width of the output field.

Range 1–32

Tip Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

Optional Argument

d

specifies the number of digits to the right of the decimal point in the numeric value.

Range 0–31

Requirement must be less than *w*

Tip If *d* is 0 or you omit *d*, *w.d* writes the value without a decimal point.

Details

The *w.d* format rounds to the nearest number that fits in the output field. If *w.d* is too small, SAS might shift the decimal to the BEST*w*. format. The *w.d* format writes negative numbers with leading minus signs. In addition, *w.d* right aligns before writing and pads the output with leading blanks.

Comparisons

The Z*w.d* format is similar to the *w.d* format except that Z*w.d* pads right-aligned output with 0s instead of blanks.

Example

The following program illustrates the *w.d* format:

```
data _null_;
   declare char(20) a;
   method run();
      a=put(23.45, 6.3);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=23.450
```

See Also

Formats:

- "Zw.d Format" on page 226

WEEKDATEw. Format

Writes SAS date values as the day of the week and the date in the form *day-of-week*, *month-name dd*, *yy* (or *yyyy*).

Categories: CAS
Date and Time

Alignment: Right

Syntax

WEEKDATE*w*.

Required Argument

w
specifies the width of the output field.

Default 29

Range 3–37

Details

The WEEKDATE*w*. format writes SAS date values in the form *day-of-week*, *month-name dd*, *yy* (or *yyyy*). Here is an explanation of the syntax:

dd
is an integer that represents the day of the month.

yy or *yyyy*
is a two-digit or four-digit integer that represents the year.

If *w* is too small to write the complete day of the week and month, SAS abbreviates as needed.

Comparisons

The WEEKDATE*w*. format is the same as the WEEKDATX*w*. format except that WEEKDATX*w*. prints *dd* before the month's name.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(30) a b c d;
   method run();
      a=put(20999,weekdate3.);
      b=put(20999,weekdate9.);
      c=put(20999,weekdate15.);
      d=put(20999,weekdate17.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=Thu b= Thursday c=Thu, Jun 29, 17 d=Thu, Jun 29, 2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DTWKDATXw. Format" on page 132
- "DATEw. Format" on page 113

- “DDMMYYw. Format” on page 120
- “MMDDYYw. Format” on page 166
- “TODw.d Format” on page 194
- “WEEKDATXw. Format” on page 201
- “YYMMDDw. Format” on page 211

Functions:

- “JULDATE Function” on page 765
- “MDY Function” on page 818
- “WEEKDAY Function” on page 1181

WEEKDATXw. Format

Writes SAS date values as the day of the week and date in the form *day-of-week*, *dd month-name yy* (or *yyyy*).

Categories: CAS
Date and Time

Alignment: Right

Syntax

WEEKDATX*w*.

Required Argument

w
specifies the width of the output field.

Default 29

Range 3–37

Details

The WEEKDATX*w*. format writes SAS date values in the form *day-of-week*, *dd month-name*, *yy* (or *yyyy*). Here is an explanation of the syntax:

dd
is an integer that represents the day of the month.

yy or *yyyy*
is a two-digit or a four-digit integer that represents the year.

If *w* is too small to write the complete day of the week and month, then SAS abbreviates as needed.

Comparisons

The WEEKDATE*w*. format is the same as the WEEKDATX*w*. format, except that WEEKDATE*w*. prints *dd* after the month's name.

The DTWKDATX*w*. format is the same as the WEEKDATX*w*. format, except that DTWKDATX*w*. expects a datetime value as input.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(30) a;
   method run();
      a=put(20999,weekdatx.);
      put a= ;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=       Thursday, 29 June 2017
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DTWKDATXw. Format" on page 132
- "DATEw. Format" on page 113
- "DDMMYYw. Format" on page 120
- "MMDDYYw. Format" on page 166
- "TODw.d Format" on page 194
- "WEEKDATEw. Format" on page 199
- "YYMMDDw. Format" on page 211

Functions:

- “JULDATE Function” on page 765
- “MDY Function” on page 818
- “WEEKDAY Function” on page 1181

WEEKDAYw. Format

Writes SAS date values as the day of the week.

Categories: CAS
Date and Time

Alignment: Right

Syntax

WEEKDAY*w*.

Required Argument

w
specifies the width of the output field.

Default 1

Range 1–32

Details

The WEEKDAY*w*. format writes a SAS date value as the day of the week (where 1=Sunday, 2=Monday, and so on).

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June, 29, 2017.

```
data _null_;
   declare char(20) a;
   method run();
      a=put(20999,weekday.);
      put a= ;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log.

```
a=5
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DOWNAMEw. Format" on page 127

YEARw. Format

Writes SAS date values as the year.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YEAR*w*.

Required Argument

w
specifies the width of the output field.

Default 4

Range 2–32

Tip If *w* is less than 4, the last two digits of the year print. Otherwise, the year value prints as four digits.

Comparisons

The YEAR*w*. format is similar to the DTYEAR*w*. format in that they both write date values. The difference is that YEAR*w*. expects a SAS date value as input, and DTYEAR*w*. expects a SAS datetime value.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(20) a;
   method run();
      a=put(20999,weeku.);
      put a= ;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2017-W26-05
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DTYEARw. Format" on page 133

YENw.d Format

Writes numeric values with yen signs, commas, and decimal points.

Categories: CAS
Numeric

Alignment: Right

Syntax

YEN*w.d*

Required Arguments

w

specifies the width of the output field.

Default 8

Range 1–32

d

specifies the number of digits to the right of the decimal point in the numeric value.

Restriction must be either 0 or 2

Tip If *d* is 2, then YEN*w.d* writes a decimal point and two decimal digits. If *d* is 0, then YEN*w.d* does not write a decimal point or decimal digits.

Details

The YEN*w.d* format writes numeric values with a leading yen sign and with a comma that separates every three digits of each value.

The hexadecimal representation of the code for the yen sign character is 5B on EBCDIC systems and 5C on ASCII systems. The monetary character these codes represent might be different in other countries.

Example

The following program illustrates the YEN*w.d* format:

```
data _null_;
   declare char(20) a ;
   method run();
      a=put(1254.71,yen10.2);
      put a= ;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= ¥1,254.71
```

YYMMw. Format

Writes SAS date values in the form <*yy*>*yy*M*mm*, where M is a character separator to indicate that the month number follows the M and the year appears as either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYMM*w*.

Required Argument

w
specifies the width of the output field.

Default 7

Range 5–32

Interaction When *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

Details

The YYMM*w*. format writes SAS date values in the form <*yy*>*yy*M*mm*. Here is an explanation of the syntax:

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

M
is the character separator.

mm
is an integer that represents the month.

Comparisons

- The YYMM*w.d* format is similar to the YYMM*xw.d* format, except the YYMM*xw.d* format contains a separator such as a hyphen, colon, slash, or period between the year and month.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(20) a b c d e ;
   method run();
      a=put(20999,yymm5.);
      b=put(20999,yymm6.);
      c=put(20999,yymm.);
      d=put(20999,yymm7.);
      e=put(20999,yymm10.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17M06 b= 17M06 c=2017M06 d=2017M06 e=   2017M06
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "MMYYw. Format" on page 172
- "YYMMxw. Format" on page 208

YYMMxw. Format

Writes SAS date values in the form <*yy*>*yymm* or <*yy*>*yy-mm*, where the *x* in the format name is a character that represents the special character that separates the year and the month, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Categories: CAS

Date and Time

Alignment: Right

Syntax

YYMM*xw*.

Required Arguments

x

identifies a separator or specifies that no separator appear between the year and the month. Here are the valid values:

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a forward slash

w

specifies the width of the output field.

Default 7

Range 5–32

Interactions When *x* is set to *N*, no separator is specified. The width range is then 4–32, and the default changes to 6.

When *x* has a value of C, D, P, or S and *w* has a value of 5 or 6, the date appears with only the last two digits of the year. When *w* is 7 or more, the date appears with a four-digit year.

When *x* has a value of *N* and *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *x* has a value of *N* and *w* is 6 or more, the date appears with a four-digit year.

Details

The YYMM*xw*. format writes SAS date values in the form <*yy*>*yymm* or <*yy*>*yy*X*mm*. Here is an explanation of the syntax:

<yy>yy
is a two-digit or four-digit integer that represents the year.

X
is a specified separator.

mm
is an integer that represents the month.

Comparisons

- The YYMM*w.d* format is similar to the YYMM*xw.d* format, except the YYMM*xw.d* format contains a separator such as a hyphen, colon, slash, or period, between the year and month.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(20) a b c d e;
   method run();
      a=put(20999,yymmc5.);
      b=put(20999,yymmd.);
      c=put(20999,yymmn4.);
      d=put(20999,yymmp8.);
      e=put(20999,yymms10.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17:06 b=2017-06 c=1706 d= 2017.06 e=   2017/06
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "MMYYxw. Format" on page 173
- "YYMMw. Format" on page 207

YYMMDDw. Format

Writes SAS date values in the form <*yy*>*yymmdd* or <*yy*>*yy*–*mm*–*dd*, where a hyphen is the separator and the year appears as either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYMMDD*w*.

Required Argument

w
specifies the width of the output field.

Default 8

Range 2–10

Interaction When *w* has a value of from 2 to 5, the date appears with as much of the year and the month as possible. When *w* is 7, the date appears as a two-digit year without hyphens.

Details

The YYMMDD*w*. format writes SAS date values in the form <*yy*>*yymmdd* or <*yy*>*yy*–*mm*–*dd*. Here is an explanation of the syntax:

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

–
is the separator.

mm
is an integer that represents the month.

dd
is an integer that represents the day of the month.

Comparisons

- The YYMMDD*w.d* format is similar to the YYMMDD*xw.d* format, except the YYMMDD*xw.d* format contains separators, such as a colon, slash, or period between the year, month, and day.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
ddata _null_;
   declare varchar(20) a b c d e f g h;
   method run();
      a=put(20999,yymmdd2.);
      b=put(20999,yymmdd3.);
      c=put(20999,yymmdd4.);
      d=put(20999,yymmdd5.);
      e=put(20999,yymmdd6.);
      f=put(20999,yymmdd7.);
      g=put(20999,yymmdd8.);
      h=put(20999,yymmdd10.);
      put a= b= c= d= e= f= g= h=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17 b= 17 c=1706 d=17-06 e=170629 f= 170629 g=17-06-29 h=2017-06-29
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATEw. Format" on page 113
- "DDMMYYw. Format" on page 120
- "MMDDYYw. Format" on page 166
- "YYMMDDxw. Format" on page 213

Functions:

- "DAY Function" on page 505
- "MDY Function" on page 818
- "MONTH Function" on page 834

- "YEAR Function" on page 1185

YYMMDDxw. Format

Writes SAS date values in the form <*yy*>*yymmdd* or <*yy*>*yy-mm-dd*, where the *x* in the format name is a character that represents the special character that separates the year, month, and day. The special character can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYMMDD*xw*.

Required Arguments

x
identifies a separator or specifies that no separator appear between the year, the month, and the day. Here are the valid values:

B
separates with a blank

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a slash.

w
specifies the width of the output field.

Default 8

Range 2–10

Interactions When *w* has a value of from 2 to 5, the date appears with as much of the year and the month. When *w* is 7, the date appears as a two-digit year without separators.

When *x* has a value of N, the width range is 2–8.

Details

The YYMMDD*xw*. format writes SAS date values in the form <*yy*>*yymmdd* or <*yy*>*yy*X*mm*X*dd*. Here is an explanation of the syntax:

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

X
is a specified separator.

mm
is an integer that represents the month.

dd
is an integer that represents the day of the month.

Comparisons

- The YYMMDD*w.d* format is similar to the YYMMDD*xw.d* format, but YYMMDD*xw.d* format contains a separator between the year and month, such as a colon, slash, or period.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare varchar(20) a b c d;
   method run();
      a=put(20999,yymmddc5.);
      b=put(20999,yymmddd8.);
      c=put(20999,yymmddn8.);
      d=put(20999,yymmddp10.);
      put a= b= c= d=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17:06 b=17-06-29 c=20170629 d=2017.06.29
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "DATEw. Format" on page 113
- "DDMMYYxw. Format" on page 121
- "MMDDYYxw. Format" on page 168
- "YYMMDDw. Format" on page 211

Functions:

- "DAY Function" on page 505
- "MDY Function" on page 818
- "MONTH Function" on page 834
- "YEAR Function" on page 1185

YYMONw. Format

Writes SAS date values in the form *yymmm* or *yyyymmm*.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYMON*w*.

Required Argument

w
specifies the width of the output field. If the format width is too small to print a four-digit year, only the last two digits of the year are printed.

Default 7

Range 5–32

Details

The YYMON*w*. format abbreviates the month's name to three characters.

Example

The example table uses the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(20) a ;
   method run();
      a=put(20999,yymon.);
      put a= ;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2017JUN
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "MMYYw. Format" on page 172

YYQw. Format

Writes SAS date values in the form <*yy*>*yy*Q*q*, where Q is the separator, the year appears as either 2 or 4 digits, and *q* is the quarter of the year.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYQ*w*.

Required Argument

w
specifies the width of the output field.

Default 6

Range 4–32

Interaction When *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *w* is 6 or more, the date appears with a four-digit year.

Details

The YYQ*w*. format writes SAS date values in the form <*yy*>*yy*Q*q*. Here is an explanation of the syntax:

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

Q
is the character separator.

q
is an integer (1, 2, 3, or 4) that represents the quarter of the year.

Comparisons

The YYQ*w*. format is similar to the YYQ*xw*. format, but the YYQ*xw*. format has separators between the YY and Q, such as a hyphen, slash, period, or colon.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(20) a b c d e;
   method run();
      a=put(20999,yyq4.);
      b=put(20999,yyq5.);
      c=put(20999,yyq.);
      d=put(20999,yyq6.);
      e=put(20999,yyq10.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17Q2                  b= 17Q2                 c=2017Q2
d=2017Q2                e=   2017Q2
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "YYQxw. Format" on page 218
- "YYQRw. Format" on page 220
- "YYQRxw. Format" on page 222
- "YYQZw. Format" on page 224

YYQxw. Format

Writes SAS date values in the form <*yy*>*yyq* or <*yy*>*yy*-*q*, where the *x* in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYQ*xw*.

Required Arguments

x

identifies a separator or specifies that no separator appear between the year and the quarter. Here are the valid values:

C

separates with a colon

D

separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a forward slash.

w
specifies the width of the output field.

Default 6

Range 4–32

Interactions When *x* is set to *N*, no separator is specified. The width range is then 3–32, and the default changes to 5.

When *w* has a value of 4 or 5, the date appears with only the last two digits of the year. When *w* is 6 or more, the date appears with a four-digit year.

When *x* has a value of *N* and *w* has a value of 3 or 4, the date appears with only the last two digits of the year. When *x* has a value of *N* and *w* is 5 or more, the date appears with a four-digit year.

Details

The YYQ*xw*. format writes SAS date values in the form <*yy*>*yyq* or <*yy*>*yy*X*q*. Here is an explanation of the syntax:

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

X
is a specified separator.

q
is an integer (1, 2, 3, or 4) that represents the quarter of the year.

Comparisons

The YYQ*w*. format is similar to the YYQ*xw*. format, but the YYQ*xw*. format has separators between the YY and Q, such as a hyphen, slash, period, or colon.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(20) a b c d e;
   method run();
      a=put(20999,yyqc4.);
      b=put(20999,yyqd.);
      c=put(20999,yyqn3.);
      d=put(20999,yyqp6.);
      e=put(20999,yyqs8.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=17:2                b=2017-2              c=172
d=2017.2              e= 2017/2
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "YYQw. Format" on page 216
- "YYQRw. Format" on page 220
- "YYQRxw. Format" on page 222
- "YYQZw. Format" on page 224

YYQRw. Format

Writes SAS date values in the form <*yy*>*yy*Q*qr*, where Q is the separator, the year appears as either 2 or 4 digits, and *qr* is the quarter of the year expressed in roman numerals.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYQR*w*.

Required Argument

w
specifies the width of the output field.

Default 8

Range 6–32

Interaction When the value of *w* is too small to write a four-digit year, the date appears with only the last two digits of the year.

Details

The YYQR*w*. format writes SAS date values in the form *<yy>yyQqr*. Here is an explanation of the syntax:

<yy>yy
is a two-digit or four-digit integer that represents the year.

Q
is the character separator.

qr
is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Comparisons

The YYQR*w*. format is similar to the YYQR*xw*. format, but the YYQR*xw*. format has separators between the YY and QR, such as a hyphen, slash, period, or colon.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
  declare char(20) a b c d e;
  method run();
     a=put(20999,yyqr6.);
     b=put(20999,yyqr7.);
     c=put(20999,yyqr.);
     d=put(20999,yyqr8.);
     e=put(20999,yyqr10.);
```

```
        put a= b= c= d= e=;
    end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 17QII              b=2017QII             c= 2017QII            d=
2017QII
e=  2017QII
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "YYQw. Format" on page 216
- "YYQRxw. Format" on page 222
- "YYQxw. Format" on page 218
- "YYQZw. Format" on page 224

YYQRxw. Format

Writes SAS date values in the form <*yy*>*yyqr* or <*yy*>*yy*-*qr*, where the *x* in the format name is a character that represents the special character that separates the year and the quarter of the year, which can be a hyphen (–), period (.), blank character, slash (/), colon (:), or no separator; the year can be either 2 or 4 digits and *qr* is the quarter of the year expressed in roman numerals.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYQR*xw*.

Required Arguments

x
identifies a separator or specifies that no separator appear between the year and the quarter. Here are the valid values:

C
separates with a colon

D
separates with a hyphen

N
indicates no separator

P
separates with a period

S
separates with a forward slash.

w
specifies the width of the output field.

Default 8

Range 6–32

Interactions When *x* is set to *N*, no separator is specified. The width range is then 5–32, and the default changes to 7.

When the value of *w* is too small to write a four-digit year, the date appears with only the last two digits of the year.

Details

The YYQR*xw*. format writes SAS date values in the form <*yy*>*yyqr* or <*yy*>*yy*X*qr*. Here is an explanation of the syntax:

<*yy*>*yy*
is a two-digit or four-digit integer that represents the year.

X
is a specified separator.

qr
is a roman numeral (I, II, III, or IV) that represents the quarter of the year.

Comparisons

The YYQR*w*. format is similar to the YYQR*xw*. format, but the YYQR*xw*. format has separators between the YY and QR, such as a hyphen, slash, period, or colon.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(20) a b c d e;
   method run();
      a=put(18985,yyqrc6.);
      b=put(20999,yyqrd.);
      c=put(20999,yyqrn5.);
      d=put(20999,yyqrp8.);
      e=put(20999,yyqrs10.);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 11:IV              b= 2017-II            c= 17II
d= 2017.II            e=  2017/II
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "YYQxw. Format" on page 218
- "YYQRw. Format" on page 220
- "YYQw. Format" on page 216
- "YYQZw. Format" on page 224

YYQZw. Format

Writes SAS date values in the form <*yy*><*qq*>, the year appears as 2 or 4 digits, and *qq* is the quarter of the year.

Categories: CAS
Date and Time

Alignment: Right

Syntax

YYQZ*w*.

Required Arguments

Z
specifies that no separator appear between the year and the quarter.

w
specifies the width of the output field.

Default 4

Range 4–6

Details

The YYQZ*w*. format writes SAS date values in the form <*yy*> <*qq*>. Here is an explanation of the syntax:

<*yy*>
is a two-digit or four-digit integer that represents the year.

Z
specifies that there is no separator.

<*qq*>
is an integer (01, 02, 03, or 04) that represents the quarter of the year.

Example

The following examples use the input value of 20999. 20999 is the SAS date value that corresponds to June 29, 2017.

```
data _null_;
   declare char(20) a b;
   method run();
      a=put (20999,yyqz6.);
      b=put (20999,yyqz4.);
      put a= b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=201702              b=1702
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Formats:

- "YYQw. Format" on page 216
- "YYQxw. Format" on page 218
- "YYQRw. Format" on page 220
- "YYQRxw. Format" on page 222

Zw.d Format

Writes standard numeric data with leading 0s.

Categories: CAS
Numeric

Alignment: Right

Interaction: When the DECIMALCONV= system option is set to STDIEEE, the output that is written using this format might differ slightly from previous releases. For more information, see "DECIMALCONV= System Option" in *SAS System Options: Reference*.

Syntax

Z*w*.[*d*]

Required Argument

w

specifies the width of the output field.

Default 1

Range 1–32

Tip Allow enough space to write the value, the decimal point, and a minus sign, if necessary.

Optional Argument

d

specifies the number of digits to the right of the decimal point in the numeric value.

Default 0

Range 0–31

Tip If *d* is 0 or you omit *d*, Z*w.d* writes the value without a decimal point.

Details

The Z*w.d* format writes standard numeric values one digit per byte and fills in 0s to the left of the data value.

The Z*w.d* format rounds to the nearest number that will fit in the output field. If *w.d* is too large to fit, SAS might shift the decimal to the BESTw. format. The Z*w.d* format writes negative numbers with leading minus signs. In addition, it right aligns before writing and pads the output with leading zeros.

Example

The following program illustrates the Z*w.d* format:

```
data _null_;
   declare char(20) a ;
   method run();
      a=put(1350,z8.);
      put a= ;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=00001350
```

7

DS2 Functions

Overview of DS2 Functions ... **237**

General Function Syntax ... **238**

Using Functions ... **239**
- Restrictions on Function Arguments ... 239
- Using the OF Operator with Arrays ... 239
- Data Type Conversion in Functions ... 239
- Missing and Null Values in DS2 Numeric Functions ... 239
- Missing and Null Values in DS2 Character Functions ... 239
- Using a System Expression to Execute a Function ... 239
- Notes on Descriptive Statistic Functions ... 239
- Notes about Financial Functions ... 239

DS2 Function Examples ... **244**

Function Categories ... **244**

Dictionary ... **289**
- ABS Function ... 289
- AIRY Function ... 291
- ANYALNUM Function ... 292
- ANYALPHA Function ... 295
- ANYCNTRL Function ... 298
- ANYDIGIT Function ... 300
- ANYFIRST Function ... 302
- ANYGRAPH Function ... 304
- ANYLOWER Function ... 307
- ANYNAME Function ... 309
- ANYPRINT Function ... 311
- ANYPUNCT Function ... 314
- ANYSPACE Function ... 317
- ANYUPPER Function ... 319
- ANYXDIGIT Function ... 321
- ARCOS Function ... 324
- ARCOSH Function ... 325
- ARSIN Function ... 326
- ARSINH Function ... 327
- ARTANH Function ... 329
- ATAN Function ... 330

ATAN2 Function ... 332
BAND Function ... 333
BETA Function ... 334
BETAINV Function ... 336
BLACKCLPRC Function ... 337
BLACKPTPRC Function ... 340
BLKSHCLPRC Function ... 342
BLKSHPTPRC Function ... 345
BLSHIFT Function ... 347
BNOT Function ... 349
BOR Function ... 350
BRSHIFT Function ... 351
BXOR Function ... 352
BYTE Function ... 353
CAT Function ... 354
CATQ Function ... 356
CATS Function ... 361
CATT Function ... 364
CATX Function ... 366
CDF Function ... 370
CDF BERNOULLI Distribution Function ... 372
CDF BETA Distribution Function ... 374
CDF BINOMIAL Distribution Function ... 376
CDF CAUCHY Distribution Function ... 378
CDF Chi-Square Distribution Function ... 380
CDF Conway-Maxwell-Poisson Distribution Function ... 382
CDF Exponential Distribution Function ... 383
CDF F Distribution Function ... 385
CDF GAMMA Distribution Function ... 387
CDF Generalized Poisson Distribution Function ... 389
CDF GEOMETRIC Distribution Function ... 391
CDF HYPERGEOMETRIC Distribution Function ... 392
CDF LAPLACE Distribution Function ... 395
CDF LOGISTIC Distribution Function ... 396
CDF LOGNORMAL Distribution Function ... 398
CDF NEGBINOMIAL Distribution Function ... 400
CDF NORMAL Distribution Function ... 402
CDF NORMALMIX Distribution Function ... 404
CDF PARETO Distribution Function ... 406
CDF POISSON Distribution Function ... 407
CDF T Distribution Function ... 409
CDF TWEEDIE Distribution Function ... 411
CDF UNIFORM Distribution Function ... 413
CDF WALD (Inverse Gaussian) Distribution Function ... 415
CDF WEIBULL Distribution Function ... 417
CEIL Function ... 419
CEILZ Function ... 421
CHOOSEC Function ... 422
CHOOSEN Function ... 424
CMISS Function ... 425
CMP Function ... 427
CMPT Function ... 428
CNONCT Function ... 430
COALESCE Function ... 432

COALESCEC Function ... 434
COMB Function ... 435
COMPARE Function ... 437
COMPBL Function ... 441
COMPCOST Function ... 443
COMPFUZZ Function ... 445
COMPGED Function ... 448
COMPLEV Function ... 455
COMPOUND Function ... 458
COMPRESS Function ... 460
CONSTANT Function ... 465
CONVX Function ... 471
CONVXP Function ... 472
COS Function ... 475
COSH Function ... 476
COT Function ... 477
COUNT Function ... 478
COUNTC Function ... 481
COUNTW Function ... 484
CSC Function ... 489
CSS Function ... 490
CUMIPMT Function ... 491
CUMPRINC Function ... 493
CV Function ... 495
DAIRY Function ... 496
DATDIF Function ... 497
DATE Function ... 500
DATEJUL Function ... 501
DATEPART Function ... 503
DATETIME Function ... 504
DAY Function ... 505
DEQUOTE Function ... 507
DEVIANCE Function ... 511
DHMS Function ... 515
DIF Function ... 518
DIGAMMA Function ... 520
DIM Function ... 521
DIVIDE Function ... 523
DUR Function ... 525
DURP Function ... 527
EFFRATE Function ... 529
ERF Function ... 531
ERFC Function ... 532
EXP Function ... 533
FACT Function ... 535
FINANCE Function ... 537
FINANCE ACCRINT Function ... 541
FINANCE ACCRINTM Function ... 543
FINANCE AMORDEGRC Function ... 545
FINANCE AMORLINC Function ... 547
FINANCE COUPDAYBS Function ... 549
FINANCE COUPDAYS Function ... 551
FINANCE COUPDAYSNC Function ... 552
FINANCE COUPNCD Function ... 554

FINANCE COUPNUM Function ... 556
FINANCE COUPPCD Function ... 557
FINANCE CUMIPMT Function ... 559
FINANCE CUMPRINC Function ... 561
FINANCE DB Function ... 562
FINANCE DDB Function ... 564
FINANCE DISC Function ... 565
FINANCE DOLLARDE Function ... 567
FINANCE DOLLARFR Function ... 568
FINANCE DURATION Function ... 569
FINANCE EFFECT Function ... 571
FINANCE FV Function ... 572
FINANCE FVSCHEDULE Function ... 573
FINANCE INTRATE Function ... 574
FINANCE IPMT Function ... 576
FINANCE IRR Function ... 578
FINANCE ISPMT Function ... 579
FINANCE MDURATION Function ... 580
FINANCE MIRR Function ... 582
FINANCE NOMINAL Function ... 583
FINANCE NPER Function ... 584
FINANCE NPV Function ... 586
FINANCE ODDFPRICE Function ... 587
FINANCE ODDFYIELD Function ... 589
FINANCE ODDLPRICE Function ... 591
FINANCE ODDLYIELD Function ... 594
FINANCE PMT Function ... 596
FINANCE PPMT Function ... 597
FINANCE PRICE Function ... 599
FINANCE PRICEDISC Function ... 601
FINANCE PRICEMAT Function ... 603
FINANCE PV Function ... 605
FINANCE RATE Function ... 606
FINANCE RECEIVED Function ... 608
FINANCE SLN Function ... 609
FINANCE SYD Function ... 611
FINANCE TBILLEQ Function ... 612
FINANCE TBILLPRICE Function ... 613
FINANCE TBILLYIELD Function ... 614
FINANCE VDB Function ... 615
FINANCE XIRR Function ... 617
FINANCE XNPV Function ... 618
FINANCE YIELD Function ... 620
FINANCE YIELDDISC Function ... 622
FINANCE YIELDMAT Function ... 623
FIND Function ... 625
FINDC Function ... 628
FINDW Function ... 637
FLOOR Function ... 644
FLOORZ Function ... 646
FMTINFO Function ... 648
FNONCT Function ... 650
FUZZ Function ... 652
GAMINV Function ... 654

GAMMA Function ... 655
GARKHCLPRC Function ... 656
GARKHPTPRC Function ... 659
GCD Function ... 662
GEODIST Function ... 663
GEOMEAN Function ... 666
GEOMEANZ Function ... 668
HARMEAN Function ... 670
HARMEANZ Function ... 672
HBOUND Function ... 674
HMS Function ... 676
HOLIDAY Function ... 677
HOUR Function ... 682
IBESSEL Function ... 683
INDEX Function ... 685
INDEXC Function ... 687
INDEXW Function ... 689
INPUTC Function ... 691
INPUTN Function ... 693
INT Function ... 694
INTCINDEX Function ... 696
INTCK Function ... 701
INTCYCLE Function ... 710
INTDT Function ... 715
INTFIT Function ... 717
INTGET Function ... 719
INTINDEX Function ... 722
INTNEST Function ... 729
INTNX Function ... 732
INTRR Function ... 742
INTSEAS Function ... 744
INTSHIFT Function ... 748
INTTEST Function ... 752
INTTS Function ... 755
INTZ Function ... 757
IPMT Function ... 759
IQR Function ... 761
IRR Function ... 762
JBESSEL Function ... 764
JULDATE Function ... 765
JULDATE7 Function ... 767
KURTOSIS Function ... 768
LAG Function ... 770
LARGEST Function ... 774
LBOUND Function ... 776
LCM Function ... 778
LCOMB Function ... 780
LEFT Function ... 781
LENGTH Function ... 782
LENGTHC Function ... 784
LENGTHM Function ... 787
LFACT Function ... 789
LGAMMA Function ... 790
LOG Function ... 791

LOG10 Function ... 792
LOG1PX Function ... 793
LOG2 Function ... 795
LOGBETA Function ... 796
LOGCDF Function ... 798
LOGISTIC Function ... 800
LOGPDF Function ... 801
LOGSDF Function ... 804
LOWCASE Function ... 806
MAD Function ... 808
MARGRCLPRC Function ... 809
MARGRPTPRC Function ... 812
MAX Function ... 815
MD5 Function ... 816
MDY Function ... 818
MEAN Function ... 820
MEDIAN Function ... 821
MIN Function ... 823
MINUTE Function ... 824
MISSING Function ... 826
MOD Function ... 829
MODZ Function ... 831
MONTH Function ... 834
MORT Function ... 835
N Function ... 837
NDIMS Function ... 838
NETPV Function ... 840
NMISS Function ... 842
NOMRATE Function ... 844
NOTALNUM Function ... 845
NOTALPHA Function ... 848
NOTCNTRL Function ... 850
NOTDIGIT Function ... 852
NOTFIRST Function ... 855
NOTGRAPH Function ... 857
NOTLOWER Function ... 860
NOTNAME Function ... 862
NOTPRINT Function ... 864
NOTPUNCT Function ... 866
NOTSPACE Function ... 869
NOTUPPER Function ... 872
NOTXDIGIT Function ... 874
NPV Function ... 876
NULL Function ... 878
NWKDOM Function ... 880
ORDINAL Function ... 883
PCTL Function ... 884
PDF Function ... 886
PDF BERNOULLI Distribution Function ... 888
PDF BETA Distribution Function ... 890
PDF BINOMIAL Distribution Function ... 892
PDF CAUCHY Distribution Function ... 894
PDF Chi-Square Distribution Function ... 896
PDF Conway-Maxwell-Poisson Distribution Function ... 898

PDF EXPONENTIAL Distribution Function ... 901
PDF F Distribution Function ... 902
PDF GAMMA Distribution Function ... 904
PDF Generalized Poisson Distribution Function ... 906
PDF GEOMETRIC Distribution Function ... 908
PDF Hypergeometric Distribution Function ... 909
PDF LAPLACE Distribution Function ... 911
PDF LOGISTIC Distribution Function ... 913
PDF LOGNORMAL Distribution Function ... 915
PDF NEGBINOMIAL Distribution Function ... 916
PDF NORMAL Distribution Function ... 918
PDF NORMALMIX Distribution Function ... 920
PDF PARETO Distribution Function ... 922
PDF POISSON Distribution Function ... 924
PDF T Distribution Function ... 926
PDF TWEEDIE Distribution Function ... 927
PDF UNIFORM Distribution Function ... 930
PDF Wald (Inverse Gaussian) Distribution Function ... 932
PDF WEIBULL Distribution Function ... 933
PERM Function ... 935
PMT Function ... 937
POISSON Function ... 939
POWER Function ... 940
PPMT Function ... 941
PROBBETA Function ... 943
PROBBNML Function ... 945
PROBBNRM Function ... 946
PROBCHI Function ... 947
PROBDF Function ... 949
PROBF Function ... 955
PROBGAM Function ... 957
PROBHYPR Function ... 958
PROBIT Function ... 959
PROBMC Function ... 961
PROBMED Function ... 972
PROBNEGB Function ... 973
PROBNORM Function ... 975
PROBT Function ... 976
PRXCHANGE Function ... 977
PRXMATCH Function ... 981
PRXPARSE Function ... 985
PRXPOSN Function ... 988
PUT Function ... 992
PVP Function ... 995
QTR Function ... 997
QUANTILE Function ... 998
QUOTE Function ... 1002
RANBIN Function ... 1004
RANCAU Function ... 1004
RAND Function ... 1004
RANEXP Function ... 1026
RANGAM Function ... 1026
RANGE Function ... 1026
RANK Function ... 1027

RANNOR Function ... 1028
RANPOI Function ... 1029
RANTBL Function ... 1029
RANTRI Function ... 1029
RANUNI Function ... 1029
REPEAT Function ... 1029
REVERSE Function ... 1031
RIGHT Function ... 1032
RMS Function ... 1033
ROUND Function ... 1035
ROUNDE Function ... 1044
ROUNDZ Function ... 1047
SAVING Function ... 1050
SAVINGS Function ... 1052
SCAN Function ... 1055
SDF Function ... 1065
SEC Function ... 1069
SECOND Function ... 1071
SHA256 Function ... 1073
SHA256HEX Function ... 1074
SHA256HMACHEX Function ... 1077
SIGN Function ... 1079
SIN Function ... 1080
SINH Function ... 1081
SKEWNESS Function ... 1082
SLEEP Function ... 1083
SMALLEST Function ... 1085
SQLEXEC Function ... 1087
SQRT Function ... 1089
SQUANTILE Function ... 1090
STD Function ... 1094
STDERR Function ... 1095
STREAMINIT Function ... 1096
STRIP Function ... 1102
SUBSTR (left of =) Function ... 1105
SUBSTR (right of =) Function ... 1108
SUBSTRN Function ... 1111
SUM Function ... 1116
SUMABS Function ... 1118
SYSGET Function ... 1119
TAN Function ... 1121
TANH Function ... 1122
TIME Function ... 1124
TIMEPART Function ... 1125
TIMEVALUE Function ... 1126
TINV Function ... 1129
TNONCT Function ... 1131
TO_DATE Function ... 1133
TO_DOUBLE Function ... 1134
TO_TIME Function ... 1137
TO_TIMESTAMP Function ... 1139
TODAY Function ... 1140
TRANSLATE Function ... 1141
TRANSTRN Function ... 1143

TRANWRD Function 1146
TRIGAMMA Function 1150
TRIM Function 1151
TRUNC Function 1153
UNIFORM Function 1155
UPCASE Function 1155
URLDECODE Function 1157
URLENCODE Function 1158
USS Function 1160
UUIDGEN Function 1162
VAR Function 1163
VERIFY Function 1164
VFORMAT Function 1166
VINARRAY Function 1167
VINFORMAT Function 1169
VLABEL Function 1171
VLENGTH Function 1172
VNAME Function 1174
VTYPE Function 1175
WEEK Function 1177
WEEKDAY Function 1181
WHICHC Function 1182
WHICHN Function 1183
YEAR Function 1185
YIELDP Function 1186
YRDIF Function 1188
YYQ Function 1191

Overview of DS2 Functions

A **DS2 function** performs a computation or system manipulation on arguments and returns a value. A function expression invokes a function from anywhere in a DS2 program, method, or thread. Most functions use arguments supplied by the user, but a few obtain their arguments from the operating environment.

If the data types of the arguments in the function expression are not what is expected by the DS2 function, DS2 performs a type conversion on the arguments so that they have the appropriate data type. If the type conversion is successful, the function executes. Otherwise, an error is issued. For information, see "DS2 Type Conversions" in *SAS DS2 Programmer's Guide*.

Note: DS2 does not support CALL routines. DS2 functions and methods can alter variable argument values if the return type of the function or method is VOID.

Note: The date and time functions work only with SAS date, time, and datetime values. They do not work with values having the DATE, TIME, and TIMESTAMP

data types. For information about working with dates, see "DS2 Expressions" in *SAS DS2 Programmer's Guide*.

General Function Syntax

The syntax of a function has one of the following forms:

function-name (*argument*[, ...*argument*])

function-name (OF *variable-list*)

function-name ([*argument* | OF *variable-list* | OF *array-name*[*]]
[..., [*argument* | OF *variable-list* | OF *array-name*[*]]])

function-name
names the function.

argument
can be a variable name, constant, or any DS2 expression, including another function. The number and type of arguments that DS2 allows are described with individual functions. Multiple arguments are separated by a comma.

Note: If the value of an argument is invalid (for example, missing or outside the prescribed range), an error occurs and the function's return expression is set to a missing or null value.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example Here are examples:

```
x=max(cash,credit);
x=sqrt(1500);
NewCity=left(upcase(City));
x=min(YearTemperature-July,YearTemperature-Dec);
s=repeat('-+.16);
x=min((enroll-drop),(enroll-fail));
if sum(cash,credit)>1000 then put 'Goal reached';
```

variable-list
can be any form of a DS2 variable list, including individual variable names. If more than one variable list appears, separate them with a space or with a comma and another OF.

```
a=sum(of x y z);


z=sum(of y1-y10);

z=msplint(x0,5,of x1-x5,of y1-y5,-2,2);
```

See "The OF Operator with Variable Lists" in *SAS DS2 Programmer's Guide*

Example The following two examples are equivalent.

```
a=sum(of x1-x10 y1-y10 z1-z10);
a=sum(of x1-x10, of y1-y10, of z1-z10);
```

***array-name*[*]**
names a currently defined array. Specifying an array with an asterisk as a subscript causes DS2 to treat each element of the array as a separate argument.

See "Using the OF Operator with Arrays" on page 239

Using Functions

Restrictions on Function Arguments

If the value of an argument is invalid, an error occurs and the return expression is set to a missing or null value. Here are some common restrictions on function arguments:

- Some functions require that their arguments be restricted within a certain range. For example, the argument of the LOG function must be greater than 0.
- Most functions do not permit missing or null values as arguments. Exceptions include some of the descriptive statistics functions and financial functions.

  By default when a function argument contains a missing or null value, an error occurs and a message is printed to the SAS log. You can use the MISSING_NOTE option in the DS2_OPTIONS statement to not produce an error and write a note to the SAS log when a function argument contains a missing or null value. For more information, see "DS2_OPTIONS Statement" on page 1247.
- In general, the allowed range of the arguments is platform-dependent, such as with the EXP function.

Using the OF Operator with Arrays

You can use the OF operator with DS2 variable arrays. This capability enables the passing of variable arrays to most functions whose arguments contain a varying number of parameters.

There are some rules and limitations when using variable arrays. These rules and limitations are listed after the example.

The following example shows how you can use a variable array in a SUM function:

```
data _null_;
```

```
   vararray double a[4];
   method init();
      a:=(1,2,3,4);
   end;

   method run();
      dcl double x y z;
         y=99.0;
        z=100.0;
        x=sum(z, of a[*], y);
        put x=;

      x= sum(of a:, z);
      put x=;
   end;
enddata;
run;
```

Example Code 7.1 *Log Output for the Example of Using Variable Arrays*

```
x=209
x=110
```

The following rules and limitations apply to variable array OF lists:

- can be used in functions where the number of parameters matches the number of elements in the OF list

  Note: A function can contain no OF lists, one OF lists, some OF lists, some OF lists with other expressions, and so on. The total number of arguments after the all of the OF lists are expanded must match the number of arguments that the function takes if the function has a fixed number of arguments.

- can be used in functions that take a varying number of parameters
- cannot be used as array indices
- cannot be used with the DIF and LAGfunctions, nor with any of the variable information functions such as VLENGTH
- cannot be used with functions that are specified in a WHERE clause. Here is an example:

```
where range(of x1-x3);
```

Data Type Conversion in Functions

The number of arguments and the argument data types that DS2 expects for a function is called the **function signature**. When a function executes, the arguments in the function expression are compared to the function signature. If the arguments and the signature match, the function executes. If the number of arguments do not match, an error occurs.

If the number of arguments match but one or more argument data types do not match, DS2 attempts to convert the argument data types to those of the function signature. If the type conversion is successful, the function executes. Otherwise, an error occurs.

The documentation for each of the functions includes a valid data type for all function arguments. The valid data type is the argument data type that DS2 uses in executing the function. Because DS2 does type conversion, some argument data types in the function expression do not need to be the same as the valid data types for the function argument. For example, if the function expression contains an argument with a data type of INTEGER and the valid data type for that argument is DOUBLE, DS2 converts the data type to DOUBLE when the function executes. Only four data types, VARBINARY and the date/time data types, DATE, TIME, TIMESTAMP, are non-coercible, meaning that the function expression must contain the valid data type.

Missing and Null Values in DS2 Numeric Functions

For functions that have an input data type of DOUBLE, if a null is passed to the function, the null value is converted to a missing value.

After the function is processed, if the function returns a DOUBLE and if the function returns a missing value, a missing value is returned in SAS mode and a null value is returned in ANSI mode.

For more information, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.

Missing and Null Values in DS2 Character Functions

If you pass a character function a null value and the function should return a null value, a null value is returned regardless of whether you are in ANSI mode or SAS mode.

Using a System Expression to Execute a Function

The syntax for a function is similar to the syntax for a method, and when DS2 encounters a method or function expression, it must determine what type of expression it is. If a method expression and a function expression have the same name, DS2 executes the method. The only way to execute a function that has the same name as a method is to use a system expression. A system expression has the following syntax:

```
system.function-expression
```

When DS2 encounters a system expression, the call to the method of the same name is bypassed and DS2 calls the function.

Notes on Descriptive Statistic Functions

DS2 provides functions that return descriptive statistics. Except for the MISSING function, the functions correspond to the statistics produced by the Base SAS MEANS procedure. The computing method for each statistic is discussed in the statistical procedures section of the *Base SAS Procedures Guide*. SAS calculates descriptive statistics for the non-null or nonmissing values of the arguments.

Notes about Financial Functions

Types of Financial Functions

SAS provides a group of functions that perform financial calculations. The functions are grouped into the following types:

Table 7.1 *Types of Financial Functions*

Function Type	Function	Description
Cash Flow	CONVX, CONVXP	calculates convexities for cash flows
	DUR, DURP	calculates modifies duration for cash flows.
	PVP, YIELDP	calculates present value and yield-to-maturity for a periodic cash flow
General	FINANCE	calculates depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
Internal rate of return	INTRR, IRR	calculates the internal rate of return
Net present and future value	NETPV, NPV	calculates net present and future values
	SAVINGS	returns the balance of periodic savings by using variable interest rates

Function Type	Function	Description
Parameter calculations	COMPOUND	calculates compound interest parameters
	MORT	calculates amortization parameters
Pricing	BLKSHCLPRC, BLKSHPTPRC	calculated call prices and put prices for European options on stocks, based on the Black-Scholes model
	BLACKCLPRC, BLACKPTPRC	calculates call prices and put prices for European options on futures, based on the Black model
	GARKHCLPRC, GARKHPTPRC	calculates call prices and put prices for European options on stocks, based on the Garman-Kohlhagen model
	MARGRCLPRC, MARGRPTPRC	calculates call options and put prices for European options on stocks, based on the Margrabe model

Using Pricing Functions

A pricing model is used to calculate a theoretical market value (price) for a financial instrument. This value is referred to as a mark-to-market (MtM) value. Typically, a pricing function has the following form:

$$price = function(rf1, rf2, rf3, ...)$$

In the pricing function, *rf1*, *rf2*, and *rf3* are risk factors such as interest rates or foreign exchange rates. The specific values of the risk factors that are used to calculate the MtM value are the base case values. The set of base case values is known as the base case market state.

After determining the MtM value, you can perform the following tasks with the base case values of the risk factors (*rf1*, *rf2*, and *rf3*):

- Set the base case values to specific values to perform scenario analyses.
- Set the base case values to a range of values to perform profit/loss curve analyses and profit/loss surface analyses.
- Automatically set the base case values to different values to calculate sensitivities - that is, to calculate the delta and gamma values of the risk factors.
- Perturb the base case values to create many possible market states so that many possible future prices can be calculated, and simulation analyses can be

performed. For Monte Carlo simulation, the values of the risk factors are generated using mathematical models and the copula methodology.

A list of pricing functions and their descriptions is included in "Types of Financial Functions" on page 242.

DS2 Function Examples

```
x=max(cash,credit);

x=sqrt(1500);

NewCity=left(upcase(City));

x=min(YearTemperature-July,YearTemperature-Dec);

s=repeat('----+',16);

x=min((enroll-drop),(enroll-fail));

if sum(cash,credit)>1000 then
   put 'Goal reached';
```

Function Categories

Functions can be categorized by the types of values that they operate on. Each DS2 function belongs to one of the following categories:

Table 7.2 *Function Category Descriptions*

Category	Description
Arithmetic	returns the result of a division that handles special missing values for ODS output
Array	operates on a named aggregate collection of homogenous data
Bitwise Logical Operations	operates on one or more bit patterns or binary numbers at the level of their individual bits
CAS	functions that can be used on the CAS server
Character	operates on character data and SQL expressions
Date and Time	operates on date and time values

Category	Description
Descriptive Statistics	operates on values that measure central tendency, variation among values, and the shape of distribution values
Distance	returns the geodetic distance
Financial	calculates financial values such as interest, periodic payments, depreciation, and prices for European options on stocks
Hyperbolic	performs hyperbolic calculations such as sine, cosine, and tangent
Mathematical	operates on values to perform general mathematical calculations
Numeric	operates on numeric values
Probability	returns probability calculations
Quantile	returns a quantile from specific distributions
Random Number	returns random variates from specific distributions
Special	operates on null values and SAS missing values, suspends execution of a program, specifies numeric informats at run time, and executes a FedSQL statement.
Trigonometric	operates on values to perform trigonometric calculations
Truncation	operates on values to limit the number of digits
Variable Information	operates on variables and returns names, types, lengths, informats, labels, and other variable information
Web Tools	encodes and decodes a string of data

The following table provides brief descriptions of DS2 functions. For more detailed information, see the individual functions.

Category	Language Elements	Description
Arithmetic	DIVIDE Function (p. 523)	Returns the result of a division that handles special missing values for ODS output.
Array	DIM Function (p. 521)	Returns the number of elements in an array.
	HBOUND Function (p. 674)	Returns the upper bound of an array.
	LBOUND Function (p. 776)	Returns the lower bound of an array.

Category	Language Elements	Description
	NDIMS Function (p. 838)	Returns the number of dimensions in an array.
Bitwise Logical Operations	BAND Function (p. 333)	Returns the bitwise logical AND of two arguments.
	BLSHIFT Function (p. 347)	Returns the bitwise logical left shift of two arguments.
	BNOT Function (p. 349)	Returns the bitwise logical NOT of an argument.
	BOR Function (p. 350)	Returns the bitwise logical OR of two arguments.
	BRSHIFT Function (p. 351)	Returns the bitwise logical right shift of two arguments.
	BXOR Function (p. 352)	Returns the bitwise logical EXCLUSIVE OR of two arguments.
CAS	ABS Function (p. 289)	Returns the absolute value of a numeric value expression.
	AIRY Function (p. 291)	Returns the value of the Airy function.
	ANYALNUM Function (p. 292)	Searches a character string for an alphanumeric character, and returns the first character position at which the character is found.
	ANYALPHA Function (p. 295)	Searches a character string for an alphabetic character, and returns the first character position at which the character is found.
	ANYCNTRL Function (p. 298)	Searches a character string for a control character, and returns the first character position at which that character is found.
	ANYDIGIT Function (p. 300)	Searches a character string for a digit, and returns the first character position at which the digit is found.
	ANYFIRST Function (p. 302)	Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	ANYGRAPH Function (p. 304)	Searches a character string for a graphical character, and returns the first character position at which that character is found.
	ANYLOWER Function (p. 307)	Searches a character string for a lowercase letter, and returns the first character position at which the letter is found.
	ANYNAME Function (p. 309)	Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	ANYPRINT Function (p. 311)	Searches a character string for a printable character, and returns the first character position at which that character is found.

Category	Language Elements	Description
	ANYPUNCT Function (p. 314)	Searches a character string for a punctuation character, and returns the first character position at which that character is found.
	ANYSPACE Function (p. 317)	Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.
	ANYUPPER Function (p. 319)	Searches a character string for an uppercase letter, and returns the first character position at which the letter is found.
	ANYXDIGIT Function (p. 321)	Searches a character string for a hexadecimal character that represents a digit, and returns the first character position at which that character is found.
	ARCOS Function (p. 324)	Returns the arccosine in radians.
	ARCOSH Function (p. 325)	Returns the inverse hyperbolic cosine.
	ARSIN Function (p. 326)	Returns the arcsine in radians.
	ARSINH Function (p. 327)	Returns the inverse hyperbolic sine.
	ARTANH Function (p. 329)	Returns the inverse hyperbolic tangent.
	ATAN Function (p. 330)	Returns the arctangent in radians.
	ATAN2 Function (p. 332)	Returns the arctangent of the x and y coordinates of a right triangle, in radians.
	BAND Function (p. 333)	Returns the bitwise logical AND of two arguments.
	BETA Function (p. 334)	Returns the value of the beta function.
	BETAINV Function (p. 336)	Returns a quantile from the beta distribution.
	BLACKCLPRC Function (p. 337)	Calculates call prices for European options on futures, based on the Black model.
	BLACKPTPRC Function (p. 340)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 342)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	BLKSHPTPRC Function (p. 345)	Calculates put prices for European options on stocks, based on the Black-Scholes model.
	BLSHIFT Function (p. 347)	Returns the bitwise logical left shift of two arguments.
	BNOT Function (p. 349)	Returns the bitwise logical NOT of an argument.

Category	Language Elements	Description
	BOR Function (p. 350)	Returns the bitwise logical OR of two arguments.
	BRSHIFT Function (p. 351)	Returns the bitwise logical right shift of two arguments.
	BXOR Function (p. 352)	Returns the bitwise logical EXCLUSIVE OR of two arguments.
	BYTE Function (p. 353)	Returns one character in the ASCII or the EBCDIC collating sequence.
	CAT Function (p. 354)	Does not remove leading or trailing blanks, and returns a concatenated character string.
	CATS Function (p. 361)	Removes leading and trailing blanks, and returns a concatenated character string.
	CATT Function (p. 364)	Removes trailing blanks, and returns a concatenated character string.
	CATX Function (p. 366)	Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.
	CDF Function (p. 370)	Computes the left cumulative distribution function from various continuous and discrete probability distributions.
	CDF BERNOULLI Distribution Function (p. 372)	Returns a value from the Bernoulli cumulative probability distribution.
	CDF BETA Distribution Function (p. 374)	Returns a value from the beta cumulative probability distribution.
	CDF BINOMIAL Distribution Function (p. 376)	Returns a value from the binomial cumulative probability distribution.
	CDF CAUCHY Distribution Function (p. 378)	Returns a value from the Cauchy cumulative probability distribution.
	CDF Chi-Square Distribution Function (p. 380)	Returns a value from the chi-square cumulative probability distribution.
	CDF Conway-Maxwell-Poisson Distribution Function (p. 382)	Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.
	CDF Exponential Distribution Function (p. 383)	Returns a value from the exponential cumulative probability distribution.
	CDF F Distribution Function (p. 385)	Returns a value from the F cumulative probability distribution.
	CDF GAMMA Distribution Function (p. 387)	Returns a value from the gamma cumulative probability distribution.

Category	Language Elements	Description
	CDF Generalized Poisson Distribution Function (p. 389)	Returns a value from the generalized Poisson cumulative probability distribution.
	CDF GEOMETRIC Distribution Function (p. 391)	Returns a value from the geometric cumulative probability distribution.
	CDF HYPERGEOMETRIC Distribution Function (p. 392)	Returns a value from the hypergeometric cumulative probability distribution.
	CDF LAPLACE Distribution Function (p. 395)	Returns a value from the Laplace cumulative probability distribution.
	CDF LOGISTIC Distribution Function (p. 396)	Returns a value from the logistic cumulative probability distribution.
	CDF LOGNORMAL Distribution Function (p. 398)	Returns a value from the lognormal cumulative probability distribution.
	CDF NEGBINOMIAL Distribution Function (p. 400)	Returns a value from the negative binomial cumulative probability distribution.
	CDF NORMAL Distribution Function (p. 402)	Returns a value from the normal cumulative probability distribution.
	CDF NORMALMIX Distribution Function (p. 404)	Returns a value from the normal mixture cumulative probability distribution.
	CDF PARETO Distribution Function (p. 406)	Returns a value from the Pareto cumulative probability distribution.
	CDF POISSON Distribution Function (p. 407)	Returns a value from the Poisson cumulative probability distribution.
	CDF T Distribution Function (p. 409)	Returns a value from the T cumulative probability distribution.
	CDF TWEEDIE Distribution Function (p. 411)	Returns a value from the Tweedie cumulative probability distribution.
	CDF UNIFORM Distribution Function (p. 413)	Returns a value from the uniform cumulative probability distribution.
	CDF WALD (Inverse Gaussian) Distribution Function (p. 415)	Returns a value from the Wald (also known as the inverse Gaussian) cumulative probability distribution.
	CDF WEIBULL Distribution Function (p. 417)	Returns a value from the Weibull cumulative probability distribution.

Category	Language Elements	Description
	CEIL Function (p. 419)	Returns the smallest integer greater than or equal to a numeric value expression.
	CEILZ Function (p. 421)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	CHOOSEC Function (p. 422)	Returns a character value that represents the results of choosing from a list of arguments.
	CHOOSEN Function (p. 424)	Returns a numeric value that represents the results of choosing from a list of arguments.
	CMISS Function (p. 425)	Counts the number of missing arguments.
	CMP Function (p. 427)	Compares two character strings including trailing blanks.
	CMPT Function (p. 428)	Compares two character strings excluding trailing blanks.
	CNONCT Function (p. 430)	Returns the noncentrality parameter from a chi-square distribution.
	COALESCE Function (p. 432)	Returns the first non-null or nonmissing value from a list of numeric arguments.
	COALESCEC Function (p. 434)	Returns the first non-null or nonmissing value from a list of character arguments.
	COMB Function (p. 435)	Computes the number of combinations of n elements taken r at a time.
	COMPARE Function (p. 437)	Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.
	COMPBL Function (p. 441)	Removes multiple blanks from a character string.
	COMPCOST Function (p. 443)	Sets the costs of operations for later use by the COMPGED function.
	COMPFUZZ Function (p. 445)	Performs a fuzzy comparison of two numeric values.
	COMPGED Function (p. 448)	Returns the generalized edit distance between two strings.
	COMPLEV Function (p. 455)	Returns the Levenshtein edit distance between two strings.
	COMPOUND Function (p. 458)	Returns compound interest parameters.
	COMPRESS Function (p. 460)	Returns a character string with specified characters removed from the original string.
	CONSTANT Function (p. 465)	Computes machine and mathematical constants.

Category	Language Elements	Description
	CONVX Function (p. 471)	Returns the convexity for an enumerated cash flow.
	CONVXP Function (p. 472)	Returns the convexity for a periodic cash flow stream, such as a bond.
	COS Function (p. 475)	Returns the cosine in radians.
	COSH Function (p. 476)	Returns the hyperbolic cosine in radians.
	COT Function (p. 477)	Returns the cotangent.
	COUNT Function (p. 478)	Counts the number of times that a specified substring appears within a character string.
	COUNTC Function (p. 481)	Counts the number of characters in a string that appear or do not appear in a list of characters.
	COUNTW Function (p. 484)	Counts the number of words in a character string.
	CSC Function (p. 489)	Returns the cosecant.
	CSS Function (p. 490)	Returns the corrected sum of squares.
	CUMIPMT Function (p. 491)	Returns the cumulative interest paid on a loan between the start and end period.
	CUMPRINC Function (p. 493)	Returns the cumulative principal paid on a loan between the start and end period.
	CV Function (p. 495)	Returns the coefficient of variation.
	DAIRY Function (p. 496)	Returns the derivative of the AIRY function.
	DATDIF Function (p. 497)	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.
	DATE Function (p. 500)	Returns the current date as a SAS date value.
	DATEJUL Function (p. 501)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 503)	Extracts the date from a SAS datetime value.
	DATETIME Function (p. 504)	Returns the current date and time of day as a SAS datetime value.
	DAY Function (p. 505)	Returns the day of the month from a SAS date value.
	DEQUOTE Function (p. 507)	Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.
	DEVIANCE Function (p. 511)	Returns the deviance based on a probability distribution.

Category	Language Elements	Description
	DHMS Function (p. 515)	Returns a SAS datetime value from date, hour, minute, and second values.
	DIF Function (p. 518)	Returns differences between an argument and its nth lag.
	DIGAMMA Function (p. 520)	Returns the value of the digamma function.
	DIM Function (p. 521)	Returns the number of elements in an array.
	DIVIDE Function (p. 523)	Returns the result of a division that handles special missing values for ODS output.
	DUR Function (p. 525)	Returns the modified duration for an enumerated cash flow.
	DURP Function (p. 527)	Returns the modified duration for a periodic cash flow stream, such as a bond.
	EFFRATE Function (p. 529)	Returns the effective annual interest rate.
	ERF Function (p. 531)	Returns the value of the (normal) error function.
	ERFC Function (p. 532)	Returns the value of the complementary (normal) error function.
	EXP Function (p. 533)	Returns the value of the e constant raised to a specified power.
	FACT Function (p. 535)	Computes a factorial.
	FINANCE Function (p. 537)	Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
	FINANCE ACCRINT Function (p. 541)	Computes the accrued interest for a security that pays periodic interest.
	FINANCE ACCRINTM Function (p. 543)	Computes the accrued interest for a security that pays interest at maturity.
	FINANCE AMORDEGRC Function (p. 545)	Computes the depreciation for each accounting period by using a depreciation coefficient.
	FINANCE AMORLINC Function (p. 547)	Computes the depreciation for each accounting period.
	FINANCE COUPDAYBS Function (p. 549)	Computes the number of days from the beginning of the coupon period to the settlement date.
	FINANCE COUPDAYS Function (p. 551)	Computes the number of days in the coupon period that contains the settlement date.
	FINANCE COUPDAYSNC Function (p. 552)	Computes the number of days from the settlement date to the next coupon date.

Category	Language Elements	Description
	FINANCE COUPNCD Function (p. 554)	Computes the next coupon date after the settlement date.
	FINANCE COUPNUM Function (p. 556)	Computes the number of coupons that are payable between the settlement date and the maturity date.
	FINANCE COUPPCD Function (p. 557)	Computes the previous coupon date before the settlement date.
	FINANCE CUMIPMT Function (p. 559)	Computes the cumulative interest paid between two periods.
	FINANCE CUMPRINC Function (p. 561)	Computes the cumulative principal that is paid on a loan between two periods.
	FINANCE DB Function (p. 562)	Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
	FINANCE DDB Function (p. 564)	Computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.
	FINANCE DISC Function (p. 565)	Computes the discount rate for a security.
	FINANCE DOLLARDE Function (p. 567)	Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
	FINANCE DOLLARFR Function (p. 568)	Converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.
	FINANCE DURATION Function (p. 569)	Computes the annual duration of a security with periodic interest payments.
	FINANCE EFFECT Function (p. 571)	Computes the effective annual interest rate.
	FINANCE FV Function (p. 572)	Computes the future value of an investment.
	FINANCE FVSCHEDULE Function (p. 573)	Computes the future value of the initial principal after applying a series of compound interest rates.
	FINANCE INTRATE Function (p. 574)	Computes the interest rate for a fully invested security.
	FINANCE IPMT Function (p. 576)	Computes the interest payment for an investment for a specified period.
	FINANCE IRR Function (p. 578)	Computes the internal rate of return for a series of cash flows.
	FINANCE ISPMT Function (p. 579)	Calculates the interest paid during a specific period of an investment.

Category	Language Elements	Description
	FINANCE MDURATION Function (p. 580)	Computes the Macaulay modified duration for a security with an assumed face value of $100.
	FINANCE MIRR Function (p. 582)	Computes the internal rate of return where positive and negative cash flows are financed at different rates.
	FINANCE NOMINAL Function (p. 583)	Computes the annual nominal interest rates.
	FINANCE NPER Function (p. 584)	Computes the number of periods for an investment.
	FINANCE NPV Function (p. 586)	Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
	FINANCE ODDFPRICE Function (p. 587)	Computes the price of a security per $100 face value with an odd first period.
	FINANCE ODDFYIELD Function (p. 589)	Computes the yield of a security with an odd first period.
	FINANCE ODDLPRICE Function (p. 591)	Computes the price of a security per $100 face value with an odd last period.
	FINANCE ODDLYIELD Function (p. 594)	Computes the yield of a security with an odd last period.
	FINANCE PMT Function (p. 596)	Computes the periodic payment of an annuity.
	FINANCE PPMT Function (p. 597)	Computes the payment on the principal for an investment for a specified period.
	FINANCE PRICE Function (p. 599)	Computes the price of a security per $100 face value that pays periodic interest.
	FINANCE PRICEDISC Function (p. 601)	Computes the price of a discounted security per $100 face value.
	FINANCE PRICEMAT Function (p. 603)	Computes the price of a security per $100 face value that pays interest at maturity.
	FINANCE PV Function (p. 605)	Computes the present value of an investment.
	FINANCE RATE Function (p. 606)	Computes the interest rate per period of an annuity.
	FINANCE RECEIVED Function (p. 608)	Computes the amount that is received at maturity for a fully invested security.
	FINANCE SLN Function (p. 609)	Computes the straight-line depreciation of an asset for one period.

Category	Language Elements	Description
	FINANCE SYD Function (p. 611)	Computes the sum-of-years digits depreciation of an asset for a specified period.
	FINANCE TBILLEQ Function (p. 612)	Computes the bond-equivalent yield for a treasury bill.
	FINANCE TBILLPRICE Function (p. 613)	Computes the price of a treasury bill per $100 face value.
	FINANCE TBILLYIELD Function (p. 614)	Computes the yield for a treasury bill.
	FINANCE VDB Function (p. 615)	Computes the depreciation of an asset for a specified or partial period by using a declining balance method.
	FINANCE XIRR Function (p. 617)	Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
	FINANCE XNPV Function (p. 618)	Computes the net present value for a schedule of cash flows that is not necessarily periodic.
	FINANCE YIELD Function (p. 620)	Computes the yield on a security that pays periodic interest.
	FINANCE YIELDDISC Function (p. 622)	Computes the annual yield for a discounted security (for example, a treasury bill).
	FINANCE YIELDMAT Function (p. 623)	Computes the annual yield of a security that pays interest at maturity.
	FIND Function (p. 625)	Searches for a specific substring of characters within a character string.
	FINDC Function (p. 628)	Searches a string for any character in a list of characters.
	FINDW Function (p. 637)	Returns the character position of a word in a string, or returns the number of the word in a string.
	FLOOR Function (p. 644)	Returns the largest integer less than or equal to a numeric value expression.
	FLOORZ Function (p. 646)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
	FMTINFO Function (p. 648)	Returns information about a SAS format or informat.
	FNONCT Function (p. 650)	Returns the value of the noncentrality parameter of an F distribution.
	FUZZ Function (p. 652)	Returns the nearest whole number if the argument is within 1E-12 of that number.
	GAMINV Function (p. 654)	Returns a quantile from the gamma distribution.

Category	Language Elements	Description
	GAMMA Function (p. 655)	Returns the value of the gamma function.
	GARKHCLPRC Function (p. 656)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
	GARKHPTPRC Function (p. 659)	Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.
	GCD Function (p. 662)	Returns the greatest common divisor for a set of integers.
	GEODIST Function (p. 663)	Returns the geodetic distance between two latitude and longitude coordinates.
	GEOMEAN Function (p. 666)	Returns the geometric mean.
	GEOMEANZ Function (p. 668)	Returns the geometric mean, using zero fuzzing.
	HARMEAN Function (p. 670)	Returns the harmonic mean.
	HARMEANZ Function (p. 672)	Returns the harmonic mean, using zero fuzzing.
	HBOUND Function (p. 674)	Returns the upper bound of an array.
	HMS Function (p. 676)	Returns a SAS time value from hour, minute, and second values.
	HOLIDAY Function (p. 677)	Returns a SAS date value of a specified holiday for a specified year.
	HOUR Function (p. 682)	Returns the hour from a SAS time or datetime value.
	IBESSEL Function (p. 683)	Returns the value of the modified Bessel function.
	INDEX Function (p. 685)	Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.
	INDEXC Function (p. 687)	Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.
	INDEXW Function (p. 689)	Searches a character expression for a string that is specified as a word, and returns the position of the first character in the word.
	INPUTC Function (p. 691)	Enables you to specify a character informat at run time.
	INPUTN Function (p. 693)	Enables you to specify a numeric informat at run time.
	INT Function (p. 694)	Returns the whole number, fuzzed to avoid unexpected floating-point results.

Category	Language Elements	Description
	INTCINDEX Function (p. 696)	Returns the cycle index when a date, time, or timestamp interval and value are specified.
	INTCK Function (p. 701)	Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.
	INTCYCLE Function (p. 710)	Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.
	INTDT Function (p. 715)	Specifies the number of days to add to a DATE value.
	INTFIT Function (p. 717)	Returns a time interval that is aligned between two dates.
	INTGET Function (p. 719)	Returns a time interval based on three date or datetime values.
	INTINDEX Function (p. 722)	Returns the seasonal index when a date, time, or timestamp interval and value are specified.
	INTNEST Function (p. 729)	Calculates the number of whole periods of the smaller interval that will fit into the period of the larger interval.
	INTNX Function (p. 732)	Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.
	INTRR Function (p. 742)	Returns the internal rate of return as a decimal value.
	INTSEAS Function (p. 744)	Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.
	INTSHIFT Function (p. 748)	Returns the shift interval that corresponds to the base interval.
	INTTEST Function (p. 752)	Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.
	INTTS Function (p. 755)	Specifies the number of seconds to add to a TIMESTAMP value.
	INTZ Function (p. 757)	Returns the whole number portion of the argument, using zero fuzzing.
	IPMT Function (p. 759)	Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.
	IQR Function (p. 761)	Returns the interquartile range.
	IRR Function (p. 762)	Returns the internal rate of return as a percentage.
	JBESSEL Function (p. 764)	Returns the value of the Bessel function.

Category	Language Elements	Description
	JULDATE Function (p. 765)	Returns the Julian date from a SAS date value.
	JULDATE7 Function (p. 767)	Returns a seven-digit Julian date from a SAS date value.
	KURTOSIS Function (p. 768)	Returns the kurtosis.
	LAG Function (p. 770)	Returns values from a queue.
	LARGEST Function (p. 774)	Returns the kth largest non-null or nonmissing value.
	LBOUND Function (p. 776)	Returns the lower bound of an array.
	LCM Function (p. 778)	Returns the least common multiple for a set of whole numbers.
	LCOMB Function (p. 780)	Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of n objects taken r at a time.
	LEFT Function (p. 781)	Left aligns a character expression.
	LENGTH Function (p. 782)	Returns the length of a character string, excluding trailing blanks, in characters.
	LENGTHC Function (p. 784)	Returns the length of a character string, including trailing blanks, in characters.
	LENGTHM Function (p. 787)	Returns the amount of memory, in bytes, that could or might be allocated for a character string.
	LFACT Function (p. 789)	Computes the logarithm of the FACT (factorial) function.
	LGAMMA Function (p. 790)	Returns the natural logarithm of the Gamma function.
	LOG Function (p. 791)	Returns the natural logarithm (base e) of a numeric value expression.
	LOG10 Function (p. 792)	Returns the base-10 logarithm of a numeric value expression.
	LOG1PX Function (p. 793)	Returns the log of 1 plus the argument.
	LOG2 Function (p. 795)	Returns the base 2 logarithm of a numeric value expression.
	LOGBETA Function (p. 796)	Returns the logarithm of the beta function.
	LOGCDF Function (p. 798)	Returns the logarithm of a left cumulative distribution function.
	LOGISTIC Function (p. 800)	Returns the logistic transformation of the argument.
	LOGPDF Function (p. 801)	Computes the logarithm of the probability density (mass) function from various continuous and discrete distributions.

Category	Language Elements	Description
	LOGSDF Function (p. 804)	Returns the logarithm of a survival function.
	LOWCASE Function (p. 806)	Converts all letters in a character expression to lowercase.
	MAD Function (p. 808)	Returns the median absolute deviation from the median.
	MARGRCLPRC Function (p. 809)	Calculates call prices for European options on stocks, based on the Margrabe model.
	MARGRPTPRC Function (p. 812)	Calculates put prices for European options on stocks, based on the Margrabe model.
	MAX Function (p. 815)	Returns the largest value from a list of arguments.
	MD5 Function (p. 816)	Returns the result of the message digest of a specified string in binary format.
	MDY Function (p. 818)	Returns a SAS date value from month, day, and year values.
	MEAN Function (p. 820)	Returns the arithmetic mean (average) of the non-null or nonmissing arguments.
	MEDIAN Function (p. 821)	Returns the median value.
	MIN Function (p. 823)	Returns the smallest value.
	MINUTE Function (p. 824)	Returns the minute from a SAS time or datetime value.
	MISSING Function (p. 826)	Returns a number that indicates whether the argument contains a missing value.
	MOD Function (p. 829)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.
	MODZ Function (p. 831)	Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.
	MONTH Function (p. 834)	Returns a number that represents the month from a SAS date value.
	MORT Function (p. 835)	Returns amortization parameters.
	N Function (p. 837)	Returns the number of non-null or nonmissing numeric values.
	NDIMS Function (p. 838)	Returns the number of dimensions in an array.
	NETPV Function (p. 840)	Returns the net present value as a percent.
	NMISS Function (p. 842)	Returns the number of null and SAS missing numeric values.
	NOMRATE Function (p. 844)	Returns the nominal annual interest rate.

Category	Language Elements	Description
	NOTALNUM Function (p. 845)	Searches a character string for a non-alphanumeric character, and returns the first character position at which the character is found.
	NOTALPHA Function (p. 848)	Searches a character string for a nonalphabetic character, and returns the first character position at which the character is found.
	NOTCNTRL Function (p. 850)	Searches a character string for a character that is not a control character, and returns the first character position at which that character is found.
	NOTDIGIT Function (p. 852)	Searches a character string for any character that is not a digit, and returns the first character position at which that character is found.
	NOTFIRST Function (p. 855)	Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTGRAPH Function (p. 857)	Searches a character string for a non-graphical character, and returns the first character position at which that character is found.
	NOTLOWER Function (p. 860)	Searches a character string for a character that is not a lowercase letter, and returns the first character position at which that character is found.
	NOTNAME Function (p. 862)	Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTPRINT Function (p. 864)	Searches a character string for a nonprintable character, and returns the first character position at which that character is found.
	NOTPUNCT Function (p. 866)	Searches a character string for a character that is not a punctuation character, and returns the first character position at which that character is found.
	NOTSPACE Function (p. 869)	Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.
	NOTUPPER Function (p. 872)	Searches a character string for a character that is not an uppercase letter, and returns the first character position at which that character is found.
	NOTXDIGIT Function (p. 874)	Searches a character string for a character that is not a hexadecimal character, and returns the first character position at which that character is found.
	NPV Function (p. 876)	Returns the net present value with the rate expressed as a percentage.

Category	Language Elements	Description
	NULL Function (p. 878)	Returns a 1 if the argument is null and a 0 if the argument is not null.
	NWKDOM Function (p. 880)	Returns the date for the nth occurrence of a weekday for the specified month and year.
	ORDINAL Function (p. 883)	Orders a list of values, and returns a value that is based on a position in the list.
	PCTL Function (p. 884)	Returns the percentile that corresponds to the percentage.
	PDF Function (p. 886)	Returns a value from a probability density (mass) distribution.
	PDF BERNOULLI Distribution Function (p. 888)	Returns a value from the Bernoulli probability density (mass) distribution.
	PDF BETA Distribution Function (p. 890)	Returns a value from the beta probability density (mass) distribution.
	PDF BINOMIAL Distribution Function (p. 892)	Returns a value from the binomial probability density (mass) distribution.
	PDF CAUCHY Distribution Function (p. 894)	Returns a value from the Cauchy probability density (mass) distribution.
	PDF Chi-Square Distribution Function (p. 896)	Returns a value from the chi-square probability density (mass) distribution.
	PDF Conway-Maxwell-Poisson Distribution Function (p. 898)	Returns a value from the Conway-Maxwell-Poisson probability density (mass) distribution.
	PDF EXPONENTIAL Distribution Function (p. 901)	Returns a value from the exponential probability density (mass) distribution.
	PDF F Distribution Function (p. 902)	Returns a value from the F probability density (mass) distribution.
	PDF GAMMA Distribution Function (p. 904)	Returns a value from the gamma probability density (mass) distribution.
	PDF Generalized Poisson Distribution Function (p. 906)	Returns a value from the generalized Poisson probability density (mass) distribution.
	PDF GEOMETRIC Distribution Function (p. 908)	Returns a value from the geometric probability density (mass) distribution.
	PDF Hypergeometric Distribution Function (p. 909)	Returns a value from a hypergeometric probability density (mass) distribution.
	PDF LAPLACE Distribution Function (p. 911)	Returns a value from the Laplace probability density (mass) distribution.

Category	Language Elements	Description
	PDF LOGISTIC Distribution Function (p. 913)	Returns a value from the logistic probability density (mass) distribution.
	PDF LOGNORMAL Distribution Function (p. 915)	Returns a value from the lognormal probability density (mass) distribution.
	PDF NEGBINOMIAL Distribution Function (p. 916)	Returns the value from the negative binomial probability density (mass) distribution.
	PDF NORMAL Distribution Function (p. 918)	Returns a value from the normal probability density (mass) distribution.
	PDF NORMALMIX Distribution Function (p. 920)	Returns a value from the normal mixture probability density (mass) distribution.
	PDF PARETO Distribution Function (p. 922)	Returns a value from the Pareto probability density (mass) distribution.
	PDF POISSON Distribution Function (p. 924)	Returns a value from the Poisson probability density (mass) distribution.
	PDF T Distribution Function (p. 926)	Returns a value from the T probability density (mass) distribution.
	PDF TWEEDIE Distribution Function (p. 927)	Returns a value from the Tweedie probability density (mass) distribution.
	PDF UNIFORM Distribution Function (p. 930)	Returns a value from the uniform probability density (mass) distribution.
	PDF Wald (Inverse Gaussian) Distribution Function (p. 932)	Returns a value from the Wald (also known as the inverse Gaussian) probability density (mass) distribution.
	PDF WEIBULL Distribution Function (p. 933)	Returns a value from the Weibull probability density (mass) distribution.
	PERM Function (p. 935)	Computes the number of permutations of n items that are taken r at a time.
	PMT Function (p. 937)	Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.
	POISSON Function (p. 939)	Returns the probability from a Poisson distribution.
	POWER Function (p. 940)	Returns the value of a numeric value expression raised to a specified power.
	PPMT Function (p. 941)	Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.

Category	Language Elements	Description
	PROBBETA Function (p. 943)	Returns the probability from a beta distribution.
	PROBBNML Function (p. 945)	Returns the probability from a binomial distribution.
	PROBBNRM Function (p. 946)	Returns a probability from a bivariate normal distribution.
	PROBCHI Function (p. 947)	Returns the probability from a chi-square distribution.
	PROBDF Function (p. 949)	Calculates significance probabilities for Dickey-Fuller tests for unit roots in time series.
	PROBF Function (p. 955)	Returns the probability from an F distribution.
	PROBGAM Function (p. 957)	Returns the probability from a gamma distribution.
	PROBHYPR Function (p. 958)	Returns the probability from a hypergeometric distribution.
	PROBIT Function (p. 959)	Returns a quantile from the standard normal distribution.
	PROBMC Function (p. 961)	Returns a probability or a quantile from various distributions for multiple comparisons of means.
	PROBMED Function (p. 972)	Computes cumulative probabilities for the sample median.
	PROBNEGB Function (p. 973)	Returns the probability from a negative binomial distribution.
	PROBNORM Function (p. 975)	Returns the probability from the standard normal distribution.
	PROBT Function (p. 976)	Returns the probability from a t distribution.
	PRXCHANGE Function (p. 977)	Performs a pattern-matching replacement.
	PRXMATCH Function (p. 981)	Searches for a pattern match and returns the position at which the pattern is found.
	PRXPARSE Function (p. 985)	Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.
	PRXPOSN Function (p. 988)	Returns a character string that contains the value for a capture buffer.
	PUT Function (p. 992)	Returns a value using a specified format.
	PVP Function (p. 995)	Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.
	QTR Function (p. 997)	Returns the quarter of the year from a SAS date value.

Category	Language Elements	Description
	QUANTILE Function (p. 998)	Returns the quantile from a distribution when you specify the left probability (CDF).
	QUOTE Function (p. 1002)	Adds double quotation marks to a character value.
	RAND Function (p. 1004)	Generates random numbers from a distribution that you specify.
	RANGE Function (p. 1026)	Returns the difference between the largest and the smallest values.
	RANK Function (p. 1027)	Returns the position of a character in the ASCII collating sequence.
	REPEAT Function (p. 1029)	Repeats a character expression.
	REVERSE Function (p. 1031)	Reverses a character expression.
	RIGHT Function (p. 1032)	Right aligns a character expression.
	RMS Function (p. 1033)	Returns the root mean square.
	ROUND Function (p. 1035)	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.
	ROUNDE Function (p. 1044)	Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.
	ROUNDZ Function (p. 1047)	Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.
	SAVINGS Function (p. 1052)	Returns the balance of a periodic savings by using variable interest rates.
	SCAN Function (p. 1055)	Returns the nth word from a character expression.
	SDF Function (p. 1065)	Returns a survival function.
	SEC Function (p. 1069)	Returns the secant.
	SECOND Function (p. 1071)	Returns the second from a SAS time or datetime value.
	SHA256 Function (p. 1073)	Returns the result of the message digest of a specified string.
	SHA256HEX Function (p. 1074)	Returns the result of the message digest of a specified string and converts the string to hexadecimal representation.

Category	Language Elements	Description
	SHA256HMACHEX Function (p. 1077)	Returns the result of the message digest of a specified string by using the Hash-based Message Authentication (HMAC) algorithm.
	SIGN Function (p. 1079)	Returns a number that indicates the sign of a numeric value expression.
	SIN Function (p. 1080)	Returns the trigonometric sine.
	SINH Function (p. 1081)	Returns the hyperbolic sine.
	SKEWNESS Function (p. 1082)	Returns the skewness.
	SLEEP Function (p. 1083)	For a specified period of time, suspends the execution of a program that invokes this function.
	SMALLEST Function (p. 1085)	Returns the kth smallest non-null or nonmissing value.
	SQRT Function (p. 1089)	Returns the square root of a value.
	SQUANTILE Function (p. 1090)	Returns the quantile from a distribution when you specify the right probability (SDF).
	STD Function (p. 1094)	Returns the standard deviation.
	STDERR Function (p. 1095)	Returns the standard error of the mean.
	STREAMINIT Function (p. 1096)	Specifies a random-number generator and seed value for generating random numbers.
	STRIP Function (p. 1102)	Returns a character string with all leading and trailing blanks removed.
	SUBSTR (left of =) Function (p. 1105)	Replaces a substring of content in a character variable.
	SUBSTR (right of =) Function (p. 1108)	Extracts a substring from an argument.
	SUBSTRN Function (p. 1111)	Returns a substring, allowing a result with a length of zero.
	SUM Function (p. 1116)	Returns the sum of the non-null or nonmissing arguments.
	SUMABS Function (p. 1118)	Returns the sum of the absolute values of the nonmissing arguments.
	SYSGET Function (p. 1119)	Returns the value of the specified operating environment variable.
	TAN Function (p. 1121)	Returns the tangent.
	TANH Function (p. 1122)	Returns the hyperbolic tangent.

Category	Language Elements	Description
	TIME Function (p. 1124)	Returns the current time of day as a numeric SAS time value.
	TIMEPART Function (p. 1125)	Extracts a time value from a SAS datetime value.
	TIMEVALUE Function (p. 1126)	Returns the equivalent of a reference amount at a base date by using variable interest rates.
	TINV Function (p. 1129)	Returns a quantile from the t distribution.
	TNONCT Function (p. 1131)	Returns the value of the noncentrality parameter from the Student's t distribution.
	TO_DATE Function (p. 1133)	Returns a DATE value from a DOUBLE value that specifies a SAS date value.
	TO_DOUBLE Function (p. 1134)	Returns a DOUBLE value that specifies a SAS date, time, or datetime value, from a DATE, TIME, or TIMESTAMP value.
	TO_TIME Function (p. 1137)	Returns a TIME value from a DOUBLE value that specifies a SAS time value.
	TO_TIMESTAMP Function (p. 1139)	Returns a TIMESTAMP value from a DOUBLE value that specifies a SAS time value.
	TODAY Function (p. 1140)	Returns the current date as a numeric SAS date value.
	TRANSLATE Function (p. 1141)	Replaces specific characters in a character expression.
	TRANSTRN Function (p. 1143)	Replaces or removes all occurrences of a substring in a character string.
	TRANWRD Function (p. 1146)	Replaces all occurrences of a substring in a character string.
	TRIGAMMA Function (p. 1150)	Returns the value of the trigamma function.
	TRIM Function (p. 1151)	Removes trailing blanks from a character expression, and returns a string with a length of zero if the expression is missing.
	TRUNC Function (p. 1153)	Truncates a numeric value to a specified length.
	UPCASE Function (p. 1155)	Converts all letters in an argument to uppercase.
	URLDECODE Function (p. 1157)	Returns a string that was decoded using the URL escape syntax.
	URLENCODE Function (p. 1158)	Returns a string that was encoded using the URL escape syntax.
	USS Function (p. 1160)	Returns the uncorrected sum of squares.

Category	Language Elements	Description
	UUIDGEN Function (p. 1162)	Returns the short form of a Universally Unique Identifier (UUID).
	VAR Function (p. 1163)	Returns the variance.
	VERIFY Function (p. 1164)	Returns the position of the first character that is unique to an expression.
	VFORMAT Function (p. 1166)	Returns the format that is associated with the specified variable.
	VINARRAY Function (p. 1167)	Returns a value that indicates whether the specified variable is a member of an array.
	VINFORMAT Function (p. 1169)	Returns the informat that is associated with the specified variable.
	VLABEL Function (p. 1171)	Returns the label that is associated with the specified variable.
	VLENGTH Function (p. 1172)	Returns the size of the specified variable.
	VNAME Function (p. 1174)	Returns the name of the specified variable.
	VTYPE Function (p. 1175)	Returns the full name of the data type that is associated with a variable.
	WEEK Function (p. 1177)	Returns the week-number value.
	WEEKDAY Function (p. 1181)	From a SAS date value, returns a whole number that corresponds to the day of the week.
	WHICHC Function (p. 1182)	Returns the first position of a character string from a list of character strings.
	WHICHN Function (p. 1183)	Returns the first position of a number from a list of numbers.
	YEAR Function (p. 1185)	Returns the year from a SAS date value.
	YIELDP Function (p. 1186)	Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.
	YRDIF Function (p. 1188)	Returns the difference in years between two dates according to specified day count conventions; returns a person's age.
	YYQ Function (p. 1191)	Returns a SAS date value from year and quarter year values.
Character	ANYALNUM Function (p. 292)	Searches a character string for an alphanumeric character, and returns the first character position at which the character is found.

Category	Language Elements	Description
	ANYALPHA Function (p. 295)	Searches a character string for an alphabetic character, and returns the first character position at which the character is found.
	ANYCNTRL Function (p. 298)	Searches a character string for a control character, and returns the first character position at which that character is found.
	ANYDIGIT Function (p. 300)	Searches a character string for a digit, and returns the first character position at which the digit is found.
	ANYFIRST Function (p. 302)	Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	ANYGRAPH Function (p. 304)	Searches a character string for a graphical character, and returns the first character position at which that character is found.
	ANYLOWER Function (p. 307)	Searches a character string for a lowercase letter, and returns the first character position at which the letter is found.
	ANYNAME Function (p. 309)	Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	ANYPRINT Function (p. 311)	Searches a character string for a printable character, and returns the first character position at which that character is found.
	ANYPUNCT Function (p. 314)	Searches a character string for a punctuation character, and returns the first character position at which that character is found.
	ANYSPACE Function (p. 317)	Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.
	ANYUPPER Function (p. 319)	Searches a character string for an uppercase letter, and returns the first character position at which the letter is found.
	ANYXDIGIT Function (p. 321)	Searches a character string for a hexadecimal character that represents a digit, and returns the first character position at which that character is found.
	BYTE Function (p. 353)	Returns one character in the ASCII or the EBCDIC collating sequence.
	CAT Function (p. 354)	Does not remove leading or trailing blanks, and returns a concatenated character string.

Category	Language Elements	Description
	CATQ Function (p. 356)	Concatenates character or numeric values by using a delimiter to separate items and by adding quotation marks to strings that contain the delimiter.
	CATS Function (p. 361)	Removes leading and trailing blanks, and returns a concatenated character string.
	CATT Function (p. 364)	Removes trailing blanks, and returns a concatenated character string.
	CATX Function (p. 366)	Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.
	CHOOSEC Function (p. 422)	Returns a character value that represents the results of choosing from a list of arguments.
	CMP Function (p. 427)	Compares two character strings including trailing blanks.
	CMPT Function (p. 428)	Compares two character strings excluding trailing blanks.
	COALESCEC Function (p. 434)	Returns the first non-null or nonmissing value from a list of character arguments.
	COMPARE Function (p. 437)	Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.
	COMPBL Function (p. 441)	Removes multiple blanks from a character string.
	COMPCOST Function (p. 443)	Sets the costs of operations for later use by the COMPGED function.
	COMPGED Function (p. 448)	Returns the generalized edit distance between two strings.
	COMPLEV Function (p. 455)	Returns the Levenshtein edit distance between two strings.
	COMPRESS Function (p. 460)	Returns a character string with specified characters removed from the original string.
	COUNT Function (p. 478)	Counts the number of times that a specified substring appears within a character string.
	COUNTC Function (p. 481)	Counts the number of characters in a string that appear or do not appear in a list of characters.
	COUNTW Function (p. 484)	Counts the number of words in a character string.
	DEQUOTE Function (p. 507)	Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.
	FIND Function (p. 625)	Searches for a specific substring of characters within a character string.
	FINDC Function (p. 628)	Searches a string for any character in a list of characters.

Category	Language Elements	Description
	FINDW Function (p. 637)	Returns the character position of a word in a string, or returns the number of the word in a string.
	INDEX Function (p. 685)	Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.
	INDEXC Function (p. 687)	Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.
	INDEXW Function (p. 689)	Searches a character expression for a string that is specified as a word, and returns the position of the first character in the word.
	LEFT Function (p. 781)	Left aligns a character expression.
	LENGTH Function (p. 782)	Returns the length of a character string, excluding trailing blanks, in characters.
	LENGTHC Function (p. 784)	Returns the length of a character string, including trailing blanks, in characters.
	LENGTHM Function (p. 787)	Returns the amount of memory, in bytes, that could or might be allocated for a character string.
	LOWCASE Function (p. 806)	Converts all letters in a character expression to lowercase.
	MD5 Function (p. 816)	Returns the result of the message digest of a specified string in binary format.
	NOTALNUM Function (p. 845)	Searches a character string for a non-alphanumeric character, and returns the first character position at which the character is found.
	NOTALPHA Function (p. 848)	Searches a character string for a nonalphabetic character, and returns the first character position at which the character is found.
	NOTCNTRL Function (p. 850)	Searches a character string for a character that is not a control character, and returns the first character position at which that character is found.
	NOTDIGIT Function (p. 852)	Searches a character string for any character that is not a digit, and returns the first character position at which that character is found.
	NOTFIRST Function (p. 855)	Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTGRAPH Function (p. 857)	Searches a character string for a non-graphical character, and returns the first character position at which that character is found.

Category	Language Elements	Description
	NOTLOWER Function (p. 860)	Searches a character string for a character that is not a lowercase letter, and returns the first character position at which that character is found.
	NOTNAME Function (p. 862)	Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.
	NOTPRINT Function (p. 864)	Searches a character string for a nonprintable character, and returns the first character position at which that character is found.
	NOTPUNCT Function (p. 866)	Searches a character string for a character that is not a punctuation character, and returns the first character position at which that character is found.
	NOTSPACE Function (p. 869)	Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.
	NOTUPPER Function (p. 872)	Searches a character string for a character that is not an uppercase letter, and returns the first character position at which that character is found.
	NOTXDIGIT Function (p. 874)	Searches a character string for a character that is not a hexadecimal character, and returns the first character position at which that character is found.
	QUOTE Function (p. 1002)	Adds double quotation marks to a character value.
	RANK Function (p. 1027)	Returns the position of a character in the ASCII collating sequence.
	REPEAT Function (p. 1029)	Repeats a character expression.
	REVERSE Function (p. 1031)	Reverses a character expression.
	RIGHT Function (p. 1032)	Right aligns a character expression.
	SCAN Function (p. 1055)	Returns the nth word from a character expression.
	SHA256 Function (p. 1073)	Returns the result of the message digest of a specified string.
	SHA256HEX Function (p. 1074)	Returns the result of the message digest of a specified string and converts the string to hexadecimal representation.
	SHA256HMACHEX Function (p. 1077)	Returns the result of the message digest of a specified string by using the Hash-based Message Authentication (HMAC) algorithm.
	STRIP Function (p. 1102)	Returns a character string with all leading and trailing blanks removed.

Category	Language Elements	Description
	SUBSTR (left of =) Function (p. 1105)	Replaces a substring of content in a character variable.
	SUBSTR (right of =) Function (p. 1108)	Extracts a substring from an argument.
	SUBSTRN Function (p. 1111)	Returns a substring, allowing a result with a length of zero.
	TRANSLATE Function (p. 1141)	Replaces specific characters in a character expression.
	TRANSTRN Function (p. 1143)	Replaces or removes all occurrences of a substring in a character string.
	TRANWRD Function (p. 1146)	Replaces all occurrences of a substring in a character string.
	TRIM Function (p. 1151)	Removes trailing blanks from a character expression, and returns a string with a length of zero if the expression is missing.
	UPCASE Function (p. 1155)	Converts all letters in an argument to uppercase.
	VERIFY Function (p. 1164)	Returns the position of the first character that is unique to an expression.
	WHICHC Function (p. 1182)	Returns the first position of a character string from a list of character strings.
Character String Matching	PRXCHANGE Function (p. 977)	Performs a pattern-matching replacement.
	PRXMATCH Function (p. 981)	Searches for a pattern match and returns the position at which the pattern is found.
	PRXPARSE Function (p. 985)	Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.
	PRXPOSN Function (p. 988)	Returns a character string that contains the value for a capture buffer.
Combinatorial	COMB Function (p. 435)	Computes the number of combinations of n elements taken r at a time.
	LCOMB Function (p. 780)	Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of n objects taken r at a time.
	LFACT Function (p. 789)	Computes the logarithm of the FACT (factorial) function.
	PERM Function (p. 935)	Computes the number of permutations of n items that are taken r at a time.

Category	Language Elements	Description
Date and Time	DATDIF Function (p. 497)	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.
	DATE Function (p. 500)	Returns the current date as a SAS date value.
	DATEJUL Function (p. 501)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 503)	Extracts the date from a SAS datetime value.
	DATETIME Function (p. 504)	Returns the current date and time of day as a SAS datetime value.
	DAY Function (p. 505)	Returns the day of the month from a SAS date value.
	DHMS Function (p. 515)	Returns a SAS datetime value from date, hour, minute, and second values.
	HMS Function (p. 676)	Returns a SAS time value from hour, minute, and second values.
	HOLIDAY Function (p. 677)	Returns a SAS date value of a specified holiday for a specified year.
	HOUR Function (p. 682)	Returns the hour from a SAS time or datetime value.
	INTCINDEX Function (p. 696)	Returns the cycle index when a date, time, or timestamp interval and value are specified.
	INTCK Function (p. 701)	Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.
	INTCYCLE Function (p. 710)	Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.
	INTDT Function (p. 715)	Specifies the number of days to add to a DATE value.
	INTFIT Function (p. 717)	Returns a time interval that is aligned between two dates.
	INTGET Function (p. 719)	Returns a time interval based on three date or datetime values.
	INTINDEX Function (p. 722)	Returns the seasonal index when a date, time, or timestamp interval and value are specified.
	INTNEST Function (p. 729)	Calculates the number of whole periods of the smaller interval that will fit into the period of the larger interval.
	INTNX Function (p. 732)	Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.

Category	Language Elements	Description
	INTSEAS Function (p. 744)	Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.
	INTSHIFT Function (p. 748)	Returns the shift interval that corresponds to the base interval.
	INTTEST Function (p. 752)	Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.
	INTTS Function (p. 755)	Specifies the number of seconds to add to a TIMESTAMP value.
	JULDATE Function (p. 765)	Returns the Julian date from a SAS date value.
	JULDATE7 Function (p. 767)	Returns a seven-digit Julian date from a SAS date value.
	MDY Function (p. 818)	Returns a SAS date value from month, day, and year values.
	MINUTE Function (p. 824)	Returns the minute from a SAS time or datetime value.
	MONTH Function (p. 834)	Returns a number that represents the month from a SAS date value.
	NWKDOM Function (p. 880)	Returns the date for the nth occurrence of a weekday for the specified month and year.
	QTR Function (p. 997)	Returns the quarter of the year from a SAS date value.
	SECOND Function (p. 1071)	Returns the second from a SAS time or datetime value.
	TIME Function (p. 1124)	Returns the current time of day as a numeric SAS time value.
	TIMEPART Function (p. 1125)	Extracts a time value from a SAS datetime value.
	TO_DATE Function (p. 1133)	Returns a DATE value from a DOUBLE value that specifies a SAS date value.
	TO_DOUBLE Function (p. 1134)	Returns a DOUBLE value that specifies a SAS date, time, or datetime value, from a DATE, TIME, or TIMESTAMP value.
	TO_TIME Function (p. 1137)	Returns a TIME value from a DOUBLE value that specifies a SAS time value.
	TO_TIMESTAMP Function (p. 1139)	Returns a TIMESTAMP value from a DOUBLE value that specifies a SAS time value.
	TODAY Function (p. 1140)	Returns the current date as a numeric SAS date value.
	WEEK Function (p. 1177)	Returns the week-number value.
	WEEKDAY Function (p. 1181)	From a SAS date value, returns a whole number that corresponds to the day of the week.

Category	Language Elements	Description
	YEAR Function (p. 1185)	Returns the year from a SAS date value.
	YRDIF Function (p. 1188)	Returns the difference in years between two dates according to specified day count conventions; returns a person's age.
	YYQ Function (p. 1191)	Returns a SAS date value from year and quarter year values.
Descriptive Statistics	CMISS Function (p. 425)	Counts the number of missing arguments.
	CSS Function (p. 490)	Returns the corrected sum of squares.
	CV Function (p. 495)	Returns the coefficient of variation.
	GEOMEAN Function (p. 666)	Returns the geometric mean.
	GEOMEANZ Function (p. 668)	Returns the geometric mean, using zero fuzzing.
	HARMEAN Function (p. 670)	Returns the harmonic mean.
	HARMEANZ Function (p. 672)	Returns the harmonic mean, using zero fuzzing.
	IQR Function (p. 761)	Returns the interquartile range.
	KURTOSIS Function (p. 768)	Returns the kurtosis.
	LARGEST Function (p. 774)	Returns the kth largest non-null or nonmissing value.
	MAD Function (p. 808)	Returns the median absolute deviation from the median.
	MAX Function (p. 815)	Returns the largest value from a list of arguments.
	MEAN Function (p. 820)	Returns the arithmetic mean (average) of the non-null or nonmissing arguments.
	MEDIAN Function (p. 821)	Returns the median value.
	MIN Function (p. 823)	Returns the smallest value.
	NMISS Function (p. 842)	Returns the number of null and SAS missing numeric values.
	ORDINAL Function (p. 883)	Orders a list of values, and returns a value that is based on a position in the list.
	PCTL Function (p. 884)	Returns the percentile that corresponds to the percentage.
	RANGE Function (p. 1026)	Returns the difference between the largest and the smallest values.
	RMS Function (p. 1033)	Returns the root mean square.

Category	Language Elements	Description
	SKEWNESS Function (p. 1082)	Returns the skewness.
	SMALLEST Function (p. 1085)	Returns the kth smallest non-null or nonmissing value.
	STD Function (p. 1094)	Returns the standard deviation.
	STDERR Function (p. 1095)	Returns the standard error of the mean.
	SUM Function (p. 1116)	Returns the sum of the non-null or nonmissing arguments.
	SUMABS Function (p. 1118)	Returns the sum of the absolute values of the nonmissing arguments.
	USS Function (p. 1160)	Returns the uncorrected sum of squares.
	VAR Function (p. 1163)	Returns the variance.
Distance	GEODIST Function (p. 663)	Returns the geodetic distance between two latitude and longitude coordinates.
Financial	BLACKCLPRC Function (p. 337)	Calculates call prices for European options on futures, based on the Black model.
	BLACKPTPRC Function (p. 340)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 342)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	BLKSHPTPRC Function (p. 345)	Calculates put prices for European options on stocks, based on the Black-Scholes model.
	COMPOUND Function (p. 458)	Returns compound interest parameters.
	CONVX Function (p. 471)	Returns the convexity for an enumerated cash flow.
	CONVXP Function (p. 472)	Returns the convexity for a periodic cash flow stream, such as a bond.
	CUMIPMT Function (p. 491)	Returns the cumulative interest paid on a loan between the start and end period.
	CUMPRINC Function (p. 493)	Returns the cumulative principal paid on a loan between the start and end period.
	DUR Function (p. 525)	Returns the modified duration for an enumerated cash flow.
	DURP Function (p. 527)	Returns the modified duration for a periodic cash flow stream, such as a bond.
	EFFRATE Function (p. 529)	Returns the effective annual interest rate.

Category	Language Elements	Description
	FINANCE Function (p. 537)	Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
	FINANCE ACCRINT Function (p. 541)	Computes the accrued interest for a security that pays periodic interest.
	FINANCE ACCRINTM Function (p. 543)	Computes the accrued interest for a security that pays interest at maturity.
	FINANCE AMORDEGRC Function (p. 545)	Computes the depreciation for each accounting period by using a depreciation coefficient.
	FINANCE AMORLINC Function (p. 547)	Computes the depreciation for each accounting period.
	FINANCE COUPDAYBS Function (p. 549)	Computes the number of days from the beginning of the coupon period to the settlement date.
	FINANCE COUPDAYS Function (p. 551)	Computes the number of days in the coupon period that contains the settlement date.
	FINANCE COUPDAYSNC Function (p. 552)	Computes the number of days from the settlement date to the next coupon date.
	FINANCE COUPNCD Function (p. 554)	Computes the next coupon date after the settlement date.
	FINANCE COUPNUM Function (p. 556)	Computes the number of coupons that are payable between the settlement date and the maturity date.
	FINANCE COUPPCD Function (p. 557)	Computes the previous coupon date before the settlement date.
	FINANCE CUMIPMT Function (p. 559)	Computes the cumulative interest paid between two periods.
	FINANCE CUMPRINC Function (p. 561)	Computes the cumulative principal that is paid on a loan between two periods.
	FINANCE DB Function (p. 562)	Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
	FINANCE DDB Function (p. 564)	Computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.
	FINANCE DISC Function (p. 565)	Computes the discount rate for a security.
	FINANCE DOLLARDE Function (p. 567)	Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

Category	Language Elements	Description
	FINANCE DOLLARFR Function (p. 568)	Converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.
	FINANCE DURATION Function (p. 569)	Computes the annual duration of a security with periodic interest payments.
	FINANCE EFFECT Function (p. 571)	Computes the effective annual interest rate.
	FINANCE FV Function (p. 572)	Computes the future value of an investment.
	FINANCE FVSCHEDULE Function (p. 573)	Computes the future value of the initial principal after applying a series of compound interest rates.
	FINANCE INTRATE Function (p. 574)	Computes the interest rate for a fully invested security.
	FINANCE IPMT Function (p. 576)	Computes the interest payment for an investment for a specified period.
	FINANCE IRR Function (p. 578)	Computes the internal rate of return for a series of cash flows.
	FINANCE ISPMT Function (p. 579)	Calculates the interest paid during a specific period of an investment.
	FINANCE MDURATION Function (p. 580)	Computes the Macaulay modified duration for a security with an assumed face value of $100.
	FINANCE MIRR Function (p. 582)	Computes the internal rate of return where positive and negative cash flows are financed at different rates.
	FINANCE NOMINAL Function (p. 583)	Computes the annual nominal interest rates.
	FINANCE NPER Function (p. 584)	Computes the number of periods for an investment.
	FINANCE NPV Function (p. 586)	Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
	FINANCE ODDFPRICE Function (p. 587)	Computes the price of a security per $100 face value with an odd first period.
	FINANCE ODDFYIELD Function (p. 589)	Computes the yield of a security with an odd first period.
	FINANCE ODDLPRICE Function (p. 591)	Computes the price of a security per $100 face value with an odd last period.
	FINANCE ODDLYIELD Function (p. 594)	Computes the yield of a security with an odd last period.

Category	Language Elements	Description
	FINANCE PMT Function (p. 596)	Computes the periodic payment of an annuity.
	FINANCE PPMT Function (p. 597)	Computes the payment on the principal for an investment for a specified period.
	FINANCE PRICE Function (p. 599)	Computes the price of a security per $100 face value that pays periodic interest.
	FINANCE PRICEDISC Function (p. 601)	Computes the price of a discounted security per $100 face value.
	FINANCE PRICEMAT Function (p. 603)	Computes the price of a security per $100 face value that pays interest at maturity.
	FINANCE PV Function (p. 605)	Computes the present value of an investment.
	FINANCE RATE Function (p. 606)	Computes the interest rate per period of an annuity.
	FINANCE RECEIVED Function (p. 608)	Computes the amount that is received at maturity for a fully invested security.
	FINANCE SLN Function (p. 609)	Computes the straight-line depreciation of an asset for one period.
	FINANCE SYD Function (p. 611)	Computes the sum-of-years digits depreciation of an asset for a specified period.
	FINANCE TBILLEQ Function (p. 612)	Computes the bond-equivalent yield for a treasury bill.
	FINANCE TBILLPRICE Function (p. 613)	Computes the price of a treasury bill per $100 face value.
	FINANCE TBILLYIELD Function (p. 614)	Computes the yield for a treasury bill.
	FINANCE VDB Function (p. 615)	Computes the depreciation of an asset for a specified or partial period by using a declining balance method.
	FINANCE XIRR Function (p. 617)	Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
	FINANCE XNPV Function (p. 618)	Computes the net present value for a schedule of cash flows that is not necessarily periodic.
	FINANCE YIELD Function (p. 620)	Computes the yield on a security that pays periodic interest.
	FINANCE YIELDDISC Function (p. 622)	Computes the annual yield for a discounted security (for example, a treasury bill).

Category	Language Elements	Description
	FINANCE YIELDMAT Function (p. 623)	Computes the annual yield of a security that pays interest at maturity.
	GARKHCLPRC Function (p. 656)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
	GARKHPTPRC Function (p. 659)	Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.
	INTRR Function (p. 742)	Returns the internal rate of return as a decimal value.
	IPMT Function (p. 759)	Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.
	IRR Function (p. 762)	Returns the internal rate of return as a percentage.
	MARGRCLPRC Function (p. 809)	Calculates call prices for European options on stocks, based on the Margrabe model.
	MARGRPTPRC Function (p. 812)	Calculates put prices for European options on stocks, based on the Margrabe model.
	MORT Function (p. 835)	Returns amortization parameters.
	NETPV Function (p. 840)	Returns the net present value as a percent.
	NOMRATE Function (p. 844)	Returns the nominal annual interest rate.
	NPV Function (p. 876)	Returns the net present value with the rate expressed as a percentage.
	PMT Function (p. 937)	Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.
	PPMT Function (p. 941)	Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.
	PVP Function (p. 995)	Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.
	SAVING Function (p. 1050)	Returns the future value of a periodic saving.
	SAVINGS Function (p. 1052)	Returns the balance of a periodic savings by using variable interest rates.
	TIMEVALUE Function (p. 1126)	Returns the equivalent of a reference amount at a base date by using variable interest rates.
	YIELDP Function (p. 1186)	Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.
Hyperbolic	ARCOSH Function (p. 325)	Returns the inverse hyperbolic cosine.

Category	Language Elements	Description
	ARSINH Function (p. 327)	Returns the inverse hyperbolic sine.
	ARTANH Function (p. 329)	Returns the inverse hyperbolic tangent.
	SINH Function (p. 1081)	Returns the hyperbolic sine.
	TANH Function (p. 1122)	Returns the hyperbolic tangent.
Mathematical	ABS Function (p. 289)	Returns the absolute value of a numeric value expression.
	AIRY Function (p. 291)	Returns the value of the Airy function.
	BETA Function (p. 334)	Returns the value of the beta function.
	CNONCT Function (p. 430)	Returns the noncentrality parameter from a chi-square distribution.
	COALESCE Function (p. 432)	Returns the first non-null or nonmissing value from a list of numeric arguments.
	COMPFUZZ Function (p. 445)	Performs a fuzzy comparison of two numeric values.
	CONSTANT Function (p. 465)	Computes machine and mathematical constants.
	DAIRY Function (p. 496)	Returns the derivative of the AIRY function.
	DEVIANCE Function (p. 511)	Returns the deviance based on a probability distribution.
	DIGAMMA Function (p. 520)	Returns the value of the digamma function.
	ERF Function (p. 531)	Returns the value of the (normal) error function.
	ERFC Function (p. 532)	Returns the value of the complementary (normal) error function.
	EXP Function (p. 533)	Returns the value of the e constant raised to a specified power.
	FACT Function (p. 535)	Computes a factorial.
	FNONCT Function (p. 650)	Returns the value of the noncentrality parameter of an F distribution.
	GAMMA Function (p. 655)	Returns the value of the gamma function.
	GCD Function (p. 662)	Returns the greatest common divisor for a set of integers.
	IBESSEL Function (p. 683)	Returns the value of the modified Bessel function.
	JBESSEL Function (p. 764)	Returns the value of the Bessel function.

Category	Language Elements	Description
	LCM Function (p. 778)	Returns the least common multiple for a set of whole numbers.
	LGAMMA Function (p. 790)	Returns the natural logarithm of the Gamma function.
	LOG Function (p. 791)	Returns the natural logarithm (base e) of a numeric value expression.
	LOG10 Function (p. 792)	Returns the base-10 logarithm of a numeric value expression.
	LOG1PX Function (p. 793)	Returns the log of 1 plus the argument.
	LOG2 Function (p. 795)	Returns the base 2 logarithm of a numeric value expression.
	LOGBETA Function (p. 796)	Returns the logarithm of the beta function.
	LOGISTIC Function (p. 800)	Returns the logistic transformation of the argument.
	MOD Function (p. 829)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.
	MODZ Function (p. 831)	Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.
	POWER Function (p. 940)	Returns the value of a numeric value expression raised to a specified power.
	SIGN Function (p. 1079)	Returns a number that indicates the sign of a numeric value expression.
	SQRT Function (p. 1089)	Returns the square root of a value.
	TNONCT Function (p. 1131)	Returns the value of the noncentrality parameter from the Student's t distribution.
	TRIGAMMA Function (p. 1150)	Returns the value of the trigamma function.
	WHICHN Function (p. 1183)	Returns the first position of a number from a list of numbers.
Numeric	CHOOSEN Function (p. 424)	Returns a numeric value that represents the results of choosing from a list of arguments.
Probability	CDF Function (p. 370)	Computes the left cumulative distribution function from various continuous and discrete probability distributions.
	CDF BERNOULLI Distribution Function (p. 372)	Returns a value from the Bernoulli cumulative probability distribution.

Category	Language Elements	Description
	CDF BETA Distribution Function (p. 374)	Returns a value from the beta cumulative probability distribution.
	CDF BINOMIAL Distribution Function (p. 376)	Returns a value from the binomial cumulative probability distribution.
	CDF CAUCHY Distribution Function (p. 378)	Returns a value from the Cauchy cumulative probability distribution.
	CDF Chi-Square Distribution Function (p. 380)	Returns a value from the chi-square cumulative probability distribution.
	CDF Conway-Maxwell-Poisson Distribution Function (p. 382)	Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.
	CDF Exponential Distribution Function (p. 383)	Returns a value from the exponential cumulative probability distribution.
	CDF F Distribution Function (p. 385)	Returns a value from the F cumulative probability distribution.
	CDF GAMMA Distribution Function (p. 387)	Returns a value from the gamma cumulative probability distribution.
	CDF Generalized Poisson Distribution Function (p. 389)	Returns a value from the generalized Poisson cumulative probability distribution.
	CDF GEOMETRIC Distribution Function (p. 391)	Returns a value from the geometric cumulative probability distribution.
	CDF HYPERGEOMETRIC Distribution Function (p. 392)	Returns a value from the hypergeometric cumulative probability distribution.
	CDF LAPLACE Distribution Function (p. 395)	Returns a value from the Laplace cumulative probability distribution.
	CDF LOGISTIC Distribution Function (p. 396)	Returns a value from the logistic cumulative probability distribution.
	CDF LOGNORMAL Distribution Function (p. 398)	Returns a value from the lognormal cumulative probability distribution.
	CDF NEGBINOMIAL Distribution Function (p. 400)	Returns a value from the negative binomial cumulative probability distribution.
	CDF NORMAL Distribution Function (p. 402)	Returns a value from the normal cumulative probability distribution.

Category	Language Elements	Description
	CDF NORMALMIX Distribution Function (p. 404)	Returns a value from the normal mixture cumulative probability distribution.
	CDF PARETO Distribution Function (p. 406)	Returns a value from the Pareto cumulative probability distribution.
	CDF POISSON Distribution Function (p. 407)	Returns a value from the Poisson cumulative probability distribution.
	CDF T Distribution Function (p. 409)	Returns a value from the T cumulative probability distribution.
	CDF TWEEDIE Distribution Function (p. 411)	Returns a value from the Tweedie cumulative probability distribution.
	CDF UNIFORM Distribution Function (p. 413)	Returns a value from the uniform cumulative probability distribution.
	CDF WALD (Inverse Gaussian) Distribution Function (p. 415)	Returns a value from the Wald (also known as the inverse Gaussian) cumulative probability distribution.
	CDF WEIBULL Distribution Function (p. 417)	Returns a value from the Weibull cumulative probability distribution.
	LOGCDF Function (p. 798)	Returns the logarithm of a left cumulative distribution function.
	LOGPDF Function (p. 801)	Computes the logarithm of the probability density (mass) function from various continuous and discrete distributions.
	LOGSDF Function (p. 804)	Returns the logarithm of a survival function.
	PDF Function (p. 886)	Returns a value from a probability density (mass) distribution.
	PDF BERNOULLI Distribution Function (p. 888)	Returns a value from the Bernoulli probability density (mass) distribution.
	PDF BETA Distribution Function (p. 890)	Returns a value from the beta probability density (mass) distribution.
	PDF BINOMIAL Distribution Function (p. 892)	Returns a value from the binomial probability density (mass) distribution.
	PDF CAUCHY Distribution Function (p. 894)	Returns a value from the Cauchy probability density (mass) distribution.
	PDF Chi-Square Distribution Function (p. 896)	Returns a value from the chi-square probability density (mass) distribution.

Category	Language Elements	Description
	PDF Conway-Maxwell-Poisson Distribution Function (p. 898)	Returns a value from the Conway-Maxwell-Poisson probability density (mass) distribution.
	PDF EXPONENTIAL Distribution Function (p. 901)	Returns a value from the exponential probability density (mass) distribution.
	PDF F Distribution Function (p. 902)	Returns a value from the F probability density (mass) distribution.
	PDF GAMMA Distribution Function (p. 904)	Returns a value from the gamma probability density (mass) distribution.
	PDF Generalized Poisson Distribution Function (p. 906)	Returns a value from the generalized Poisson probability density (mass) distribution.
	PDF GEOMETRIC Distribution Function (p. 908)	Returns a value from the geometric probability density (mass) distribution.
	PDF Hypergeometric Distribution Function (p. 909)	Returns a value from a hypergeometric probability density (mass) distribution.
	PDF LAPLACE Distribution Function (p. 911)	Returns a value from the Laplace probability density (mass) distribution.
	PDF LOGISTIC Distribution Function (p. 913)	Returns a value from the logistic probability density (mass) distribution.
	PDF LOGNORMAL Distribution Function (p. 915)	Returns a value from the lognormal probability density (mass) distribution.
	PDF NEGBINOMIAL Distribution Function (p. 916)	Returns the value from the negative binomial probability density (mass) distribution.
	PDF NORMAL Distribution Function (p. 918)	Returns a value from the normal probability density (mass) distribution.
	PDF NORMALMIX Distribution Function (p. 920)	Returns a value from the normal mixture probability density (mass) distribution.
	PDF PARETO Distribution Function (p. 922)	Returns a value from the Pareto probability density (mass) distribution.
	PDF POISSON Distribution Function (p. 924)	Returns a value from the Poisson probability density (mass) distribution.
	PDF T Distribution Function (p. 926)	Returns a value from the T probability density (mass) distribution.

Category	Language Elements	Description
	PDF TWEEDIE Distribution Function (p. 927)	Returns a value from the Tweedie probability density (mass) distribution.
	PDF UNIFORM Distribution Function (p. 930)	Returns a value from the uniform probability density (mass) distribution.
	PDF Wald (Inverse Gaussian) Distribution Function (p. 932)	Returns a value from the Wald (also known as the inverse Gaussian) probability density (mass) distribution.
	PDF WEIBULL Distribution Function (p. 933)	Returns a value from the Weibull probability density (mass) distribution.
	POISSON Function (p. 939)	Returns the probability from a Poisson distribution.
	PROBBETA Function (p. 943)	Returns the probability from a beta distribution.
	PROBBNML Function (p. 945)	Returns the probability from a binomial distribution.
	PROBBNRM Function (p. 946)	Returns a probability from a bivariate normal distribution.
	PROBCHI Function (p. 947)	Returns the probability from a chi-square distribution.
	PROBDF Function (p. 949)	Calculates significance probabilities for Dickey-Fuller tests for unit roots in time series.
	PROBF Function (p. 955)	Returns the probability from an F distribution.
	PROBGAM Function (p. 957)	Returns the probability from a gamma distribution.
	PROBHYPR Function (p. 958)	Returns the probability from a hypergeometric distribution.
	PROBMC Function (p. 961)	Returns a probability or a quantile from various distributions for multiple comparisons of means.
	PROBMED Function (p. 972)	Computes cumulative probabilities for the sample median.
	PROBNEGB Function (p. 973)	Returns the probability from a negative binomial distribution.
	PROBNORM Function (p. 975)	Returns the probability from the standard normal distribution.
	PROBT Function (p. 976)	Returns the probability from a t distribution.
	SDF Function (p. 1065)	Returns a survival function.
Quantile	BETAINV Function (p. 336)	Returns a quantile from the beta distribution.
	GAMINV Function (p. 654)	Returns a quantile from the gamma distribution.

Category	Language Elements	Description
	PROBIT Function (p. 959)	Returns a quantile from the standard normal distribution.
	QUANTILE Function (p. 998)	Returns the quantile from a distribution when you specify the left probability (CDF).
	SQUANTILE Function (p. 1090)	Returns the quantile from a distribution when you specify the right probability (SDF).
	TINV Function (p. 1129)	Returns a quantile from the t distribution.
Random Number	RAND Function (p. 1004)	Generates random numbers from a distribution that you specify.
	STREAMINIT Function (p. 1096)	Specifies a random-number generator and seed value for generating random numbers.
Special	DIF Function (p. 518)	Returns differences between an argument and its nth lag.
	FMTINFO Function (p. 648)	Returns information about a SAS format or informat.
	INPUTC Function (p. 691)	Enables you to specify a character informat at run time.
	INPUTN Function (p. 693)	Enables you to specify a numeric informat at run time.
	LAG Function (p. 770)	Returns values from a queue.
	MISSING Function (p. 826)	Returns a number that indicates whether the argument contains a missing value.
	N Function (p. 837)	Returns the number of non-null or nonmissing numeric values.
	NULL Function (p. 878)	Returns a 1 if the argument is null and a 0 if the argument is not null.
	PUT Function (p. 992)	Returns a value using a specified format.
	SLEEP Function (p. 1083)	For a specified period of time, suspends the execution of a program that invokes this function.
	SQLEXEC Function (p. 1087)	Executes a FedSQL statement to create, delete, or update a table or to insert rows into a table.
	SYSGET Function (p. 1119)	Returns the value of the specified operating environment variable.
	UUIDGEN Function (p. 1162)	Returns the short form of a Universally Unique Identifier (UUID).
Trigonometric	ARCOS Function (p. 324)	Returns the arccosine in radians.
	ARSIN Function (p. 326)	Returns the arcsine in radians.
	ATAN Function (p. 330)	Returns the arctangent in radians.

Category	Language Elements	Description
	ATAN2 Function (p. 332)	Returns the arctangent of the x and y coordinates of a right triangle, in radians.
	COS Function (p. 475)	Returns the cosine in radians.
	COSH Function (p. 476)	Returns the hyperbolic cosine in radians.
	COT Function (p. 477)	Returns the cotangent.
	CSC Function (p. 489)	Returns the cosecant.
	SEC Function (p. 1069)	Returns the secant.
	SIN Function (p. 1080)	Returns the trigonometric sine.
	TAN Function (p. 1121)	Returns the tangent.
Truncation	CEIL Function (p. 419)	Returns the smallest integer greater than or equal to a numeric value expression.
	CEILZ Function (p. 421)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	FLOOR Function (p. 644)	Returns the largest integer less than or equal to a numeric value expression.
	FLOORZ Function (p. 646)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
	FUZZ Function (p. 652)	Returns the nearest whole number if the argument is within 1E-12 of that number.
	INT Function (p. 694)	Returns the whole number, fuzzed to avoid unexpected floating-point results.
	INTZ Function (p. 757)	Returns the whole number portion of the argument, using zero fuzzing.
	ROUND Function (p. 1035)	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.
	ROUNDE Function (p. 1044)	Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.
	ROUNDZ Function (p. 1047)	Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.
	TRUNC Function (p. 1153)	Truncates a numeric value to a specified length.
Variable Information	VFORMAT Function (p. 1166)	Returns the format that is associated with the specified variable.

Category	Language Elements	Description
	VINARRAY Function (p. 1167)	Returns a value that indicates whether the specified variable is a member of an array.
	VINFORMAT Function (p. 1169)	Returns the informat that is associated with the specified variable.
	VLABEL Function (p. 1171)	Returns the label that is associated with the specified variable.
	VLENGTH Function (p. 1172)	Returns the size of the specified variable.
	VNAME Function (p. 1174)	Returns the name of the specified variable.
	VTYPE Function (p. 1175)	Returns the full name of the data type that is associated with a variable.
Web Tools	URLDECODE Function (p. 1157)	Returns a string that was decoded using the URL escape syntax.
	URLENCODE Function (p. 1158)	Returns a string that was encoded using the URL escape syntax.

Dictionary

ABS Function

Returns the absolute value of a numeric value expression.

Categories: CAS
Mathematical

Returned data type: BIGINT, DECIMAL, DOUBLE

Syntax

ABS(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type BIGINT, DECIMAL, DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If the result is a number that does not fit into the range of the argument's data type, the ABS function fails.

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Examples

Example 1: Absolute Value of a DECIMAL and Negative Number

The following program illustrates the absolute value of a DECIMAL and a negative number:

```
data _null_;
   dcl double x y;
   method run();
      x=abs(2.4);
      y=abs(-3);
      put x= y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x=2.4 y=3
```

Example 2: Absolute Value of a Mathematical Expression

The following program illustrates the absolute value of a simple, mathematical expression.

```
data _null_;
   dcl double z;
   method run();
      z=abs((3 * 50.5) / 5);
      put z=;
  end;
enddata;
run;
```

SAS writes the following output to the log.

```
z=30.3
```

AIRY Function

Returns the value of the Airy function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

AIRY(*x*)

Arguments

x
specifies a numeric constant, variable, or expression.

Data type DOUBLE

Details

The AIRY function returns the value of the Airy function. (See a list of References.) It is the solution of the differential equation

$$w^{(2)} - xw = 0$$

with the conditions

$$w(0) = \frac{1}{3^{\frac{2}{3}}\Gamma\left(\frac{2}{3}\right)}$$

and

$$w'(0) = -\frac{1}{3^{\frac{1}{3}}\Gamma\left(\frac{1}{3}\right)}$$

Example

The following program illustrates the AIRY function:

```
data _null_;
   dcl double x y;
   method init();
      x=airy(2.0);
      y=airy(-2.0);
      put x= y=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=0.03492413042327 y=0.22740742820168
```

ANYALNUM Function

Searches a character string for an alphanumeric character, and returns the first character position at which the character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYALNUM('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR, VARCHAR, NVARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYALNUM function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYALNUM function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase letter. If such a character is found, ANYALNUM returns the position in the string of that character. If no such character is found, ANYALNUM returns a value of 0.

If you use only one argument, ANYALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALNUM function searches a character string for an alphanumeric character. The NOTALNUM function searches a character string for a non-alphanumeric character.

Examples

Example 1: Scanning a String from Left to Right

The following program uses the ANYALNUM function to search a string from left to right for alphanumeric characters.

```
data ltrtest;
   dcl char(15) string c;
   dcl double j;
```

```
      method run();
         string='Next = Last + 1';
         j=0;
         do until (j=0);
            j=anyalnum(string, j+1);
            if j=0 then put 'The end';
            else do;
               c=substr(string, j, 1);
               put j= c=;
            end;
         end;
      end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=L
j=9 c=a
j=10 c=s
j=11 c=t
j=15 c=1
The end
```

Example 2: Scanning a String from Right to Left

The following program uses the ANYALNUM function to search a string from right to left for alphanumeric characters.

```
data rtltest;
   dcl char(15) string c;
   dcl double j;
   method run();
      string='Next = Last + 1;';
      j=999999;
      do until(j=0);
         j=anyalnum(string, 1-j);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
         put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=15 c=1
j=11 c=t
j=10 c=s
j=9 c=a
j=8 c=L
j=4 c=t
j=3 c=x
j=2 c=e
j=1 c=N
The end
```

See Also

Functions:

- "NOTALNUM Function" on page 845

ANYALPHA Function

Searches a character string for an alphabetic character, and returns the first character position at which the character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYALPHA('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYALPHA function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYALPHA function searches a string for the first occurrence of any character that is an uppercase or lowercase letter. If such a character is found, ANYALPHA returns the position in the string of that character. If no such character is found, ANYALPHA returns a value of 0.

If you use only one argument, ANYALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALPHA function searches a character string for an alphabetic character. The NOTALPHA function searches a character string for a non-alphabetic character.

Examples

Example 1: Searching a String for Alphabetic Characters from Left to Right

The following program uses the ANYALPHA function to search a string from left to right for alphabetic characters.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
```

```
      do until(j=0);
         j=anyalpha(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=9 c=n
j=16 c=E
The end
```

Example 2: Identifying Alphabetic Characters By Using the ANYALPHA Function

You can execute the following program to show the alphabetic characters that are identified by the ANYALPHA function.

Note: This program works only for single-byte encodings.

```
data testany;
   dcl nchar(3) byte1 hex1;
   dcl double dec anyalpha1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         anyalpha1=anyalpha(byte1);
         output;
      end;
   end;
enddata;
run;

proc print data=testany;
run;
```

See Also

Functions:

- “NOTALPHA Function” on page 848

ANYCNTRL Function

Searches a character string for a control character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYCNTRL('*expression*' [, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYCNTRL function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYCNTRL function searches a string for the first occurrence of a control character. If such a character is found, ANYCNTRL returns the position in the string of that character. If no such character is found, ANYCNTRL returns a value of 0.

If you use only one argument, ANYCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*,

specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYCNTRL returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYCNTRL function searches a character string for a control character. The NOTCNTRL function searches a character string for a character that is not a control character.

Example

You can execute the following program to show the control characters that are identified by the ANYCNTRL function.

```
data testany);
   dcl nchar(3) byte1 hex1;
   dcl double dec anycntrl1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         anycntrl1=anycntrl(byte1);
         if anycntrl1 then output;
      end;
   end;
enddata;
run;

proc print data=testany;
run;
quit;
```

See Also

Functions:

■ "NOTCNTRL Function" on page 850

ANYDIGIT Function

Searches a character string for a digit, and returns the first character position at which the digit is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYDIGIT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYDIGIT function searches a string for the first occurrence of any character that is a digit. If such a character is found, ANYDIGIT returns the position in the string of that character. If no such character is found, ANYDIGIT returns a value of 0.

If you use only one argument, ANYDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*,

specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYDIGIT function searches a character string for a digit. The NOTDIGIT function searches a character string for any character that is not a digit.

Example

The following program uses the ANYDIGIT function to search for a character that is a digit.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anydigit(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=14 c=1
j=15 c=2
j=17 c=3
The end
```

See Also

Functions:

- “NOTDIGIT Function” on page 852

ANYFIRST Function

Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYFIRST('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_) and uppercase or

lowercase English letters. If such a character is found, ANYFIRST returns the position in the string of that character. If no such character is found, ANYFIRST returns a value of 0.

If you use only one argument, ANYFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Example

The following program uses the ANYFIRST function to search a string for any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anyfirst(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=_
j=9 c=n
j=10 c=_
j=16 c=E
The end
```

See Also

Functions:

- "NOTFIRST Function" on page 855

ANYGRAPH Function

Searches a character string for a graphical character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYGRAPH('*string*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYGRAPH function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYGRAPH function searches a string for the first occurrence of a graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, ANYGRAPH returns the position in the string of that character. If no such character is found, ANYGRAPH returns a value of 0.

If you use only one argument, ANYGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYGRAPH function searches a character string for a graphical character. The NOTGRAPH function searches a character string for a non-graphical character.

Examples

Example 1: Searching a String for Graphical Characters

The following program uses the ANYGRAPH function to search a string for graphical characters.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anygraph(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c=+
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

Example 2: Identifying Control Characters By Using the ANYGRAPH Function

You can execute the following program to show the control characters that are identified by the ANYGRAPH function.

```
data testany (overwrite=yes);
   dcl nchar(3) byte1 hex1;
   dcl double dec anygraph1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
```

```
            anygraph=anygraph(byte1);
            output;
         end;
      end;
   enddata;
   run;

   proc print data=testany;
   run;
   quit;
```

See Also

Functions:

- "NOTGRAPH Function" on page 857

ANYLOWER Function

Searches a character string for a lowercase letter, and returns the first character position at which the letter is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYLOWER('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYLOWER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYLOWER function searches a string for the first occurrence of a lowercase letter. If such a character is found, ANYLOWER returns the position in the string of that character. If no such character is found, ANYLOWER returns a value of 0.

If you use only one argument, ANYLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYLOWER function searches a character string for a lowercase letter. The NOTLOWER function searches a character string for a character that is not a lowercase letter.

Example

The following program uses the ANYLOWER function to search a string for any character that is a lowercase letter.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
```

```
            j=anylower(string, j+1);
            if j=0 then put 'The end';
            else do;
               c=substr(string, j, 1);
               put j= c=;
            end;
         end;
      end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
j=2 c=e
j=3 c=x
j=4 c=t
j=9 c=n
The end
```

See Also

Functions:

- "NOTLOWER Function" on page 860

ANYNAME Function

Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYNAME('*expression*'[,*start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, ANYNAME returns the position in the string of that character. If no such character is found, ANYNAME returns a value of 0.

If you use only one argument, ANYNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

Example

The following program uses the ANYNAME function to search a string for any character that is valid in a SAS variable name under VALIDVARNAME=V7.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anyname(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=_
j=9 c=n
j=10 c=_
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
The end
```

See Also

Functions:

- “NOTNAME Function” on page 862

ANYPRINT Function

Searches a character string for a printable character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYPRINT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYPRINT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYPRINT function searches a string for the first occurrence of a printable character. If such a character is found, ANYPRINT returns the position in the string of that character. If no such character is found, ANYPRINT returns a value of 0.

If you use only one argument, ANYPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYPRINT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.

- The value of *start* = 0.

Comparisons

The ANYPRINT function searches a character string for a printable character. The NOTPRINT function searches a character string for a non-printable character.

Examples

Example 1: Searching a String for a Printable Character

The following program uses the ANYPRINT function to search a string for printable characters.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anyprint(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

Example 2: Identifying Control Characters By Using the ANYPRINT Function

You can execute the following program to show the control characters that are identified by the ANYPRINT function.

```
data testany;
   dcl nchar(3) byte1 hex1;
   dcl double dec anyprint1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         anyprint1=anyprint(byte1);
         output;
      end;
   end;
enddata;
run;

proc print data=testany;
run;
quit;
```

See Also

Functions:

- “NOTPRINT Function” on page 864

ANYPUNCT Function

Searches a character string for a punctuation character, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYPUNCT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYPUNCT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYPUNCT function searches a string for the first occurrence of a punctuation character. If such a character is found, ANYPUNCT returns the position in the string of that character. If no such character is found, ANYPUNCT returns a value of 0.

If you use only one argument, ANYPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYPUNCT function searches a character string for a punctuation character. The NOTPUNCT function searches a character string for a character that is not a punctuation character.

Examples

Example 1: Searching a String for Punctuation Characters

The following program uses the ANYPUNCT function to search a string for punctuation characters.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anypunct(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=8 c=_
j=10 c=_
j=18 c=;
The end
```

Example 2: Identifying Control Characters By Using the ANYPUNCT Function

You can execute the following program to show the control characters that are identified by the ANYPUNCT function.

```
data testany (overwrite=yes);
   dcl nchar(3) byte1 hex1;
   dcl double dec anypunct1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         anypunct1=anypunct(byte1);
         output;
      end;
   end;
enddata;
run;
```

See Also

Functions:

- "NOTPUNCT Function" on page 866

ANYSPACE Function

Searches a character string for a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

ANYSPACE('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYSPACE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYSPACE function searches a string for the first occurrence of any character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, ANYSPACE returns the position in the string of that character. If no such character is found, ANYSPACE returns a value of 0.

If you use only one argument, ANYSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Examples

Example 1: Searching a String for a Whitespace Character

The following program uses the ANYSPACE function to search a string for a character that is a whitespace character.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anyspace(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
```

```
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=7 c=
j=11 c=
j=13 c=
The end
```

Example 2: Identifying Control Characters By Using the ANYSPACE Function

You can execute the following program to show the control characters that are identified by the ANYSPACE function.

```
data testany (overwrite=yes);
   dcl nchar(3) byte1 hex1;
   dcl double dec anyspace1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         anyspace1=anyspace(byte1);
         output;
      end;
   end;
enddata;
run;
```

See Also

Functions:

- “NOTSPACE Function” on page 869

ANYUPPER Function

Searches a character string for an uppercase letter, and returns the first character position at which the letter is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

ANYUPPER('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYUPPER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYUPPER function searches a string for the first occurrence of an uppercase letter. If such a character is found, ANYUPPER returns the position in the string of that character. If no such character is found, ANYUPPER returns a value of 0.

If you use only one argument, ANYUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYUPPER function searches a character string for an uppercase letter. The NOTUPPER function searches a character string for a character that is not an uppercase letter.

Example

The following program uses the ANYUPPER function to search a string for an uppercase letter.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anyupper(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=16 c=E
The end
```

See Also

Functions:

- “NOTUPPER Function” on page 872

ANYXDIGIT Function

Searches a character string for a hexadecimal character that represents a digit, and returns the first character position at which that character is found.

Categories: CAS

Character

Returned data type: DOUBLE

Syntax

ANYXDIGIT('*expression*'[,*start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The ANYXDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The ANYXDIGIT function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, ANYXDIGIT returns the position in the string of that character. If no such character is found, ANYXDIGIT returns a value of 0.

If you use only one argument, ANYXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.

- The value of *start* = 0.

Comparisons

The ANYXDIGIT function searches a character string for a character that is a hexadecimal character. The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character.

Example

The following program uses the ANYXDIGIT function to search a string for a hexadecimal character that represents a digit.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=anyxdigit(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=2 c=e
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
The end
```

See Also

Functions:

- “NOTXDIGIT Function” on page 874

ARCOS Function

Returns the arccosine in radians.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

ARCOS(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range –1 to 1

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ARCOS function returns the arccosine (inverse cosine) of the argument. The value that is returned is specified in radians.

Example

The following program illustrates the ARCOS function:

```
data test (overwrite=yes);
   dcl double a b c;
   method run();
      a=arcos(1);
      b=arcos(0);
      c=arcos(-.5);
      put 'a=' a;
      put 'b=' b;
      put 'c=' c;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log.

```
a= 0
b= 1.57079632679489
c= 2.09439510239319
```

ARCOSH Function

Returns the inverse hyperbolic cosine.

Categories: CAS
Hyperbolic

Returned data type: DOUBLE

Syntax

ARCOSH(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range *expression* >= 1

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ARCOSH function computes the inverse hyperbolic cosine. The ARCOSH function is mathematically defined by the following equation, where *expression* >= 1. In the equation, *expression* is represented by *x*.

$$ARCOSH(x) = log\left(x + \sqrt{x^2 - 1}\right)$$

Example

The following program illustrates the ARCOSH function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=arcosh(5);
      b=arcosh(13);
      put 'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 2.29243166956117
b= 3.25661395480005
```

See Also

Functions:

- “ARSINH Function” on page 327
- “ARTANH Function” on page 329
- “COSH Function” on page 476
- “TANH Function” on page 1122
- “SINH Function” on page 1081

ARSIN Function

Returns the arcsine in radians.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

ARSIN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range –1 to 1

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ARSIN function returns the arcsine (inverse sine) of the argument. The value that is returned is specified in radians.

Example

The following program illustrates the ARSIN function:

```
data test (overwrite=yes);
   dcl double a b c;
   method run();
      a=arsin(0);
      b=arsin(1);
      c=arsin(-0.5);
      put 'a=' a;
      put 'b=' b;
      put 'c=' c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 0
b= 1.57079632679489
c= -0.52359877559829
```

ARSINH Function

Returns the inverse hyperbolic sine.

Categories: CAS
Hyperbolic

Returned data type: DOUBLE

Syntax

ARSINH(*expression*)

Arguments

expression

specifies any valid expression that evaluates to a numeric value.

Range $-\infty < x < \infty$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ARSINH function computes the inverse hyperbolic sine. The ARSINH function is mathematically defined by the following equation, where $-\infty < x < \infty$.

$$ARSINH(x) = log\left(x + \sqrt{x^2 + 1}\right)$$

Replace the infinity symbol with the largest double precision number that is available on your machine. In the equation, *expression* is represented by *x*.

Example

The following program illustrates the ARSINH function:

```
data test (overwrite=yes);
   dcl double a b ;
   method run();
      a=arsinh(5);
      b=arsinh(-5);
      put 'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 2.31243834127275
b= -2.31243834127275
```

See Also

Functions:

- “ARCOSH Function” on page 325
- “ARTANH Function” on page 329
- “COSH Function” on page 476
- “TANH Function” on page 1122
- “SINH Function” on page 1081

ARTANH Function

Returns the inverse hyperbolic tangent.

Categories: CAS
Hyperbolic

Returned data type: DOUBLE

Syntax

ARTANH(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range –1 < *expression* < 1

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

The ARTANH function computes the inverse hyperbolic tangent. The ARTANH function is mathematically defined by the following equation, where –1 < *expression*< 1. In the equation, *expression* is represented by *x*.

$$ARTANH(x) = \frac{1}{2} log\left(\frac{1+x}{1-x}\right)$$

Example

The following program illustrates the ARTANH function:

```
data test (overwrite=yes);
   dcl double a b ;
   method run();
      a=artanh(.5);
      b=artanh(-.5);
      put 'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 0.54930614433405
b= -0.54930614433405
```

See Also

Functions:

- “ARCOSH Function” on page 325
- “ARSINH Function” on page 327
- “COSH Function” on page 476
- “TANH Function” on page 1122
- “SINH Function” on page 1081

ATAN Function

Returns the arctangent in radians.

Categories:	CAS Trigonometric
Alias:	ARTAN
Returned data type:	DOUBLE

Syntax

ATAN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ATAN function returns the 2-quadrant arctangent (inverse tangent) of the argument. The value that is returned is the angle (in radians) whose tangent is x and whose value ranges from $-\pi/2$ to $\pi/2$.

Comparisons

The ATAN function is similar to the ATAN2 function except that ATAN2 calculates the arctangent of the angle from the ratio of two arguments rather than from one argument.

Example

The following program illustrates the ATAN function:

```
data test (overwrite=yes);
   dcl double a b c;
   method run();
      a=atan(0);
      b=atan(1);
      c=atan(-9);
      put 'a=' a;
      put 'b=' b;
      put 'c=' c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 0
b= 0.78539816339744
c= -1.460139105621
```

See Also

Functions:

- “ATAN2 Function” on page 332

ATAN2 Function

Returns the arctangent of the *x* and *y* coordinates of a right triangle, in radians.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

ATAN2 (*expression-1*, *expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value. *expression-1* specifies the x coordinate of the end of the hypotenuse of a right triangle.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

expression-2
specifies any valid expression that evaluates to a numeric value. *expression-2* specifies the y coordinate of the end of the hypotenuse of a right triangle.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

The ATAN2 function returns the arctangent (inverse tangent) of two numeric variables. The result of this function is similar to the result of calculating the arc tangent of *expression-1* / *expression-2*, except that the signs of both arguments are used to determine the quadrant of the result.

Comparisons

The ATAN2 function is similar to the ATAN function except that ATAN calculates the arctangent of the angle from the value of one argument rather than from two arguments.

Example

The following program illustrates the ATAN2 function:

```
data test (overwrite=yes);
   dcl double a b c;
   method run();
      a=atan2(-1, .5);
      b=atan2(6, 8);
      c=atan2(5, -3);
      put 'a=' a;
      put 'b=' b;
      put 'c=' c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= -1.10714871779409
b= 0.64350110879328
c= 2.11121582706548
```

See Also

- "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*

Functions:

- "ATAN Function" on page 330

BAND Function

Returns the bitwise logical AND of two arguments.

Categories:	Bitwise Logical Operations CAS
Returned data type:	DOUBLE

Syntax

BAND(*expression-1, expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

Example

The following program illustrates the BAND function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=band(9, 11);
      b=band(15, 5);
      put 'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 9
b= 5
```

BETA Function

Returns the value of the beta function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

BETA(*a*, *b*)

Arguments

a

is the first shape parameter.

Range *a* >0

Data type DOUBLE

b

is the second shape parameter.

Range *b* >0

Data type DOUBLE

Details

The BETA function is mathematically given by this equation:

$$\beta(a,b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx$$

Note the following:

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

In the previous equation, $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, BETA returns a missing value.

Example

The following program illustrates the BETA function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=beta(5, 3);
      b=beta(15, 45);
      put 'a=' a;
```

```
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 0.0095238095238
b= -4.65396035015752
```

See Also

Functions:

- "LOGBETA Function" on page 796

BETAINV Function

Returns a quantile from the beta distribution.

Categories: CAS
Quantile

Returned data type: DOUBLE

Syntax

BETAINV(*p*, *a*, *b*)

Arguments

p
is a numeric probability.

Range $0 \leq p \leq 1$

Data type DOUBLE

a
is a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

b
is a numeric shape parameter.

Range $b > 0$

Data type DOUBLE

Details

The BETAINV function returns the *p*th quantile from the beta distribution with shape parameters *a* and *b*. The probability that an observation from a beta distribution is less than or equal to the returned quantile is *p*.

Note: BETAINV is the inverse of the PROBBETA function.

Example

The following program illustrates the BETAINV function.

```
data test (overwrite=yes);
   dcl double y z;
   method run();
      y=betainv(0.001, 2, 4);
      put 'y=' y;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y= 0.01010178788373
```

See Also

Functions:

- “PROBBETA Function” on page 943

BLACKCLPRC Function

Calculates call prices for European options on futures, based on the Black model.

Categories: CAS

Financial

Returned data type: DOUBLE

Syntax

BLACKCLPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E
is a nonmissing, positive value that specifies exercise price.

Requirement: Specify *E* and *F* in the same units.

Data type: DOUBLE

t
is a nonmissing value that specifies time to maturity, in years.

Data type: DOUBLE

F
is a nonmissing, positive value that specifies future price.

Requirement: Specify *F* and *E* in the same units.

Data type: DOUBLE

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type: DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Data type: DOUBLE

Details

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$\text{CALL} = \varepsilon^{-rt}(FN(d_1) - EN(d_2))$$

Arguments

F
specifies future price.

N
specifies the cumulative normal density function.

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

t
specifies the time to expiration, in years.

$$d_1 = \frac{\left(\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of t=0, the following equation is true:

$$\text{CALL} = \max((F - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following program illustrates the BLACKCLPRC function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=blackclprc(50, .25, 48, .05, .25);
      b=blackclprc(9, 1/12, 10, .05, .2);
      put 'a=' a;
      put 'b=' b;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 1.55130142723116
b= 1
```

See Also

Functions:

- "BLACKPTPRC Function" on page 340

BLACKPTPRC Function

Calculates put prices for European options on futures, based on the Black model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

BLACKPTPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies time to maturity, in years.

Data type DOUBLE

F

is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

Data type DOUBLE

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Data type DOUBLE

Details

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} + \varepsilon^{-rt}(E - F)$$

Arguments

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

t
specifies the time to expiration, in years.

F
specifies future price.

$$d_1 = \frac{\left(\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{PUT} = \max((E - F), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following program illustrates the BLACKPTPRC function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=blackptprc(298, .25, 350, .06, .25);
      b=blackptprc(145, .5, 170, .05, .2);
      put 'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 1.85980563934967
b= 1.41234979911583
```

See Also

Functions:

- “BLACKCLPRC Function” on page 337

BLKSHCLPRC Function

Calculates call prices for European options on stocks, based on the Black-Scholes model.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

BLKSHCLPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t
is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S
is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$\text{CALL} = SN(d_1) - EN(d_2)\varepsilon^{-rt}$$

Arguments

S
is a nonmissing, positive value that specifies the share price.

N
specifies the cumulative normal density function.

E
is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration, in years.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

σ
specifies the volatility (the square root of the variance).

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{CALL} = \max((S - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following program illustrates the BLKSHCLPRC function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=blkshclprc(50, .25, 48, .05, .25);
      b=blkshclprc(9, 1/12, 10, .05, .2);
      put 'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 1.79894201954462
b= 1
```

See Also

Functions:

- "BLKSHPTPRC Function" on page 345

BLKSHPTPRC Function

Calculates put prices for European options on stocks, based on the Black-Scholes model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

BLKSHPTPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} - S + E\varepsilon^{-rt}$$

Arguments

S
is a nonmissing, positive value that specifies the share price.

E
is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration, in years.

r
specifies the risk-free interest rate. This is an annual rate that is expressed in terms of continuous compounding.

σ
specifies the volatility (the square root of the variance).

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{PUT} = \max((E - S), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following program illustrates the BLKSHPTPRC function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=blkshptprc(230,.5,290,.04,.25);
      b=blkshptprc(350,.3,400,.05,.2);
      put 'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 1.56597442946068
b= 1.64091943067597
```

See Also

Functions:

- “BLKSHCLPRC Function” on page 342

BLSHIFT Function

Returns the bitwise logical left shift of two arguments.

Categories:	Bitwise Logical Operations CAS
Returned data type:	DOUBLE

Syntax

BLSHIFT(*expression-1*, *expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression-2
specifies any valid expression that evaluates to a numeric value.

Range 0 to 31, inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the BLSHIFT function:

```
data test (overwrite=yes);
   dcl double a;
   method run();
      a=blshift(7, 2);
      put a;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
28
```

See Also

Functions:

- "BRSHIFT Function" on page 351

BNOT Function

Returns the bitwise logical NOT of an argument.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BNOT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the BNOT function:

```
data test (overwrite=yes);
   dcl double a;
   method run();
      a=bnot(16);
      put a;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
4294967279
```

BOR Function

Returns the bitwise logical OR of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BOR(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the BOR function:

```
data one (overwrite=yes);
   dcl double x;
   method run();
      x=bor(4, 8);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
12
```

BRSHIFT Function

Returns the bitwise logical right shift of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BRSHIFT(*expression-1*, *expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression-2
specifies any valid expression that evaluates to a numeric value.

Range 0 to 31, inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the BRSHIFT function:

```
data test (overwrite=yes);
   dcl double a;
   method run();
      a=brshift(64, 2);
      put a;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
16
```

See Also

Functions:

- "BLSHIFT Function" on page 347

BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two arguments.

Categories: Bitwise Logical Operations
CAS

Returned data type: DOUBLE

Syntax

BXOR(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the BXOR function:

```
data one (overwrite=yes);
   dcl double x;
   method run();
      x=bxor(128, 64);
      put x;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
192
```

BYTE Function

Returns one character in the ASCII or the EBCDIC collating sequence.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

BYTE(*n*)

Arguments

n
specifies a whole number that represents a specific ASCII or EBCDIC character.

Range 0–255

Data type DOUBLE

Details

For EBCDIC collating sequences, *n* is between 0 and 255. For ASCII collating sequences, the characters that correspond to values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to values between 128 and 255 are available on certain ASCII operating environments, but the information those characters represent varies with the operating environment.

Example

The following program illustrates the BYTE function:

```
data one (overwrite=yes);
```

```
   dcl varchar x;
   method run();
      x=byte(80);
      put x $hex8.;
   end;
enddata;
run;
```

SAS writes the following output to the log. The ASCII equivalent of hexadecimal 50 is “P”. The EBCIDIC equivalent of hexadecimal 50 is “&”.

```
50
```

See Also

Functions:

- “RANK Function” on page 1027

CAT Function

Does not remove leading or trailing blanks, and returns a concatenated character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CAT(*item-1*[, ...*item-n*])

Arguments

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CAT function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CAT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CAT function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CAT is called from the macro processor

If CAT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CAT finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank line in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CAT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators. For more information, see “Length of Returned Variable” on page 355.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Example

The following program shows how the CAT function concatenates strings.

```
data _null_;
   dcl varchar(25) x y z a;
   dcl varchar(70) result;
   method init();
      x='  The 2012 Olym';
      y='pic Arts Festi';
      z='  val included works by D  ';
      a='ale Chihuly.';
      result=cat(x,y,z,a);
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=  The 2012 Olympic Arts Festi  val included works by D  ale Chihuly.
```

See Also

Functions:

- “CATQ Function” on page 356
- “CATS Function” on page 361
- “CATT Function” on page 364
- “LEFT Function” on page 781
- “STRIP Function” on page 1102

CATQ Function

Concatenates character or numeric values by using a delimiter to separate items and by adding quotation marks to strings that contain the delimiter.

Category:	Character
Restriction:	This function is supported only on SAS Viya and the CAS server.
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CATQ(*modifiers* [, *delimiter*], *item-1* [, ..., *item-n*])

Arguments

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the CATQ function. Blanks are ignored. You can use the following characters as modifiers:

1 or '
uses single quotation marks when CATQ adds quotation marks to a string.

2 or "
uses double quotation marks when CATQ adds quotation marks to a string.

a or A
adds quotation marks to all of the item arguments.

b or B
adds quotation marks to item arguments that have leading or trailing blanks that are not removed by the S or T modifiers.

c or C
uses a comma as a delimiter.

d or D
indicates that you have specified the delimiter argument.

h or H
uses a horizontal tab as the delimiter.

m or M
inserts a delimiter for every item argument after the first. If you do not use the M modifier, then CATQ does not insert delimiters for item arguments that have a length of zero after processing that is specified by other modifiers. The M modifier can cause delimiters to appear at the beginning or end of the result and can cause multiple consecutive delimiters to appear in the result.

n or N
converts item arguments to name literals when the value does not conform to the usual syntactic conventions for a SAS name. A name literal is a string in quotation marks that is followed by the letter "n" without any intervening blanks. To use name literals in SAS statements, you must specify the SAS option, VALIDVARNAME=ANY.

q or Q
adds quotation marks to item arguments that already contain quotation marks.

s or S
strips leading and trailing blanks from subsequently processed arguments:

- To strip leading and trailing blanks from the delimiter argument, specify the S modifier *before* the D modifier.

- To strip leading and trailing blanks from the item arguments but *not* from the delimiter argument, specify the S modifier *after* the D modifier.

t or T
trims trailing blanks from subsequently processed arguments:

- To trim trailing blanks from the delimiter argument, specify the T modifier before the D modifier.
- To trim trailing blanks from the item arguments but not from the delimiter argument, specify the T modifier after the D modifier.

x or X
converts item arguments to hexadecimal literals when the value contains nonprintable characters.

Tips If *modifier* is a constant, enclose it in quotation marks. You can also express *modifier* as a variable name or an expression.

The A, B, N, Q, S, T, and X modifiers operate internally to the CATQ function. If an item argument is a variable, then the value of that variable is not changed by CATQ unless the result is assigned to that variable.

delimiter
specifies a character constant, variable, or expression that is used as a delimiter between concatenated strings. If you specify this argument, then you must also specify the D modifier.

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

The CATQ function returns a value to a variable or if CATQ is called inside an expression, CATQ returns a value to a temporary buffer. The value that is returned has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in the DATA step except in WHERE clauses
- up to 65534 characters when CATQ is called from the macro processor

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, then SAS does the following steps:

- changes the result to a blank value in the DATA step and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment

- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1 in the DATA step

If CATQ returns a value in a temporary buffer, then the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATQ finishes processing. In this case, SAS does not write a message about the truncation to the log.

The Basics

If you do not use the C, D, or H modifiers, then CATQ uses a blank as a delimiter.

If you specify neither a quotation mark in *modifier* nor the 1 or 2 modifiers, then CATQ decides independently for each item argument which type of quotation mark to use, if quotation marks are required. The following rules apply:

- CATQ uses single quotation marks for strings that contain an ampersand (&) or percent (%) sign, or that contain more double quotation marks than single quotation marks.
- CATQ uses double quotation marks for all other strings.

The CATQ function initializes the result to a length of zero and then performs the following actions for each item argument:

1. If *item* is not a character string, then CATQ converts *item* to a character string by using the BEST*w*. format and removes leading blanks.
2. If you used the S modifier, then CATQ removes leading blanks from the string.
3. If you used the S or T modifiers, then CATQ removes trailing blanks from the string.
4. CATQ determines whether to add quotation marks based on the following conditions:
   - If you use the X modifier and the string contains control characters, then the string is converted to a hexadecimal literal.
   - If you use the N modifier, then the string is converted to a name literal if either of the following conditions is true:
     - The first character in the string is not an underscore or an English letter.
     - The string contains any character that is not a digit, underscore, or English letter.
   - If you did not use the X or the N modifiers, then CATQ adds quotation marks to the string if any of the following conditions is true:
     - You used the A modifier.
     - You used the B modifier and the string contains leading or trailing blanks that were not removed by the S or T modifiers.
     - You used the Q modifier and the string contains quotation marks.
     - The string contains a substring that equals the delimiter with leading and trailing blanks omitted.

5 For the second and subsequent item arguments, CATQ appends the delimiter to the result if either of the following conditions is true:

- You used the M modifier.
- The string has a length greater than zero after it has been processed by the preceding steps.

6 CATQ appends the string to the result.

Comparisons

The CATX function is similar to the CATQ function except that CATX does not add quotation marks.

Example: Concatenating Strings with the CATQ Function

The following program shows how the CATQ function concatenates strings.

```
data;
      dcl char(110) result1 result2 result3 result4 result5;
      method init();
         result1=CATQ(' ',
                   'noblanks',
                   'one blank',
                   12345,
                   '  lots of blanks    ');
         put result1=;
         result2=CATQ('CS',
                   'Period (.)                ',
                   'Ampersand (&)             ',
                   'Comma (,)                 ',
                   'Double quotation marks (") ',
                   '   Leading Blanks');
         put result2=;
         result3=CATQ('BCQT',
                   'Period (.)                ',
                   'Ampersand (&)             ',
                   'Comma (,)                 ',
                   'Double quotation marks (") ',
                   '   Leading Blanks');
         put result3=;
         result4=CATQ('ADT',
                   '#=#',
                   'Period (.)                ',
                   'Ampersand (&)             ',
                   'Comma (,)                 ',
                   'Double quotation marks (") ',
                   '   Leading Blanks');
         put result4=;
         result5=CATQ('N',
```

```
                    'ABC_123   ',
                    '123    ',
                    'ABC 123');
          put result5=;
        end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
 result1=noblanks "one blank" 12345 " lots of blanks "

 result2=Period (.),Ampersand (&),"Comma (,)",Double quotation marks (),Leading
Blanks

 result3=Period (.),Ampersand (&),"Comma (,)",'Double quotation marks (')'," Leading
Blanks"

 result4="Period (.)"#=#'Ampersand (&)'#=#"Comma (,)"#=#'Double quotation marks
(")'#=#" Leading Blanks"

 result5=ABC_123 "123"n "ABC 123"n
```

See Also

Functions:

- “CAT Function” on page 354
- “CATS Function” on page 361
- “CATT Function” on page 364
- “CATX Function” on page 366

CATS Function

Removes leading and trailing blanks, and returns a concatenated character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CATS(*item*[, ...*item*])

Arguments

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CATS function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CATS function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATS function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CATS is called from the macro processor

If CATS returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATS finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CATS function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators. For more information, see "Length of Returned Variable" on page 362.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Example

The following program shows how the CATS function concatenates strings.

```
data _null_;
   dcl char(25) x y z a;
   dcl char(70) result;
   method init();
      x='  The   Olym';
      y='pic Arts Festi';
      z='  val includes works by D  ';
      a='ale Chihuly.';
      result=cats(x,y,z,a);
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=The   Olympic Arts Festival includes works by Dale Chihuly.
```

See Also

Functions:

- "CAT Function" on page 354
- "CATT Function" on page 364
- "CATX Function" on page 366
- "STRIP Function" on page 1102

CATT Function

Removes trailing blanks, and returns a concatenated character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CATT(*item*[, ...*item*])

Arguments

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CATT function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CATT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATT function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CATT is called from the macro processor

If CATT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATT

finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CATT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators. For more information, see "Length of Returned Variable" on page 364.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Example

The following program shows how the CATT function concatenates strings.

```
data _null_;
   dcl char(25) x y z a;
   dcl char(70) result;
   method init();
      x='  The 2012 Olym';
      y='pic Arts Festi';
      z='  val included works by D  ';
      a='ale Chihuly.';
      result=catt(x,y,z,a);
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=  The 2012 Olympic Arts Festi  val included works by Dale Chihuly.
```

See Also

Functions:

- “CAT Function” on page 354
- “CATQ Function” on page 356
- “CATS Function” on page 361
- “CATX Function” on page 366
- “STRIP Function” on page 1102

CATX Function

Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CATX(*delimiter*, *item-1*[, ...*item-n*])

Arguments

delimiter
specifies a character string that is used as a delimiter between concatenated items.

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, SAS does not write a note to the log. For more information, see “The Basics” on page 366.

Details

The Basics

The CATX function first copies *item-1* to the result, omitting leading and trailing blanks. Then for each subsequent argument *item-i*, i=2, ..., n, if *item-i* contains at least one non-blank character, then CATX appends *delimiter* and *item-i* to the

result, omitting leading and trailing blanks from *item-i*. CATX does not insert the delimiter at the beginning or end of the result. Blank items do not produce delimiters at the beginning or end of the result, nor do blank items produce multiple consecutive delimiters.

Length of Returned Variable

The CATX function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATX function can be up to 32767 characters, except in WHERE clauses.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS truncates the result.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operator. For more information, see "Length of Returned Variable" on page 367.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Note: In the case of variables that have missing values, the concatenation produces different results.

Example

The following program shows how the CATX function concatenates strings. The first data program creates the Values table. The second and third data programs use the Values table as input.

```
/* This data program creates the Values table. */
data values;
   dcl char(4) x1 x2 x3 x4;
   method init();
         /* simple values */
      x1='A'; x2='B'; x3='C'; x4='D';
      output;
      x1='XX'; x2='YY'; x3='ZZ'; x4='WW';
      output;

         /* values with leading, trailing, and embedded white space */
      x1='XX'; x2='Y Y '; x3=' Z Z'; x4=' WW ';
      output;
```

```
         /* CHAR set to missing */
      x1='E'; x2= . ; x3='F'; x4='G';
      output;
      x1='H'; x2= . ; x3= . ; x4='J';
      output;

         /* CHAR set to zero-length strings */
      x1='X'; x2='' ; x3='' ; x4='W';
      output;

         /* CHAR set to the null value */
      x1='X'; x2=null; x3='Z' ; x4=null;
      output;
   end;
run;

   /* This data program creates the Concat1 table. */
data concat1;
   dcl char(1) sp;
   dcl char(4) x1 x2 x3 x4;
   dcl char(20) test1 test2 spacey;
   method run();
     set values;
     SP='^';
     test1 = catx(sp, x1, x2, x3, x4);
     test2 = strip(x1)
             || sp || strip(x2)
             || sp || strip(x3)
             || sp || strip(x4);
     spacey = x1 || sp || x2 || sp || x3 || sp || x4;
   end;
run;

   /* This data program creates the Concat2 table.              */
   /* The example shows what happens when the delimiter contains */
   /* space characters.                                          */
data concat2;
   dcl char(3) sp;
   dcl char(4) x1 x2 x3 x4;
   dcl char(20) test1 test2 spacey;
   method run();
      set values;
      SP = ' ^ ';
      test1 = catx(sp, x1, x2, x3, x4);
      test2 = strip(x1)
              || strip(sp) || strip(x2)
              || strip(sp) || strip(x3)
              || strip(sp) || strip(x4);
      spacey = strip(x1)
              || sp || strip(x2)
              || sp || strip(x3)
              || sp || strip(x4);
   end;
run;
quit;
```

```
proc print data=concat1;
   title 'The Concat1 table';
run;

proc print data=concat2;
   title 'The Concat2 table';
run;
```

Output 7.1 *Table Showing Concatenated Characters*

The Concat1 table

Obs	sp	x1	x2	x3	x4	test1	test2	spacey
1	^	A	B	C	D	A^B^C^D	A^B^C^D	A ^B ^C ^D
2	^	XX	YY	ZZ	WW	XX^YY^ZZ^WW	XX^YY^ZZ^WW	XX ^YY ^ZZ ^WW
3	^	XX	Y Y	Z Z	WW	XX^Y Y^Z Z^WW	XX^Y Y^Z Z^WW	XX ^Y Y ^ Z Z^ WW
4	^	E	.	F	G	E^.^F^G	E^.^F^G	E ^. ^F ^G
5	^	H	.	.	J	H^.^.^J	H^.^.^J	H ^. ^. ^J
6	^	X			W	X^W	X^^^W	X ^ ^ ^W
7	^	X		Z		X^Z	X^^Z^	X ^ ^Z ^

Output 7.2 *Table Showing Concatenated Characters with Spaces*

The Concat2 table

Obs	sp	x1	x2	x3	x4	test1	test2	spacey
1	^	A	B	C	D	A ^ B ^ C ^ D	A^B^C^D	A ^ B ^ C ^ D
2	^	XX	YY	ZZ	WW	XX ^ YY ^ ZZ ^ WW	XX^YY^ZZ^WW	XX ^ YY ^ ZZ ^ WW
3	^	XX	Y Y	Z Z	WW	XX ^ Y Y ^ Z Z ^ WW	XX^Y Y^Z Z^WW	XX ^ Y Y ^ Z Z ^ WW
4	^	E	.	F	G	E ^ . ^ F ^ G	E^.^F^G	E ^ . ^ F ^ G
5	^	H	.	.	J	H ^ . ^ . ^ J	H^.^.^J	H ^ . ^ . ^ J
6	^	X			W	X ^ W	X^^^W	X ^ ^ ^ W
7	^	X		Z		X ^ Z	X^^Z^	X ^ ^ Z ^

See Also

Functions:

- “CAT Function” on page 354

- "CATQ Function" on page 356
- "CATS Function" on page 361
- "CATT Function" on page 364
- "STRIP Function" on page 1102

CDF Function

Computes the left cumulative distribution function from various continuous and discrete probability distributions.

Categories: CAS
Probability

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF(*'distribution'*, *quantile* [, *parameter-1, ..., parameter-k*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'

Distribution	Argument
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of the random variable.

Data type DOUBLE

parameter-1, ... , parameter-k
are optional constants, variables, or expressions that specify the values of *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

Data type DOUBLE

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Function” on page 886
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF BERNOULLI Distribution Function

Returns a value from the Bernoulli cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('BERNOULLI', *x*, *p*)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

p
is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

Details

The CDF function for the Bernoulli distribution returns the probability that an observation from a Bernoulli distribution, with probability of success equal to *p*, is less than or equal to *x*.

$$CDF('BERN', x, p) = \begin{cases} 0 & x < 0 \\ 1 - p & 0 \le x < 1 \\ 1 & x \ge 1 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following program illustrates the CDF Bernoulli distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('bern', 0, .25);
      put  y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.75
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF BERNOULLI Distribution Function” on page 888

- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF BETA Distribution Function

Returns a value from the beta cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('BETA', *x*, *a*, *b* [, *l*, *r*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

b

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $b > 0$

Data type DOUBLE

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range $r > l$

Data type DOUBLE

Details

The CDF function for the beta distribution returns the probability that an observation from a beta distribution, with shape parameters *a* and *b*, is less than or equal to *v*. The following equation describes the CDF function of the beta distribution:

$$CDF('BETA', x, a, b, l, r) = \begin{cases} 0 & x \le l \\ \dfrac{1}{\beta(a,b)} \displaystyle\int_{l}^{x} \dfrac{(v-l)^{a-1}(r-v)^{b-1}}{(r-l)^{a+b-1}} dv & l < x \le r \\ 1 & x > r \end{cases}$$

The following relationship applies to the preceding equation:

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

The following relationship applies to the preceding equation:

$$\Gamma(a) = \int_{0}^{\infty} x^{a-1} e^{-x} dx$$

Example

The following program illustrates the CDF Beta distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('beta', 0.2,3,4);
      put  y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.09888
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF BETA Distribution Function” on page 890
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF BINOMIAL Distribution Function

Returns a value from the binomial cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('BINOMIAL', *m*, *p*, *n*)

Arguments

m

is a whole number, random variable that counts the number of successes.

Range $m = 0, 1, \ldots$

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success parameter.

Range $0 \leq p \leq 1$

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a whole number parameter that counts the number of independent Bernoulli trials.

Range $n = 0, 1, \ldots$

Data type DOUBLE

Details

The CDF function for the binomial distribution returns the probability that an observation from a binomial distribution, with parameters *p* and *n*, is less than or equal to *m*.

$$CDF('BINOM', m, p, n) = \begin{cases} 0 & m < 0 \\ \sum_{j=0}^{m} \binom{n}{j} p^j (1-p)^{n-j} & 0 \le m \le n \\ 1 & m > n \end{cases}$$

Note: There are no *location* or *scale* parameters for the binomial distribution.

Example

The following program illustrates the CDF Binomial distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('BINOM', 4, .5, 10);
      put  y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.376953125
```

See Also

Functions:

- “LOGCDF Function” on page 798

- "LOGPDF Function" on page 801
- "LOGSDF Function" on page 804
- "PDF BINOMIAL Distribution Function" on page 892
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

CDF CAUCHY Distribution Function

Returns a value from the Cauchy cumulative probability distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('CAUCHY', *x* [,*θ*] [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range	λ > 0
Data type	DOUBLE

Details

The CDF function for the Cauchy distribution returns the probability that an observation from a Cauchy distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('CAUCHY', x, \theta, \lambda) = \frac{1}{2} + \frac{1}{\pi} tan^{-1}\left(\frac{x-\theta}{\lambda}\right)$$

Example

The following program illustrates the CDF Cauchy distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('CAUCHY', 2);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.85241638234956
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF CAUCHY Distribution Function” on page 894
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF Chi-Square Distribution Function

Returns a value from the chi-square cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('CHISQUARE', *x*, *df* [, *nc*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df
is a numeric constant, variable, or expression that specifies a degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range $nc \geq 0$

Data type DOUBLE

Details

The CDF function for the chi-square distribution returns the probability that an observation from a chi-square distribution, with *df* degrees of freedom and the noncentrality parameter *nc*, is less than or equal to *x*. This function accepts non-

integer degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central chi-square distribution. In the following equation, let $\nu = df$ and let $\lambda = nc$. The following equation describes the CDF function of the chi-square distribution:

$$CDF('CHISQ', x, \nu, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} P_c(x, \nu + 2j) & x \geq 0 \end{cases}$$

In the equation, $P_c(.,.)$ denotes the probability from the central chi-square distribution:

$$P_c(x, a) = P_g\left(\frac{x}{2}, \frac{a}{2}\right)$$

In the equation, $P_g(y,b)$ is the probability from the gamma distribution given by the equation:

$$P_g(y, b) = \frac{1}{\Gamma(b)} \int_0^y e^{-v} v^{b-1} dv$$

Example

The following program illustrates the CDF Chi-Square distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('CHISQ', 11.264, 11);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.5785813293173
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Chi-Square Distribution Function” on page 896
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065

- “SQUANTILE Function” on page 1090

CDF Conway-Maxwell-Poisson Distribution Function

Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('CONMAXPOI', *y*, *λ*,*v*)

Arguments

y

is a numeric constant, variable, or expression that specifies a nonnegative whole number that represents counts data.

Data type DOUBLE

λ

is similar to the mean, as in the Poisson distribution.

Data type DOUBLE

v

is a numeric constant, variable, or expression that specifies a dispersion parameter.

Data type DOUBLE

Details

The CDF function returns cumulative probability from 0 to y. For more information, see “Conway-Maxwell-Poisson” distribution in the PDF function on page 898.

Example

The following program illustrates the CDF Conway-Maxwell-Poisson distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('CONMAXPOI', 5, 2.3, .4);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.2445411535065
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Conway-Maxwell-Poisson Distribution Function” on page 898
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF Exponential Distribution Function

Returns a value from the exponential cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('EXPONENTIAL', *x* [,*λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the exponential distribution returns the probability that an observation from an exponential distribution, with the scale parameter λ, is less than or equal to *x*.

$$CDF('EXPO', x, \lambda) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\frac{x}{\lambda}} & x \geq 0 \end{cases}$$

Example

The following program illustrates the CDF Exponential distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('expo', 1);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.63212055882855
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF EXPONENTIAL Distribution Function” on page 901
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF F Distribution Function

Returns a value from the F cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “CDF F Distribution Function” on page 385.

Syntax

CDF('F', *x*, *ndf*, *ddf* [, *nc*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

ndf
is a numeric constant, variable, or expression that specifies a numerator degrees of freedom parameter.

Range *ndf* > 0

Data type DOUBLE

ddf
is a numeric constant, variable, or expression that specifies a denominator degrees of freedom parameter.

Range *ddf* > 0

Data type DOUBLE

nc
is a numeric constant, variable, or expression that specifies a noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The CDF function for the *F* distribution returns the probability that an observation from an *F* distribution, with *ndf* numerator degrees of freedom, *ddf* denominator degrees of freedom, and the noncentrality parameter *nc*, is less than or equal to *x*. This function accepts noninteger degrees of freedom for *ndf* and *ddf*. If *nc* is omitted or equal to zero, the value returned is from a central *F* distribution. In the following equation, let $\nu_1 = ndf$, let $\nu_2 = ddf$, and let $\lambda = nc$. The following equation describes the CDF function of the *F* distribution:

$$CDF('F', x, \nu_1, \nu_2, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} P_F(x, \nu_1 + 2j, \nu_2) & x \geq 0 \end{cases}$$

In the equation, $P_f(f,u_1,u_2)$ is the probability from the central *F* distribution with

$$P_F(x, u_1, u_2) = P_B\left(\frac{u_1 x}{u_1 x + u_2}, \frac{u_1}{2}, \frac{u_2}{2}\right)$$

and $P_B(x,a,b)$ is the probability from the standard beta distribution.

Note: There are no *location* or *scale* parameters for the *F* distribution.

Example

The following program illustrates the CDF F distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('F', 3.32, 2, 3);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.82639336022431
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF F Distribution Function” on page 902
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF GAMMA Distribution Function

Returns a value from the gamma cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('GAMMA', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $\lambda > 0$

Data type DOUBLE

Details

The CDF function for the gamma distribution returns the probability that an observation from a gamma distribution, with the shape parameter *a* and the scale parameter λ, is less than or equal to *x*.

$$CDF('GAMMA', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ \dfrac{1}{\lambda^a \Gamma(a)} \displaystyle\int_0^x v^{a-1} e^{-\frac{v}{\lambda}} dv & x \geq 0 \end{cases}$$

Example

The following program illustrates the CDF Gamma distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=cdf('gamma',1,3);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.08030139707139
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801

- "LOGSDF Function" on page 804
- "PDF GAMMA Distribution Function" on page 904
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

CDF Generalized Poisson Distribution Function

Returns a value from the generalized Poisson cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('GENPOISSON', *x*, *θ*, *η*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≤5 and >0

Data type DOUBLE

η

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≥0 and <0.95

Data type DOUBLE

Tip When η =0, the distribution is the Poisson distribution with a mean and variance of θ. When η>0, the mean is $\theta \div (1-\eta)$ and the variance is $\theta \div (1-\eta)^3$.

Details

The probability mass function for the generalized Poisson distribution follows:

$$f(x;\theta,\eta) = \theta(\theta+\eta x)^{x-1} e^{-\theta-\eta x}/x!, \quad x = 0,1,2,..., \qquad \theta > 0,\ 0 \le \eta < 1$$

If η =0, then the generalized Poisson distribution becomes the standard Poisson distribution with the shape parameter θ.

Example

The following program illustrates the CDF Generalized Poisson distribution function:

```
data test (overwrite=yes);
   dcl double y;
   method run();
      y=cdf('GENPOISSON', 9, 1, .7);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.90616296303524
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Generalized Poisson Distribution Function” on page 906
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF GEOMETRIC Distribution Function

Returns a value from the geometric cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('GEOMETRIC', *m*, *p*)

Arguments

m

is a numeric random variable that specifies the number of failures.

Data type DOUBLE

Tip Decimal values are rounded down if they are far away from a whole number.

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

Details

The CDF function for the geometric distribution returns the probability that an observation from a geometric distribution, with the parameter *p*, is less than or equal to *m*.

$$CDF('GEOM', m, p) = \begin{cases} 0 & m < 0 \\ 1 - (1-p)^{(m+1)} & m \ge 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following program illustrates the CDF Geometric distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=cdf('GEOMETRIC',5, .35);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.924581109375
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF GEOMETRIC Distribution Function” on page 908
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF HYPERGEOMETRIC Distribution Function

Returns a value from the hypergeometric cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('HYPER', *x*, *N*, *R*, *n* [, *o*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

N

is a numeric constant, variable, or expression that specifies a population size parameter. This argument must be a whole number.

Range $N = 1, 2, \ldots$

Data type DOUBLE

R

is a numeric constant, variable, or expression that specifies a number of items in the category of interest. This argument must be a whole number.

Range $R = 0, 1, \ldots, N$

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a sample size parameter. This argument must be a whole number.

Range $n = 1, 2, \ldots, N$

Data type DOUBLE

o

is a numeric constant, variable, or expression that specifies an optional numeric odds ratio parameter.

Range $o > 0$

Data type DOUBLE

Details

The CDF function for the hypergeometric distribution returns the probability that an observation from an extended hypergeometric distribution, with population size *N*, number of items *R*, sample size *n*, and odds ratio *o*, is less than or equal to *x*. If *o* is omitted or equal to 1, the value returned is from the usual hypergeometric distribution.

$$CDF\ ('HYPER', x, N, R, n, o) = \begin{cases} 0 & x < max(0, R+n-N) \\ \dfrac{\sum_{i=0}^{x} \binom{R}{i}\binom{N-R}{n-i} o^i}{\sum_{j=max(0,R+n-N)}^{min(R,n)} \binom{R}{j}\binom{N-R}{n-j} o^j} & max(0, R+n-N) \le x \le min(R, n) \\ 1 & x > min(R, n) \end{cases}$$

Example

The following program illustrates the CDF Hypergeometric distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('HYPER',2,200,50,10);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.52367340812173
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Hypergeometric Distribution Function” on page 909
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF LAPLACE Distribution Function

Returns a value from the Laplace cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('LAPLACE', *x* [,*θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Laplace distribution returns the probability that an observation from the Laplace distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('LAPLACE', x, \theta, \lambda) = \begin{cases} \frac{1}{2}e^{\frac{(x-\theta)}{\lambda}} & x < \theta \\ 1 - \frac{1}{2}e^{\left(-\frac{(x-\theta)}{\lambda}\right)} & x \geq \theta \end{cases}$$

Example

The following program illustrates the CDF Laplace distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('LAPLACE',1);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.81606027941427
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF LAPLACE Distribution Function” on page 911
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF LOGISTIC Distribution Function

Returns a value from the logistic cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('LOGISTIC', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Logistic distribution returns the probability that an observation from a Logistic distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('LOGISTIC', x, \theta, \lambda) = \frac{1}{1 + e^{\left(-\frac{x-\theta}{\lambda}\right)}}$$

Example

The following program illustrates the CDF Logistic distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('LOGISTIC',1);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.73105857863
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF LOGISTIC Distribution Function” on page 913
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF LOGNORMAL Distribution Function

Returns a value from the lognormal cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('LOGNORMAL', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a log scale parameter. e(θ) is a scale parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the lognormal distribution returns the probability that an observation from a lognormal distribution, with the log scale parameter θ and the shape parameter λ, is less than or equal to *x*.

$$CDF('LOGN', x, \theta, \lambda) = \begin{cases} 0 & x \le 0 \\ \frac{1}{\lambda\sqrt{2\pi}} \int_{-\infty}^{log(x)} e\left(-\frac{(v-\theta)^2}{2\lambda^2}\right) dv & x > 0 \end{cases}$$

Example

The following program illustrates the CDF Lognormal distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('LOGNORMAL',1);
      put y=;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log.

```
y=0.5
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF LOGNORMAL Distribution Function” on page 915
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF NEGBINOMIAL Distribution Function

Returns a value from the negative binomial cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('NEGBINOMIAL', *m*, *p*, *n*)

Arguments

m
is a numeric constant, variable, or expression that specifies a random variable that counts the number of failures. This argument must be a positive whole number.

Range $m = 0, 1, \ldots$

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

n

is a numeric constant, variable, or expression that counts the number of successes.

Range $n > 0$

Data type DOUBLE

Details

The CDF function for the negative binomial distribution returns the probability that an observation from a negative binomial distribution, with the probability of success *p* and the number of successes *n*, is less than or equal to *m*.

$$CDF('NEGB', m, p, n) = \begin{cases} 0 & m < 0 \\ p^n \sum_{j=0}^{m} \binom{n+j-1}{n-1} (1-p)^j & m \ge 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the negative binomial distribution.

Example

The following program illustrates the CDF Negative Binomial distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('NEGB',1,.5,2);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.5
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF NEGBINOMIAL Distribution Function” on page 916
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF NORMAL Distribution Function

Returns a value from the normal cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('NORMAL', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Normal distribution returns the probability that an observation from the Normal distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*.

$$CDF('NORMAL', x, \theta, \lambda) = \frac{1}{\lambda\sqrt{2\pi}} \int_{-\infty}^{x} e\left(-\frac{(v-\theta)^2}{2\lambda^2}\right) dv$$

Example

The following program illustrates the CDF Normal distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('NORMAL',1.96);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.97500210485177
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF NORMAL Distribution Function” on page 918
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065

■ “SQUANTILE Function” on page 1090

CDF NORMALMIX Distribution Function

Returns a value from the normal mixture cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('NORMALMIX', *x*, *n*, *p*, *m*, *s*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies the number of mixtures.

Range *n* = 1, 2, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies the *n* proportions, $p_1, p_2, \ldots, p_n$, where $\sum_{i=1}^{i=n} p_i = 1$.

Range *p* = 0, 1, ...

Data type DOUBLE

m

is a numeric constant, variable, or expression that specifies the *n* means $m_1, m_2, \ldots, m_n$.

Data type DOUBLE

s

is a numeric constant, variable, or expression that specifies the *n* standard deviations $s_1, s_2, \ldots, s_n$.

Range $s > 0$

Data type DOUBLE

Details

The CDF function for the Normal Mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to *x*.

$$CDF('NORMALMIX', x, n, p, m, s) = \sum_{i=1}^{i=n} p_i \, CDF('NORMAL', x, m_i, s_i)$$

Weights for the Normal Mixture distribution must be nonnegative. If the sum of the weights does not equal 1, then the weights are treated as relative weights and adjusted so that the sum equals 1.

Note: There are no *location* or *scale* parameters for the Normal Mixture distribution.

Example

The following program illustrates the CDF Normal Mixture distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('Normalmix',2.3,3,.33,.33,.34,.5,1.5,2.5,.79,1.6,4.3);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.71813087231314
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801

- "LOGSDF Function" on page 804
- "PDF NORMALMIX Distribution Function" on page 920
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

CDF PARETO Distribution Function

Returns a value from the Pareto cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('PARETO', *x*, *a* [, *k*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

k

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $k > 0$

Data type DOUBLE

Details

The CDF function for the Pareto distribution returns the probability that an observation from a Pareto distribution, with the shape parameter *a* and the scale parameter *k*, is less than or equal to *x*.

$$CDF('PARETO',x,a,k) = \begin{cases} 0 & x < k \\ 1 - \left(\frac{k}{x}\right)^a & x \geq k \end{cases}$$

Example

The following program illustrates the CDF Pareto distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('PARETO',1,1);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF PARETO Distribution Function” on page 922
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF POISSON Distribution Function

Returns a value from the Poisson cumulative probability distribution.

Categories: CAS

Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see "QUANTILE Function" on page 998.

Syntax

CDF('POISSON', *n*, *m*)

Arguments

n

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Range *n* = 0, 1, ...

Data type DOUBLE

m

is a numeric constant, variable, or expression that specifies a mean parameter.

Range *m* > 0

Data type DOUBLE

Details

The CDF function for the Poisson distribution returns the probability that an observation from a Poisson distribution, with mean *m*, is less than or equal to *n*.

$$CDF('POISSON', n, m) = \begin{cases} 0 & n < 0 \\ \sum_{i=0}^{n} e^{-m}\frac{m^i}{i!} & n \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the Poisson distribution.

Example

The following program illustrates the CDF Poisson distribution function:

```
data _null_;
   dcl double x y;
```

```
      method init();
         y=cdf('POISSON',2,1);
         put y=;
      end;
   enddata;
   run;
```

SAS writes the following output to the log.

```
y=0.9196986029286
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF POISSON Distribution Function” on page 924
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF T Distribution Function

Returns a value from the T cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('T', *t*, *df* [, *nc*])

Arguments

t

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df

is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

Data type DOUBLE

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Data type DOUBLE

Details

The CDF function for the *T* distribution returns the probability that an observation from a *T* distribution, with degrees of freedom *df* and the noncentrality parameter *nc*, is less than or equal to *x*. This function accepts noninteger degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central *T* distribution. In the following equation, let $\nu = df$ and let $\delta = nc$.

$$CDF('T', t, \nu, \delta) = \frac{1}{2^{(\nu/2-1)}\Gamma\left(\frac{\nu}{2}\right)} \int_0^{\infty} x^{\nu-1} e^{-\frac{1}{2}x^2} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\frac{tx}{\sqrt{\nu}}} e^{-\frac{1}{2}(u-\delta)^2} du dx$$

Note: There are no *location* or *scale* parameters for the *T* distribution.

Example

The following program illustrates the CDF T distribution function:

```
data _null_;
   dcl double x y;
   method init();
     y=cdf('T',.9,5);
     put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.79531439982768
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF T Distribution Function” on page 926
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF TWEEDIE Distribution Function

Returns a value from the Tweedie cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF (‘TWEEDIE’, *y*, *p* [, *μ*, *ϕ*])

Arguments

y

is a numeric constant, variable, or expression that specifies a random variable.

Range *y* ≥0

Data type DOUBLE

Notes This argument is required.

When $p>1$, y is numeric. When $p=1$, y is a whole number.

p

is a numeric constant, variable, or expression that specifies the power parameter.

Range $p \geq 1$

Data type DOUBLE

Note This argument is required.

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu>0$

Data type DOUBLE

ϕ

is a numeric constant, variable, or expression that specifies the dispersion parameter.

Default 1

Range $\phi>0$

Data type DOUBLE

Details

The CDF function for the Tweedie distribution returns an exponential dispersion model with variance and mean related by the equation $variance = \phi \times \mu^p$.

$$\int_0^y \frac{1}{y} \sum_{j=1}^{\infty} \left(\frac{y^{-j\alpha}(p-1)^{ja}}{\phi^{j(1-\alpha)}(2-p)^j j!\,\Gamma(-j\alpha)} \right) e\left(\frac{1}{\phi}\left(y\frac{\mu^{1-p}-1}{1-p} - \frac{\mu^{2-p}-1}{2-p} \right) \right) dy$$

The following relationship applies to the preceding equation:

$$\alpha = \frac{2-p}{1-p}$$

Note: The accuracy of computed Tweedie probabilities is highly dependent on the location in parameter space. Ten digits of accuracy are usually available except when p is near 2 or phi is near 0. In that case, the accuracy might be as low as six digits.

Example

The following program illustrates the CDF Tweedie distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('TWEEDIE', .8, 5);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.59176291643197
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF TWEEDIE Distribution Function” on page 927
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF UNIFORM Distribution Function

Returns a value from the uniform cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('UNIFORM', *x* [, *l*, *r*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range *r* > *l*

Data type DOUBLE

Details

The CDF function for the uniform distribution returns the probability that an observation from a uniform distribution, with the left location parameter *l* and the right location parameter *r*, is less than or equal to *x*.

$$CDF('UNIFORM', x, l, r) = \begin{cases} 0 & x < l \\ \frac{x-l}{r-l} & l \le x < r \\ 1 & x \ge r \end{cases}$$

Note: The default values for *l* and *r* are 0 and 1, respectively.

Example

The following program illustrates the CDF Uniform distribution function:

```
data _null_;
   dcl double x y;
   method init();
```

```
      y=cdf('UNIFORM',0.25);
      put y=;
    end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.25
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF UNIFORM Distribution Function” on page 930
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF WALD (Inverse Gaussian) Distribution Function

Returns a value from the Wald (also known as the inverse Gaussian) cumulative probability distribution.

Categories:	CAS Probability
Returned data type:	DOUBLE
Note:	The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('WALD', *x*, *λ* [, *μ*])

CDF('IGAUSS', *x*, *λ* [, *μ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $\lambda > 0$

Data type DOUBLE

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu > 0$

Data type DOUBLE

Details

The CDF function for the Wald distribution returns the probability that an observation from a Wald distribution, with the shape parameter λ, is less than or equal to *x*.

$$Fx(x) = \Phi\left\{\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} - 1\right)\right\} + e^{2\lambda/\mu}\Phi\left\{-\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} + 1\right)\right\}$$

In the equation, $\Phi(.)$ is the standard normal cumulative distribution function. When $x \leq 0$, CDF is 0.

Example

The following program illustrates the CDF Wald distribution function:

```
data _null_;
   dcl double x y;
   method init();
     y=cdf('WALD',1,2);
     put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.62769783815525
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Wald (Inverse Gaussian) Distribution Function” on page 932
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

CDF WEIBULL Distribution Function

Returns a value from the Weibull cumulative probability distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 998.

Syntax

CDF('WEIBULL', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The CDF function for the Weibull distribution returns the probability that an observation from a Weibull distribution, with the shape parameter *a* and the scale parameter λ, is less than or equal to *x*.

$$CDF('WEIBULL', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\left(\frac{x}{\lambda}\right)^a} & x \geq 0 \end{cases}$$

Example

The following program illustrates the CDF Weibull distribution function:

```
data _null_;
   dcl double x y;
   method init();
      y=cdf('WEIBULL',1,2);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=0.63212055882855
```

See Also

Functions:

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF WEIBULL Distribution Function” on page 933
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065

- "SQUANTILE Function" on page 1090

CEIL Function

Returns the smallest integer greater than or equal to a numeric value expression.

Categories: CAS
Truncation

Returned data type: DECIMAL, DOUBLE, NUMERIC

Syntax

CEIL(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type: DECIMAL, DOUBLE, NUMERIC

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If the result is a number that does not fit into the range of the argument's data type, the CEIL function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

Unlike the CEILZ function, the CEIL function fuzzes the result. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following program illustrates the CEIL function:

```
data test (overwrite=yes);
   dcl double var1 a b c d e f g h;
   method run();
      var1=2.1;
      a=ceil(var1);
      b=ceil(-2.4);
      c=ceil(1+1.e-11);
      d=ceil(-1+1.e-11);
      e=ceil(1+1.e-13);
      f=ceil(223.456);
      g=ceil(763);
      h=ceil(-223.456);
      put  'a= ' a;
      put  'b= ' b;
      put  'c= ' c;
      put  'd= ' d;
      put  'e= ' e;
      put  'f= ' f;
      put  'g= ' g;
      put  'h= ' h;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=  3
b=  -2
c=  2
d=  0
e=  1
f=  224
g=  763
h=  -223
```

See Also

Functions:

- “CEILZ Function” on page 421
- “FLOOR Function” on page 644
- “FLOORZ Function” on page 646

CEILZ Function

Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

CEILZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Comparisons

Unlike the CEIL function, the CEILZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following program illustrates the CEILZ function:

```
data test (overwrite=yes);
   dcl double a b c d e f g h;
   method run();
      a=ceilz(2.1);
      b=ceilz(-2.4);
      c=ceilz(1+1.e-11);
      d=ceilz(-1+1.e-11);
      e=ceilz(1+1.e-13);
      f=ceilz(223.456);
      g=ceilz(763);
```

```
        h=ceilz(-223.456);
        put  'a= ' a;
        put  'b= ' b;
        put  'c= ' c;
        put  'd= ' d;
        put  'e= ' e;
        put  'f= ' f;
        put  'g= ' g;
        put  'h= ' h;
    end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=  3
b=  -2
c=  2
d=  0
e=  2
f=  224
g=  763
h=  -223
```

See Also

Functions:

- “CEIL Function” on page 419
- “FLOOR Function” on page 644
- “FLOORZ Function” on page 646

CHOOSEC Function

Returns a character value that represents the results of choosing from a list of arguments.

Categories:	CAS Character
Returned data type:	VARCHAR, NVARCHAR

Syntax

CHOOSEC(*index-expression*, *selection-1*[, ...*selection-n*])

Arguments

index-expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

selection
specifies a character constant, variable, or expression. The value of this argument is returned by the CHOOSEC function.

Data type DOUBLE

Details

The CHOOSEC function uses the value of *index-expression* to select from the arguments that follow. For example, if *index-expression* is 3, CHOOSEC returns the value of *selection-3*. If the first argument is negative, the function counts backward from the list of arguments, and returns that value.

Comparisons

The CHOOSEC function is similar to the CHOOSEN function except that CHOOSEC returns a character value while CHOOSEN returns a numeric value.

Example

The following program shows how CHOOSEC chooses from a series of values:

```
data test (overwrite=yes);
   dcl char fruit color planet sport;
   method init();
      Fruit=choosec(1, 'apple', 'orange', 'pear', 'fig');
      Color=choosec(3, 'red', 'blue', 'green', 'yellow');
      Planet=choosec(2, 'Mars', 'Mercury', 'Uranus');
      Sport=choosec(-3, 'soccer', 'baseball', 'gymnastics', 'skiing');
      put Fruit= Color= Planet= Sport=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
fruit=apple    color=green    planet=Mercury  sport=baseball
```

See Also

Functions:

- "CHOOSEN Function" on page 424

CHOOSEN Function

Returns a numeric value that represents the results of choosing from a list of arguments.

Categories: CAS
Numeric

Returned data type: DOUBLE

Syntax

CHOOSEN(*index-expression*, *selection-1*[, ...*selection-n*])

Arguments

index-expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

selection
specifies a numeric constant, variable, or expression. The value of this argument is returned by the CHOOSEN function.

Data type DOUBLE

Details

The CHOOSEN function uses the value of *index-expression* to select from the arguments that follow. For example, if *index-expression* is 3, CHOOSEN returns the value of *selection-3*. If the first argument is negative, the function counts backward from the list of arguments, and returns that value.

Comparisons

The CHOOSEN function is similar to the CHOOSEC function except that CHOOSEC returns a character value while CHOOSEN returns a numeric value.

Example

The following program shows how CHOOSEN chooses from a series of values:

```
data test;
   dcl double itemnumber rank score value;
   method run();
      ItemNumber=choosen(5,100,50,3784,498,679);
      Rank=choosen(-2,1,2,3,4,5);
      Score=choosen(3,193,627,33,290,5);
      Value=choosen(-5,-37,82985,-991,3,1014,-325,3,54,-618);
      put 'ItemNumber= ' ItemNumber;
      put 'Rank= ' Rank;
      put 'Score= ' Score;
      put 'Value= ' Value;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
ItemNumber=  679
Rank=  4
Score=  33
Value=  1014
```

CMISS Function

Counts the number of missing arguments.

Categories: CAS
Descriptive Statistics

Syntax

CMISS(*argument* [, ...*argument*,])

Arguments

argument
specifies a constant, variable, or expression. *argument* can be either a character value or a numeric value.

Details

A character expression is counted as missing if it evaluates to a string that contains all blanks or has a length of zero, except when you use the CMISS function in macro processing. A numeric expression is counted as missing if it evaluates to a numeric missing value: ., ._, .A, ... , .Z.

When you use the CMISS function in macro processing, use a period (.) to represent both a character and a numeric missing value. If you use a blank or null value for a character argument, SAS returns an error. Here are three examples that result in an error:

```
%let macvar=%sysfunc(cmiss(A,%str( )));
%let macvar=%sysfunc(cmiss(A, ));
%let macvar=%sysfunc(cmiss(A,));
```

Here is the example to use to avoid the error condition:

```
%let macvar=%sysfunc(cmiss(A,.));
```

Example

The following program illustrates the CMISS function:

```
data test (overwrite=yes);
   dcl char a b;
   method run();
      a=cmiss(1,0, ' ' , 2,5,' ');
      b=cmiss(1, ' ');
      put  'a=' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a= 2
b= 1
```

See Also

Functions:

- “MISSING Function” on page 826
- “NMISS Function” on page 842

CMP Function

Compares two character strings including trailing blanks.

Categories: CAS
Character

Returned data type: BIGINT

Syntax

CMP (*string-1*, *string-2*)

Arguments

string-1
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

string-2
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

In the CMP function, if *string-1* and *string-2* do not differ, CMP returns a value of zero. If the arguments differ, the sign of the result is negative if *string-1* precedes string-2 in a sort sequence, and positive if *string-1* follows *string-2* in a sort sequence.

The CMP function does not remove trailing blanks.

Comparisons

The CMP function compares two strings but does not remove trailing blanks. The CMPT function compares two strings and does remove trailing blanks.

Example

The following program uses the CMP function to compare two different strings.

```
data test (overwrite=yes);
   dcl double nopad pad greaterthan lessthan;
   method run();
      nopad=cmp('abc', 'def');
      pad=cmp('abc', 'abc ');
      greaterthan=cmp('abc', 'abcdef');
      lessthan=cmp('abcdef', 'abc');
      put nopad= pad= greaterthan= lessthan=;
   end;
enddata;
run;
```

```
nopad=-1 pad=-4 greaterthan=-4 lessthan=4
```

See Also

Functions:

- “CMPT Function” on page 428

CMPT Function

Compares two character strings excluding trailing blanks.

Categories:	CAS Character
Returned data type:	BIGINT

Syntax

CMPT (*string-1*, *string-2*)

Arguments

string-1
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

string-2
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

In the CMPT function, if *string-1* and *string-2* do not differ, CMPT returns a value of zero. If the arguments differ, the sign of the result is negative if *string-1* precedes string-2 in a sort sequence, and positive if *string-1* follows *string-2* in a sort sequence.

The CMPT function removes trailing blanks.

Comparisons

The CMPT function compares two strings and removes trailing blanks. The CMP function compares two strings and does not remove trailing blanks.

Example

The following program uses the CMPT function to compare two different strings.

```
proc ds2;
data test (overwrite=yes);
dcl double nopad pad greaterthan lessthan;
   method run();
      nopad=cmpt('abc', 'def');
      pad=cmpt('abc', 'abc ');
      greaterthan=cmpt('abc', 'abcdef');
      lessthan=cmpt('abcdef', 'abc');
      put nopad= pad= greaterthan= lessthan=;
   end;
enddata;
run;
```

```
nopad=0 pad=0 greaterthan=-4 lessthan=4
```

See Also

Functions:

- “CMP Function” on page 427

CNONCT Function

Returns the noncentrality parameter from a chi-square distribution.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

CNONCT(*x*, *df*, *probability*)

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

probability
is a probability.

Range $0 < probability < 1$

Data type DOUBLE

Details

The CNONCT function returns the nonnegative noncentrality parameter from a noncentral chi-square distribution whose parameters are *x*, *df*, and *nc*. If *probability* is greater than the probability from the central chi-square distribution with the parameters *x* and *df*, a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root *nc* of the equation

$$P_c(x|\mathrm{df}, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_c(x|\mathrm{df}, \mathrm{nc}) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} P_g\left(\frac{x}{2}\middle|\frac{df}{2} + j\right)$$

The following relationship applies to the preceding equation:

$$P_g(x|a)$$

is the probability from the gamma distribution given by

$$P_g(x|a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} \varepsilon^{-t} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

```
proc ds2;
data work /overwrite=yes;
   dcl double x df nc prob ncc;
   method init();
      x=2;
      df=4;
      do nc=1 to 3 by .5;
         prob=probchi(x, df, nc);
         ncc=cnonct(x, df, prob);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=work;
run;
```

Output 7.3 *Computations of the Noncentrality Parameters from the Chi-squared Distribution*

The SAS System

Obs	x	df	nc	prob	ncc
1	2	4	1.0	0.18611	1.0
2	2	4	1.5	0.15592	1.5
3	2	4	2.0	0.13048	2.0
4	2	4	2.5	0.10907	2.5
5	2	4	3.0	0.09109	3.0

See Also

Functions:

- “FNONCT Function” on page 650
- “TNONCT Function” on page 1131

COALESCE Function

Returns the first non-null or nonmissing value from a list of numeric arguments.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

COALESCE(*expression*[, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

COALESCE accepts one or more numeric expressions. The COALESCE function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all expressions are null or missing, then the COALESCE function returns a null or a missing value depending on whether you are in ANSI mode or SAS mode. For more information, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.

Comparisons

The COALESCE function searches numeric expressions, whereas the COALESCEC function searches character expressions.

Example

The following program illustrates the COALESCE function:

```
data one(overwrite=yes);
   dcl double w x y z;
   method run();
      w = coalesce(., .A, 33, 22, 44, .);
      x = coalesce(42, .);
      y = coalesce(.A, .B, .C);
      z = coalesce(., 7, ., ., 42);
      put w=;
      put x=;
      put y=;
      put z=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
w=33
x=42
y=.
z=7
```

See Also

Functions:

- “COALESCEC Function” on page 434

COALESCEC Function

Returns the first non-null or nonmissing value from a list of character arguments.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COALESCEC(*expression*[, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

COALESCEC accepts one or more character expressions. The COALESCEC function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCEC function returns the value of that expression. If all the values of all expressions are null or missing, then the COALESCEC function returns a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see “How DS2 Processes Nulls and SAS Missing Values” in *SAS DS2 Programmer’s Guide*.

Comparisons

The COALESCEC function searches character expressions, whereas the COALESCE function searches numeric expressions.

Example

The following program illustrates the COALESCEC function:

```
data test (overwrite=yes);
   dcl char a b;
   method run();
      a=coalescec('', 'Hello');
      put a;
      b=coalescec('', 'Goodbye', 'Hello');
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
Hello
Goodbye
```

See Also

Functions:

- "COALESCE Function" on page 432

COMB Function

Computes the number of combinations of *n* elements taken *r* at a time.

Categories: CAS
Combinatorial

Returned data type: DOUBLE

Syntax

COMB(*n*, *r*)

Arguments

n

is a nonnegative whole number that represents the total number of elements from which the sample is chosen.

Data type DOUBLE

r

is a nonnegative whole number that represents the number of chosen elements.

Restriction $r \le n$

Data type DOUBLE

Details

The mathematical representation of the COMB function is given by the following equation:

$$COMB(n, r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

In the preceding equation, $n \ge 0$, $r \ge 0$, and $n \ge r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the COMB function.

Example

The following program illustrates the COMB function:

```
data _null_;
   vararray char x[5];
   dcl double k n ncomb;
   method init();
      x:=('ant' 'bee' 'cat' 'dog' 'ewe');
   end;
   method run();
      n=dim(x);
      k=3;
      ncomb=comb(n, k);
      put ncomb=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
ncomb=10
```

See Also

Functions:

- “FACT Function” on page 535
- “PERM Function” on page 935

COMPARE Function

Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

COMPARE(*string-1*, *string-2*[, *modifiers*])

Arguments

string-1
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

string-2
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

modifiers
specifies a character string that can modify the action of the COMPARE function. You can use one or more of the following characters as a valid modifier:

i or I ignores the case in *string-1* and *string-2*.

l or L — removes leading blanks in *string-1* and *string-2* before comparing the values.

n or N — removes quotation marks from any argument that is a name literal and ignores the case of *string-1* and *string-2*. A name literal is a name token that is expressed as a string within quotation marks, followed by the uppercase or lowercase letter *n*. Name literals enable you to use special characters (including blanks) that are not otherwise allowed in table or variable names. For COMPARE to recognize a string as a name literal, the first character must be a quotation mark.

: — (colon) truncates the longer of *string-1* or *string-2* to the length of the shorter string, or to one, whichever is greater. If you do not specify this modifier, the shorter string is padded with blanks to the same length as the longer string.

Data type — CHAR, NCHAR, NVARCHAR, VARCHAR

Tip — COMPARE ignores blanks that are used as modifiers.

Details

The Basics

The order in which the modifiers appear in the COMPARE function is relevant.

- "LN" first removes leading blanks from each string, and then removes quotation marks from name literals.
- "NL" first removes quotation marks from name literals, and then removes leading blanks from each string.

In the COMPARE function, if *string-1* and *string-2* do not differ, COMPARE returns a value of zero. If the arguments differ, then the following apply:

- The sign of the result is negative if *string-1* precedes *string-2* in a sort sequence, and positive if *string-1* follows *string-2* in a sort sequence.
- The magnitude of the result is equal to the position of the leftmost character at which the strings differ.

DBCS Compatibility

The DBCS equivalent function is KCOMPARE. There are minor differences between the COMPARE and KCOMPARE functions. Both functions accept varying numbers of arguments, but usage of the third argument is not compatible. The following example shows the differences in the syntax:

COMPARE(*string-1*, *string-2*[, *modifiers*])

KCOMPARE(*string-1*[, *position*[, *count*]], *string-2*)

For more information, see the "KCOMPARE Function" in *SAS National Language Support (NLS): Reference Guide*.

Examples

Example 1: Understanding the Order of Comparisons When Comparing Two Strings

The following program compares two strings by using the COMPARE function.

```
data test;
   dcl char string1 string2 modifiers having informat $char8. format
$char8.;
   method init();
      string1='12345678'; string2='12345678'; output;
      string1='123'; string2='abc'; output;
      string1='abc'; string2='abx'; output;
      string1='xyz'; string2='abcdef'; output;
      string1='aBc'; string2='abc'; output;
      string1='aBc'; string2='AbC'; modifiers='i'; output;
      string1='   abc  '; string2='abc'; modifiers=' '; output;
      string1='   abc  '; string2='abc'; modifiers='l'; output;
      string1=' abc    '; string2='   abx  '; modifiers=' ';output;
      string1=' abc    '; string2='   abx  '; modifiers='l';  output;
   end;
enddata;
run;

data test_out;
   method run();
      set test;
      result=compare(string1, string2, modifiers);
      put 'String 1= ' string1 'String 2= ' string2 'Modifier= '
modifiers
         'Result= ' result;
   end;
run;

proc print data=test_out noobs;run;quit;
```

Output 7.4 *Results of Comparing Two Strings By Using the COMPARE Function*

The SAS System

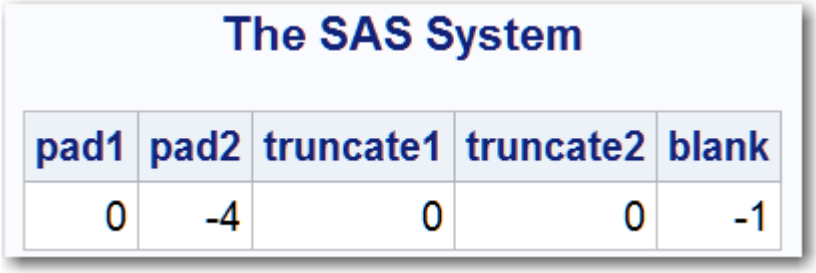
string1	string2	modifiers	result
12345678	12345678		0
123	abc		-1
abc	abx		-3
xyz	abcdef		1
aBc	abc		-2
aBc	AbC	i	0
abc	abc		-1
abc	abc	I	0
abc	abx		2
abc	abx	I	-3

Example 2: Truncating Strings Using the COMPARE Function

The following program uses the : (colon) modifier to truncate strings.

```
data test2;
   dcl double pad1 pad2 truncate1 truncate2 blank;
   method run();
      pad1=compare('abc','abc           ');
      pad2=compare('abc','abcdef        ');
      truncate1=compare('abc','abcdef',':');
      truncate2=compare('abcdef','abc',':');
      blank=compare('','abc',          ':');
      put pad1 pad2 truncate1 truncate2 blank;
   end;
enddata;
run;
proc print data=test2 noobs;run;quit;
quit;
```

Output 7.5 *Results from Using the Colon Modifier to Truncate Strings*



The SAS System

pad1	pad2	truncate1	truncate2	blank
0	-4	0	0	-1

COMPBL Function

Removes multiple blanks from a character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COMPBL(*character-expression*)

Arguments

character-expression
specifies any valid expression that evaluates or can be coerced to a character string and that specifies the character string to compress.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The COMPBL function removes multiple blanks in a character string by translating each occurrence of two or more consecutive blanks into a single blank.

Comparisons

The COMPRESS function removes every occurrence of the specific character from a string. If you specify a blank as the character to remove from the source string,

the COMPRESS function is similar to the COMPBL function. However, the COMPRESS function removes all blanks from the source string. The COMPBL function compresses multiple blanks to a single blank and has no effect on a single blank.

Examples

Example 1: Removing Blanks from a String

The following program illustrates the COMPBL function:

```
data test (overwrite=yes);
   dcl char(20) string1 string2;
   method run();
      string1='January       Status';
      string2=compbl(string1);
      put string2;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
January Status
```

Example 2: Removing Blanks from a String That Is Passed to the Function

The following program illustrates the COMPBL function:

```
data test (overwrite=yes);
   dcl char(18) string1 string2;
   method run();
      string1='125    E. Main St.';
      string2=compbl(string1);
      put string2;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
125 E. Main St.
```

See Also

Functions:

- “COMPRESS Function” on page 460

COMPCOST Function

Sets the costs of operations for later use by the COMPGED function.

Categories: CAS
Character

Note: Support for this function begins with SAS 9.4M7 and SAS Viya 3.5.

Syntax

COMPCOST(*operation-1*, *value-1* [..., *operation-n*, *value-n*]);

Arguments

operation
is a character constant, variable, or expression that specifies an operation that is performed by the COMPGED function.

value
is a numeric constant, variable, or expression that specifies the cost of the operation that is indicated by the preceding argument.

Restriction Must be an integer that ranges from –32767 to 32767, or a missing value

Details

Computing the Cost of Operations

Each argument that specifies an operation must have a value that is a character string. The character string corresponds to one of the terms that are used to denote an operation that the COMPGED function performs. See "Computing the Generalized Edit Distance" on page 449 to view a table of operations that the COMPGED function uses.

The character strings that specify operations can be in uppercase, lowercase, or mixed case. Blanks are ignored. Each character string must end with an equal sign (=). The following table lists valid values for operations and the default cost of the operations.

Operation	Default Cost
APPEND=	very large

Operation	Default Cost
BLANK=	very large
DELETE=	100
DOUBLE=	very large
FDELETE=	equal to DELETE
FINSERT=	equal to INSERT
FREPLACE=	equal to REPLACE
INSERT=	100
MATCH=	0
PUNCTUATION=	very large
REPLACE=	100
SINGLE=	very large
SWAP=	very large
TRUNCATE=	very large

If an operation does not appear in the COMPCOST function, or if the operation appears and is followed by a missing value, that operation is assigned a default cost. A “very large” cost indicates a cost that is sufficiently large that the COMPGED function does not use the corresponding operation.

After your program uses the COMPCOST function, the costs that are specified remain in effect until your program uses the COMPCOST function again, or until the data program that contains the COMPCOST function terminates.

Note: If a FCMP function calls the COMPGED function, then COMPGED is affected by COMPCOST calls within the FCMP function that is processed before the COMPGED processing. If COMPCOST is called outside of the FCMP function, the COMPGED call that is inside the FCMP function is not affected by COMPCOST calls that are outside of the FCMP function.

Abbreviating Character Strings

You can abbreviate character strings. That is, you can use the first one or more letters of a specific operation rather than use the entire term. However, you must use as many letters as necessary to uniquely identify the term. For example, you can specify the INSERT= operation as “in=” and the REPLACE= operation as “r=”. To specify the DELETE= operation or the DOUBLE= operation, you must use the first

two letters because both DELETE= and DOUBLE= begin with “d”. The character string must always end with an equal sign.

Example

This example calls the COMPCOST routine to compute the generalized edit distance for the operations that are specified.

```
data testdata(overwrite=yes);
   dcl char string operation;
   method init();
      string='baboon'; operation='match'; output;
      string='xbaboon'; operation='insert'; output;
      string='babon'; operation='delete'; output;
      string='baXoon'; operation='replace'; output;
   end;
enddata;
run;

data mytest (overwrite=yes);
   dcl double ged;
   method run();
      set testdata;
      if _n_ =1 then compcost('insert=', 10, 'DEL=', 11, 'r=', 12);
      ged=compged(string, 'baboon');
      put 'string=' string 'operation=' operation 'ged=' ged;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
string= baboon   operation= match    ged= 0
string= xbaboon  operation= insert   ged= 10
string= babon    operation= delete   ged= 11
string= baXoon   operation= replace  ged= 12
```

See Also

Functions:

- “COMPGED Function” on page 448
- “COMPLEV Function” on page 455

COMPFUZZ Function

Performs a fuzzy comparison of two numeric values.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

COMPFUZZ(*expression-1*, *expression-2*[, *fuzz*[, *scale*]])

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression-2
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

fuzz
is a nonnegative numeric value that specifies the relative threshold for comparisons. Values that are greater than or equal to one are treated as multiples of the machine precision.

Default 1024

Data type DOUBLE

scale
specifies the scale factor.

Default MAX (ABS (*expression-1*), ABS (*expression-2*))

Data type DOUBLE

Details

The COMPFUZZ function returns the following values if you specify all four arguments:

- -1 if *expression-1* < *expression-2* – threshold
- 0 if ABS(*expression-1* - *expression-2*) ≤ threshold

- 1 if *expression-1* > *expression-2* + threshold

The following relationships exist:

- threshold = *fuzz* * ABS(*scale*) if 0 ≤ *fuzz* < 1
- threshold = *fuzz* * ABS(*scale*) * CONSTANT('MACEPS') if 1 ≤ *fuzz* < 1 / CONSTANT('MACEPS')

COMPFUZZ avoids floating-point underflow or overflow.

Comparisons

The COMPFUZZ function compares two floating-point numbers and returns a value based on the comparison. The ROUND function rounds an argument to a value that is very close to a multiple of a second argument. The result might not be an exact multiple of the second argument.

Example

In floating-point arithmetic, the value of a sum sometimes depends on the order in which the numbers are added. One approximate bound for the floating-point error in the computation of a sum of *n* numbers, x1 through x*n*, is expressed by the following formula:

```
n * machine_precision * sum (abs(x1) + ... + abs(xn))
```

To compare sums of *n* floating-point numbers with the COMPFUZZ function, you can therefore use *n* as the fuzz value and the sum of the absolute values as the scale factor, as shown in the following DATA step:

```
data test (overwrite=yes);
   dcl double x1 x2 x3 x4 sum1 sum2 diff compfuzz1 compfuzz2 scale;
   method run();
      x1 = -1./3.;
      x2 = 22./7.;
      x3 = -1234567891.;
      x4 = 1234567890.;
      /* Add the numbers in two different orders. */
      sum1 = x1 + x2 + x3 + x4;
      sum2 = x4 + x3 + x2 + x1;
      diff = abs(sum1 - sum2);
      put sum1=;
      put sum2=;
      put diff=;
      /* Using only a fuzz value gives the wrong result. The fuzz
value  */
      /* is 8 because there are four numbers in each sum, for a total
of */
      /* eight
numbers.                                                    */
      compfuzz1 = compfuzz(sum1, sum2, 8);
      put 'fuzz only (wrong):        ' compfuzz1=;
```

```
      /* Using a fuzz factor and a scale value gives the correct
result. */
      scale = abs(x1) + abs(x2) + abs(x3) + abs(x4);
      compfuzz2 = compfuzz(sum1, sum2, 8, scale);
      put 'fuzz and scale (correct): ' compfuzz2=;
   end;
enddata;
run;
```

The following lines are written to the SAS log:

```
sum1=1.80952382087707
sum2=1.8095238095238
diff=1.1353265660929E-8
fuzz only (wrong):         compfuzz1=1
fuzz and scale (correct):  compfuzz2=0
```

See Also

Functions:

- “FUZZ Function” on page 652
- “ROUND Function” on page 1035

COMPGED Function

Returns the generalized edit distance between two strings.

Categories: CAS
Character

Returned data type: DOUBLE

Note: Support for this function begins with SAS 9.4M7 and SAS Viya 3.5.

Syntax

COMPGED(*string-1*, *string-2* [, *cutoff*] [, *modifier(s)*])

Arguments

string-1
specifies a character constant, variable, or expression.

string-2
specifies a character constant, variable, or expression.

cutoff
is a numeric constant, variable, or expression.

Note If the actual generalized edit distance is greater than the value of *cutoff*, the value that is returned is equal to the value of *cutoff*.

Tip Using a small value of *cutoff* improves the efficiency of COMPGED if the lengths of the values of *string-1* and *string-2* are long.

modifier
specifies a character string that can modify the action of the COMPGED function. You can use one or more of the following characters as a valid modifier:

i or I	ignores the case in *string-1* and *string-2*.
l or L	removes leading blanks in *string-1* and *string-2* before comparing the values.
n or N	removes quotation marks from any argument that is an n-literal and ignores the case of *string-1* and *string-2*.
: (colon)	truncates the longer of *string-1* or *string-2* to the length of the shorter string, or to 1, whichever is greater.

TIP COMPGED ignores blanks that are used as modifiers.

Details

The Order in Which Modifiers Appear

The order in which the modifiers appear in the COMPGED function is relevant.

- "LN" first removes leading blanks from each string, and then removes quotation marks from n-literals.
- "NL" first removes quotation marks from n-literals, and then removes leading blanks from each string.

Definition of Generalized Edit Distance

Generalized edit distance is a generalization of Levenshtein edit distance, which is a measure of dissimilarity between two strings. The Levenshtein edit distance is the number of deletions, insertions, or replacements of single characters that are required to transform *string-1* into *string-2*.

Computing the Generalized Edit Distance

The COMPGED function returns the generalized edit distance between *string-1* and *string-2*. The generalized edit distance is the minimum-cost sequence of operations for constructing *string-1* from *string-2*.

The algorithm for computing the sum of the costs involves a pointer that points to a character in *string-2* (the input string). An output string is constructed by a sequence of operations that might advance the pointer, add one or more characters to the output string, or both. Initially, the pointer points to the first character in the input string, and the output string is empty.

The operations and their costs are described in the following table.

Operation	Default Cost in Units	Description of Operation
APPEND	50	When the output string is longer than the input string, add any one character to the end of the output string without moving the pointer.
BLANK	10	Perform any of these actions: ■ Add one space character to the end of the output string without moving the pointer. ■ When the character at the pointer is a space character, advance the pointer by one position without changing the output string. ■ When the character at the pointer is a space character, add one space character to the end of the output string and advance the pointer by one position. If the cost for BLANK is set to 0 by the COMPCOST function, the COMPGED function removes all space characters from both strings before beginning the comparison.
DELETE	100	Advance the pointer by one position without changing the output string.
DOUBLE	20	Add the character at the pointer to the end of the output string without moving the pointer.
FDELETE	200	When the output string is empty, advance the pointer by one position without changing the output string.
FINSERT	200	When the pointer is in position one, add any one character to the end of

Operation	Default Cost in Units	Description of Operation
		the output string without moving the pointer.
FREPLACE	200	When the pointer is in position one and the output string is empty, add any one character to the end of the output string and advance the pointer by one position.
INSERT	100	Add any one character to the end of the output string without moving the pointer.
MATCH	0	Copy the character at the pointer from the input string to the end of the output string and advance the pointer by one position.
PUNCTUATION	30	Perform any of these actions: ■ Add one punctuation character to the end of the output string without moving the pointer. ■ When the character at the pointer is a punctuation character, advance the pointer by one position without changing the output string. ■ When the character at the pointer is a punctuation character, add one punctuation character to the end of the output string and advance the pointer by one position. If the cost for PUNCTUATION is set to 0 by the COMPCOST function, the COMPGED function removes all punctuation characters from both strings before beginning the comparison.
REPLACE	100	Add any one character to the end of the output string and advance the pointer by one position.
SINGLE	20	When the character at the pointer is the same as the character that follows in the input string, advance

Operation	Default Cost in Units	Description of Operation
		the pointer by one position without changing the output string.
SWAP	20	Copy the character that follows the pointer from the input string to the output string. Then copy the character at the pointer from the input string to the output string. Advance the pointer two positions.
TRUNCATE	10	When the output string is shorter than the input string, advance the pointer by one position without changing the output string.

To set the cost of the string operations, you can use the COMPCOST function or use default costs. If you use the default costs, the values that are returned by COMPGED are approximately 100 times greater than the values that are returned by COMPLEV.

Examples of Errors

The rationale for determining the generalized edit distance is based on the number and types of typographical errors that can occur. COMPGED assigns a cost to each error and determines the minimum sum that could be incurred. Some types of errors can be more serious than others. For example, inserting an extra letter at the beginning of a string might be more serious than omitting a letter from the end of a string. For another example, if you enter a word or phrase that exists in *string-2* and introduce a typographical error, you might produce *string-1* instead of *string-2*.

Making the Generalized Edit Distance Symmetric

Generalized edit distance is not necessarily symmetric. That is, the value that is returned by `COMPGED(string1, string2)` is not always equal to the value that is returned by `COMPGED(string2, string1)`. To make the generalized edit distance symmetric, use the COMPCOST function to assign equal costs to the operations within each of these pairs:

- INSERT, DELETE
- FINSERT, FDELETE
- APPEND, TRUNCATE
- DOUBLE, SINGLE

Comparisons

You can compute the Levenshtein edit distance by using the COMPLEV function. You can compute the generalized edit distance by using the COMPCOST and COMPGED functions. Computing generalized edit distance requires considerably more computer time than computing the Levenshtein edit distance. But generalized edit distance usually provides a more useful measure than the Levenshtein edit distance for applications such as fuzzy file merging and text mining.

Note: When called from a DS2 program, the following FCMP functions do not honor seed values that are set in the DS2 program.

COMPCOST

COMPGED

RAND

STREAMINIT

In a DATA step, all random numbers from RAND are drawn from the same stream by default, but you can create multiple independent streams by using the CALL STREAM routine.

In PROC DS2, as a result of an unexpected feature, random numbers from RAND that are called inside an FCMP package come from a separate stream. To get reproducible random numbers, use the CALL STREAMINIT routine inside each FCMP function that calls RAND. To get independent streams, also use the CALL STREAM routine inside each FCMP function that calls RAND. If you do so, the results from DS2 will be consistent with a DATA step and with future releases. For an example, see "Considerations and Limitations When Using the FCMP Package" in *SAS DS2 Programmer's Guide*.

Example

This example uses the default costs to calculate the generalized edit distance.

```
/**** input data ****/
data testdata(overwrite=yes);
   dcl char string1 string2;
   dcl char(30) operation;
   method init();
      string1='baboon';   string2='baboon';   operation='match';
output;
      string1='baXboon';  string2='baboon';   operation='insert';
output;
      string1='baoon';    string2='baboon';   operation='delete';
output;
      string1='baXoon';   string2='baboon';   operation='replace';
output;
      string1='baboonX';  string2='baboon';   operation='append';
output;
      string1='baboo';    string2='baboon';   operation='truncate';
output;
```

```
      string1='babboon';  string2='baboon';   operation='double';
output;
      string1='babon';    string2='baboon';   operation='single';
output;
      string1='baobon';   string2='baboon';   operation='swap'; output;
      string1='bab oon';  string2='baboon';   operation='blank';
output;
      string1='bab,oon';  string2='baboon';   operation='punctuation';
         output;
      string1='bXaoon';   string2='baboon';   operation='insert
+delete';
         output;
      string1='bXaYoon';  string2='baboon';   operation='insert
+replace';
         output;
      string1='bXoon';    string2='baboon';   operation='delete
+replace';
         output;
      string1='Xbaboon';  string2='baboon';   operation='finsert';
         output;
      string1='aboon';    string2='baboon';
         operation='trick question: swap+delete'; output;
      string1='Xaboon';   string2='baboon';   operation='freplace';
output;
      string1='axoon';    string2='baboon';
         operation='fdelete+replace'; output;
      string1='axoo';     string2='baboon';
         operation='fdelete+replace+truncate'; output;
      string1='axon';     string2='baboon';
         operation='fdelete+replace+single'; output;
      string1='baby';     string2='baboon';
         operation='replace+truncate*2'; output;
      string1='balloon';  string2='baboon';
         operation='replace+insert'; output;
   end;
enddata;
run;


/**** comparison code *****/
data mytest (overwrite=yes) noobs;
   dcl double ged;
   method run();
      set testdata;
      ged=compged(string1, string2);
      put 'string1=' string1 'string2=' string2 'operation='
          operation 'ged=' ged;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
string1= baboon   string2= baboon   operation= match                          ged= 0
string1= baXboon  string2= baboon   operation= insert                         ged= 100
string1= baoon    string2= baboon   operation= delete                         ged= 100
string1= baXoon   string2= baboon   operation= replace                        ged= 100
string1= baboonX  string2= baboon   operation= append                         ged= 50
string1= baboo    string2= baboon   operation= truncate                       ged= 10
string1= babboon  string2= baboon   operation= double                         ged= 20
string1= babon    string2= baboon   operation= single                         ged= 20
string1= baobon   string2= baboon   operation= swap                           ged= 20
string1= bab oon  string2= baboon   operation= blank                          ged= 10
string1= bab,oon  string2= baboon   operation= punctuation                    ged= 30
string1= bXaoon   string2= baboon   operation= insert+delete                  ged= 200
string1= bXaYoon  string2= baboon   operation= insert+replace                 ged= 200
string1= bXoon    string2= baboon   operation= delete+replace                 ged= 200
string1= Xbaboon  string2= baboon   operation= finsert                        ged= 200
string1= aboon    string2= baboon   operation= trick question: swap+delete    ged= 120
string1= Xaboon   string2= baboon   operation= freplace                       ged= 200
string1= axoon    string2= baboon   operation= fdelete+replace                ged= 300
string1= axoo     string2= baboon   operation= fdelete+replace+truncate       ged= 310
string1= axon     string2= baboon   operation= fdelete+replace+single         ged= 320
string1= baby     string2= baboon   operation= replace+truncate*2             ged= 120
string1= balloon  string2= baboon   operation= replace+insert                 ged= 200
```

See Also

Functions:

- “COMPCOST Function” on page 443
- “COMPLEV Function” on page 455

COMPLEV Function

Returns the Levenshtein edit distance between two strings.

Categories: CAS
Character

Returned data type: DOUBLE

Note: Support for this function begins with SAS 9.4M7 and SAS Viya 3.5.

Syntax

COMPLEV(*string-1*, *string-2* [, *cutoff*] [, *modifier(s)*])

Arguments

string-1
specifies a character constant, variable, or expression.

string-2
specifies a character constant, variable, or expression.

cutoff
specifies a numeric constant, variable, or expression.

Note If the actual Levenshtein edit distance is greater than the value of *cutoff*, the value that is returned is equal to the value of *cutoff*.

Tip Using a small value of *cutoff* improves the efficiency of COMPLEV if the length of the values of *string-1* and *string-2* are long.

modifier
specifies a character string that can modify the action of the COMPLEV function. You can use one or more of the following characters as a valid modifier:

i or I	ignores the case in *string-1* and *string-2*.
l or L	removes leading blanks in *string-1* and *string-2* before comparing the values.
n or N	removes quotation marks from any argument that is an n-literal and ignores the case of *string-1* and *string-2*.
: (colon)	truncates the longer of *string-1* or *string-2* to the length of the shorter string, or to 1, whichever is greater.

TIP COMPLEV ignores blanks that are used as modifiers.

Details

The order in which the modifiers appear in the COMPLEV function is relevant.

- "LN" first removes leading blanks from each string, and then removes quotation marks from n-literals.
- "NL" first removes quotation marks from n-literals, and then removes leading blanks from each string.

The COMPLEV function ignores trailing blanks.

COMPLEV returns the Levenshtein edit distance between *string-1* and *string-2*. The Levenshtein edit distance is the number of insertions, deletions, or replacements of single characters that are required to convert one string to the other. Levenshtein edit distance is symmetric. That is, `COMPLEV(string-1,string-2)` is the same as `COMPLEV(string-2,string-1)`.

Comparisons

The Levenshtein edit distance that is computed by COMPLEV is a special case of the generalized edit distance that is computed by COMPGED.

COMPLEV executes much more quickly than COMPGED.

Example

This example compares two strings by computing the Levenshtein edit distance.

```
/**** input data ****/
proc ds2;
data testdata (overwrite=yes);
   dcl char string1 string2 modifier;
   method init();
      string1='12345678'; string2='12345678'; output;
      string1='abc'; string2='abxc'; output;
      string1='ac'; string2='abc'; output;
      string1='aXc'; string2='abc'; output;
      string1='aXbZc'; string2='abc'; output;
      string1='aXYZc'; string2='abc'; output;
      string1='WaXbYcZ'; string2='abc'; output;
      string1='XYZ'; string2='abcdef'; output;
      string1='aBc'; string2='abc'; output;
      string1='aBc'; string2='AbC'; modifier='i'; output;
      string1='  abc'); string2='abc'; modifier=''; output;
      string1='  abc'); string2='abc';  modifier='l'; output;
      string1='AxC'; string2=n'abc'; modifier=''; output;
      string1='AxC'; string2=n'abc'; modifier='n'; output;
   end;
enddata;
run;


/**** comparison code *****/
proc ds2;
data test (overwrite=yes);
   dcl double result;
   method run();
      set testdata;
      result=complev(string1, string2, modifier);
      put 'string1=' string1 'string2=' string2 'modifier=' modifier
         'result=' result;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
string1= 12345678 string2= 12345678 modifier=          result= 0
string1= abc      string2= abxc     modifier=          result= 1
string1= ac       string2= abc      modifier=          result= 1
string1= aXc      string2= abc      modifier=          result= 1
string1= aXbZc    string2= abc      modifier=          result= 2
string1= aXYZc    string2= abc      modifier=          result= 3
string1= WaXbYcZ  string2= abc      modifier=          result= 4
string1= XYZ      string2= abcdef   modifier=          result= 6
string1= aBc      string2= abc      modifier=          result= 1
string1= aBc      string2= AbC      modifier= i        result= 0
string1=   abc    string2= abc      modifier=          result= 2
string1=   abc    string2= abc      modifier= l        result= 0
string1= AxC      string2= abc      modifier=          result= 3
string1= AxC      string2= abc      modifier= n        result= 1
```

See Also

Functions:

- “COMPCOST Function” on page 443
- “COMPGED Function” on page 448

COMPOUND Function

Returns compound interest parameters.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

COMPOUND(*a*, *f*, *r*, *n*)

Arguments

a

specifies the initial amount.

Range $a \geq 0$

Data type DOUBLE

f

specifies the future amount (at the end of *n* periods).

Range $f \geq 0$

Data type DOUBLE

r

specifies the periodic interest rate expressed as a fraction.

Range $r \geq 0$

Data type DOUBLE

n

specifies the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

Details

The COMPOUND function returns the missing argument in the list of four arguments from a compound interest calculation. The arguments are related by the following equation:

$$f = a(1 + r)^n$$

One missing argument must be provided. A compound interest parameter is then calculated from the remaining three values. No adjustment is made to convert the results to round numbers.

If n=0, then

$$f = a$$

and

$$(1 + r)^n$$

is equal to 1.

Note: If you choose *r* as your missing value, then COMPOUND returns an error.

Example

The accumulated value of an investment of $2000 at a nominal annual interest rate of 9%, compounded monthly after 30 months, can be calculated using this program. The second argument has been set to missing, indicating that the future amount is to be calculated. The 9% nominal annual rate has been converted to a monthly rate

of 0.09/12. The rate argument is the fractional (not the percentage) interest rate per compounding period.

```
data test (overwrite=yes);
   dcl double future;
   method run();
      future=compound(2000, ., 0.09/12, 30);
      put future;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2502.54352764668
```

COMPRESS Function

Returns a character string with specified characters removed from the original string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COMPRESS(*character-expression*[, *character-list-expression*] [, *modifier(s)*])

Arguments

character-expression
specifies any valid expression that evaluates to a character expression and from which specified characters will be removed.

Requirement: Enclose a literal string of characters in single quotation marks.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

character-list-expression
specifies a variable or any valid expression that initializes a list of characters.

By default, the characters in this list are removed from *character-expression*.

Requirement: Enclose a literal string of characters in single quotation marks.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COMPRESS function. Blanks are ignored. The following characters can be used as modifiers:

a or A	adds alphabetic characters to the list of characters.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds the underscore character and English letters to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters to be kept or removed.
k or K	keeps the characters in the list instead of removing them.
l or L	adds lowercase letters to the list of characters.
n or N	adds digits, the underscore character, and English letters to the list of characters.
o or O	processes the second and third arguments once rather than every time the COMPRESS function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make COMPRESS run much faster when you call it in a loop where the second and third arguments do not change.
p or P	adds punctuation marks to the list of characters.
s or S	adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, form feed, and NBSP ('A0'x, or 160 decimal ASCII) to the list of characters.
t or T	trims trailing blanks from the first and second arguments.
u or U	adds uppercase letters to the list of characters.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

Tip If the modifier is a constant, enclose it in single quotation marks. Specify multiple constants in a single set of quotation marks. Modifier can also be expressed as a variable or an expression.

Details

The COMPRESS function compiles a list of characters to keep or remove, comprising the characters in the second argument plus any types of characters that are specified by the modifiers.

Based on the number of arguments, the COMPRESS function works as follows:

Number of Arguments	Results
Only the first argument, *source*	All blanks have been removed. If the argument is completely blank, then the result is a string with a length of zero. If you assign the result to a character variable with a fixed length, then the value of that variable is padded with blanks to fill its defined length.
Two arguments, *source* and *chars*	All characters that appear in the second argument are removed from the result.
Three arguments, *source* , *chars*, and *modifier(s)*	The K modifier (specified in the third argument) determines whether the characters in the second argument are kept or removed from the result.

Here is an example that illustrates the effect of the third argument. The D modifier specifies digits. Both of the following function calls remove digits from the result:

```
compress(source, "1234567890");
compress(source, '', "d");
```

To remove digits and plus or minus signs, you could use the following function call:

```
COMPRESS(source, "1234567890+-");
compress(source, "+-", "d");
```

The COMPRESS function allows null values and SAS missing values in the second argument. A null or missing value is treated as a string that has a length of 0. A string that has length 0 is an empty string.

```
compress(source, null);
compress(source, '');
```

Examples

Example 1: Compressing Blanks

This program illustrates how to remove blanks from a character string.

```
data test(overwrite=yes);
   dcl char a b;
```

```
   method run();
      a='AB C D ';
      b=compress(a);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
ABCD
```

Example 2: Compressing Uppercase Letters

This program illustrates how to remove uppercase letters from a character string.

```
data test(overwrite=yes);
   dcl char a b;
   method run();
      a='AB C D';
      b=compress(a, 'A');
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
B C D
```

Example 3: Compressing a String and Returning a Length of 0

This program illustrates how to remove uppercase letters from a character string.

```
data test(overwrite=yes);
   dcl char(10) x;
   dcl double l;
   method run();
      x=' ';
      l=length(compress(x));
      put l=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
l=0
```

Example 4: Compressing Vowels

This example illustrates how to remove vowels from a string:

```
data test(overwrite=yes);
   dcl char(32) x;
   dcl char(32) y;
```

```
   method run();
      x='123-4567-8901 e 234-5678-9012 i';
      y=compress(x,'aeiou');
      put y;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
123-4567-8901  234-5678-9012
```

Example 5: Compressing Lowercase Letters by Using a Modifier

This example compresses the uppercase letters using the list of characters to be removed and any lowercase letters that are specified by the modifier “l”.

```
data test(overwrite=yes);
   dcl char(32) x y;
   method run();
      x='123-4567-8901 B 234-5678-9012 c';
      y=compress(x, 'ABCD', 'l');
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=123-4567-8901  234-5678-9012
```

Example 6: Compressing Space Characters by Using a Modifier

This example illustrates removing spaces from the given string by using the modifier “s”.

```
data test(overwrite=yes);
   dcl char(10) x y;
   method run();
      x='1 2 3 4 5';
      y=compress(x, '', 's');
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=12345
```

Example 7: Keeping Characters in the List by Using a Modifier

This example illustrates keeping characters in the specified string by using the modifier “k”.

```
data test(overwrite=yes);
```

```
   dcl char(26) x y;
   method run();
      x='Math A English B Physics A';
      y=compress(x, 'ABCD', 'k');
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
y=ABA
```

See Also

Functions:

- “COMPBL Function” on page 441
- “LEFT Function” on page 781
- “TRIM Function” on page 1151

CONSTANT Function

Computes machine and mathematical constants.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

CONSTANT(*constant*[, *parameter*])

Arguments

constant
is a character constant, variable, or expression that identifies the constant to be returned. Valid constants are as follows:

Constant	Description
'E'	The natural base

Constant	Description
'EULER'	Euler constant
'PI'	Pi
'EXACTINT' [,nbytes]	Exact integer
'BIG'	The largest double-precision number
'LOGBIG' [,base]	The log with respect to *base* of BIG
'SQRTBIG'	The square root of BIG
'SMALL'	The smallest double-precision number
'LOGSMALL' [,base]	The log with respect to *base* of SMALL
'SQRTSMALL'	The square root of SMALL
'MACEPS'	Machine precision constant
'LOGMACEPS' [,base]	The log with respect to *base* of MACEPS
'SQRTMACEPS'	The square root of MACEPS

parameter
is a numeric parameter that can be used as an optional argument with some of the constants specified in *constant*. When used, *parameter* alters the functionality of the CONSTANT function.

Details

Overview

CAUTION
In some operating environments, the run-time library might have limitations that prevent the use of the full range of floating-point numbers that the hardware provides. In such cases, the CONSTANT function attempts to return values that are compatible with the limitations of the run-time library. For example, if the run-time library cannot compute `EXP(LOG(CONSTANT('BIG')))`, then `CONSTANT('LOGBIG')` will not return the same value as `LOG(CONSTANT('BIG'))`, but returns a value such that `EXP(CONSTANT('LOGBIG'))` can be computed.

The Natural Base

CONSTANT('E')

The natural base is described by the following equation:

$$\lim_{x \to 0} (1+x)^{\frac{1}{x}} \approx 2.718281828459045$$

Euler Constant

CONSTANT('EULER')

Euler's constant is described by the following equation:

$$\lim_{n \to \infty} \left\{ \sum_{j=1}^{j=n} \frac{1}{j} - \log(n) \right\} \approx 0.577215664901532860$$

Pi

CONSTANT('PI')

Pi is the ratio between the circumference and the diameter of a circle. Many expressions exist for computing this constant. One such expression for the series is described by the following equation:

$$4 \sum_{j=0}^{j=\infty} \frac{(-1)^j}{2j+1} \approx 3.14159265358979323846$$

Exact Integer

CONSTANT('EXACTINT'[, *nbytes*])

Arguments

nbytes
is a numeric value that is the number of bytes.

Default 8

Range $2 \le nbytes \le 8$

The exact integer is the largest integer *k* such that all integers less than or equal to *k* in absolute value have an exact representation in a SAS numeric variable of length *nbytes*. This information can be useful to know before you trim a SAS numeric variable from the default 8 bytes of storage to a lower number of bytes to save storage.

The Largest Double-Precision Number

CONSTANT('BIG')

This case returns the largest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of BIG

CONSTANT('LOGBIG'[, *base*])

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given *base* raised to a power less than or equal to CONSTANT('LOGBIG', *base*) by using the power operation (**) without causing any overflows.

It is safe to exponentiate any floating-point number less than or equal to CONSTANT('LOGBIG') by using the exponential function, EXP, without causing any overflows.

The Square Root of BIG

CONSTANT('SQRTBIG')

This case returns the square root of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to square any floating-point number less than or equal to CONSTANT('SQRTBIG') without causing any overflows.

The Smallest Double-Precision Number

CONSTANT('SMALL')

This case returns the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of SMALL

CONSTANT('LOGSMALL'[, *base*])

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given *base* raised to a power greater than or equal to CONSTANT('LOGSMALL', *base*) by using the power operation (**) without causing any underflows or 0.

It is safe to exponentiate any floating-point number greater than or equal to CONSTANT('LOGSMALL') by using the exponential function, EXP, without causing any underflows or 0.

The Square Root of SMALL

CONSTANT('SQRTSMALL')

This case returns the square root of the smallest double-precision floating-point number (8-bytes) that is representable on the computer.

It is safe to square any floating-point number greater than or equal to CONSTANT('SQRTBIG') without causing any underflows or 0.

Machine Precision

CONSTANT('MACEPS')

This case returns the smallest double-precision floating-point number (8-bytes) $\varepsilon = 2^{-j}$ for some integer *j*, such that $1 + \varepsilon > 1$.

This constant is important in finite precision computations.

The Logarithm of MACEPS

CONSTANT('LOGMACEPS'[, *base*])

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of CONSTANT('MACEPS').

The Square Root of MACEPS

CONSTANT('SQRTMACEPS')

This case returns the square root of CONSTANT('MACEPS').

Example

The following program uses the CONSTANT function to return values for various constants.

```
data test;
/* dcl double a b c d; */
   method run();
      a=constant('E');
      b=constant('EULER');
      c=constant('PI');
      d=constant('EXACTINT');
      e=constant('BIG');
      f=constant('LOGBIG');
      g=constant('SQRTBIG');
      h=constant('SMALL');
      i=constant('LOGSMALL');
      j=constant('SQRTSMALL');
      k=constant('MACEPS');
      l=constant('LOGMACEPS');
      m=constant('SQRTMACEPS');
      put  'a= ' a;
      put 'b= ' b;
      put 'c= ' c;
      put 'd= ' d;
      put 'e= ' e;
      put 'f= ' f;
      put 'g= ' g;
      put 'h= ' h;
      put 'i= ' i;
      put 'j= ' j;
      put 'k= ' k;
      put 'l= ' l;
      put 'm= ' m;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=  2.71828182845904
b=  0.57721566490153
c=  3.14159265358979
d=  9007199254740992
e=  1.7976931348623E308
f=  709.782712893383
g=  1.3407807929942E154
h=  2.2250738585072E-308
i=  -708.396418532264
j=  1.49166814624E-154
k=  2.2204460492503E-16
l=  -36.0436533891171
m=  1.4901161193847E-8
```

CONVX Function

Returns the convexity for an enumerated cash flow.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

CONVX(*y*, *f*, *c(1)*, ..., *c(k)*)

Arguments

y
specifies the effective per-period yield-to-maturity.

Range $0 < y < 1$

Data type DOUBLE

Tip If you express *y* as a fraction, the dividend must be written as a decimal value. In DS2, integer division results in a value of zero. Zero is converted to a DOUBLE and is passed as the first argument to the CONVX function. The CONVX function returns missing when a zero is passed as the first parameter.

f
specifies the frequency of cash flows per period.

Range $f > 0$

Data type DOUBLE

c(1), ... , c(k)
specifies a list of cash flows.

Data type DOUBLE

Details

The CONVX function returns the value from the following equation.

$$C = \sum_{k=1}^{K} \frac{k(k+f)\frac{c(k)}{(1+y)^{\frac{k}{f}}}}{P\left((1+y)^2\right)f^2}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example: Using the CONVX Function

```
data test;
   dcl double c;
   method run();
      c=convx(1./20,.1,.33,.44,.55,.49,.50,.22,.4,.8,.01,.36,.2,.4);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
976.941120481246
```

See Also

Functions:

- “CONVXP Function” on page 472

CONVXP Function

Returns the convexity for a periodic cash flow stream, such as a bond.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

CONVXP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal per-period coupon rate, expressed as a decimal.

Range $0 \leq c < 1$

Data type DOUBLE

n

specifies the number of coupons per period.

Range $n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons.

Range $K > 0$

Data type DOUBLE

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_0 \leq \frac{1}{n}$

Data type DOUBLE

y

specifies the nominal per-period yield-to-maturity.

Range $y > 0$

Data type DOUBLE

Details

The CONVXP function returns the value from the following equation.

$$C = \frac{1}{n^2}\left(\frac{\sum_{k=1}^{K} t_k(t_k+1)\frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)^2}\right)$$

The following relationships apply to the preceding equation:

$$t_k = nk_0 + k - 1$$

$$c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$$

$$c(K) = \left(1+\frac{c}{n}\right)A$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}$$

Example: Computing the Convexity of a Bond

In the following program, the CONVXP function returns the convexity of a bond that has a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, and 14 remaining coupons. The time from settlement date to next coupon date is 0.165, and the annual yield-to-maturity is 0.08.

```
data test;
   dcl double c;
   method run();
      c=convxp(1000,.01,4,14,.33/2,.08);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
11.7290019868346
```

See Also

Functions:

- “CONVX Function” on page 471

COS Function

Returns the cosine in radians.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

COS(*expression*)

Arguments

expression
is any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the COS function:

```
data test (overwrite=yes);
   dcl double x y z;
   method run();
      x=cos(0.5);
      y=cos(0);
      z=cos(3.14159/3);
      put x;
      put y;
      put z;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
0.87758256189037
1
0.50000076602519
```

COSH Function

Returns the hyperbolic cosine in radians.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

COSH(*expression*)

Arguments

expression
is any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The COSH function returns the hyperbolic cosine of the argument, given by the following equation.

$$(e^{argument} + e^{-argument})/2$$

Example

The following program illustrates the COSH function:

```
data test (overwrite=yes);
   dcl double w x y z;
   method run();
      w=cosh(0);
      x=cosh(-5);
      y=cosh(4.37);
      z=cosh(0.5);
      put w;
      put x;
      put y;
```

```
      put z;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1
74.2099485247878
39.5281414700662
1.12762596520638
```

COT Function

Returns the cotangent.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

COT(*argument*)

Arguments

argument
specifies a numeric constant, variable, or expression and is expressed in radians.

Restriction *argument* cannot be 0 or a multiple of PI.

Comparisons

The COT function is related to the TAN function in this way:

```
cot(x)=1/tan(x)
```

Example

The following program illustrates the COMB function:

```
data test (overwrite=yes);
   dcl double x y z;
```

```
      method run();
         x=cot(0.5);
         y=cot(1);
         z=cot(3.14159/3);
         put x;
         put y;
         put z;
      end;
   enddata;
   run;
```

SAS writes the following output to the log.

```
1.83048772171245
0.64209261593433
0.57735144856346
```

Note: If you use `x=cot(0);`, then the COT function returns a missing value, and a note is written to the log that indicates you entered an invalid argument to the function. This is the correct behavior.

See Also

Functions:

- "COS Function" on page 475
- "CSC Function" on page 489
- "SEC Function" on page 1069
- "SIN Function" on page 1080
- "TAN Function" on page 1121

COUNT Function

Counts the number of times that a specified substring appears within a character string.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

COUNT(*string*, *substring*[, *modifiers*])

Arguments

string
specifies a character constant, variable, or expression in which substrings are to be counted.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

substring
specifies the character constant, variable, or expression to be counted in *string*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

modifiers
is a character constant, variable, or expression that specifies one or more modifiers. The following characters, in uppercase or lowercase, can be used as modifiers:

i or I ignores character case during the count. If this modifier is not specified, COUNT counts only character substrings with the same case as the characters in *substring*.

t or T trims trailing blanks from *string* and *substring*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip If *modifiers* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifiers* can also be expressed as a variable or an expression.

Details

The Basics

The COUNT function searches *string*, from left to right, for the number of occurrences of the specified *substring*, and returns that number of occurrences. If the substring is not found in *string*, COUNT returns a value of 0.

CAUTION
If two occurrences of the specified substring overlap in the string, the result is undefined. For example, COUNT('boobooboo', 'booboo') might return either a 1 or a 2.

Comparisons

The COUNT function counts substrings of characters in a character string, whereas the COUNTC function counts individual characters in a character string.

Example

The following program illustrates the COUNT function:

```
data test (overwrite=yes);
   dcl char(50) xyz expression1 expression2 expression3;
   dcl double howmanythis howmanyis howmanythis_i howmanyis_i
howmanyis_it;
   dcl char(45) variable1 variable3;
   dcl char(3) variable2;
   method run();
      xyz='This is a thistle? Yes, this is a thistle.';
      howmanythis=count(xyz,'this');
      put howmanythis;
      xyz='This is a thistle? Yes, this is a thistle.';
      howmanyis=count(xyz,'is');
      put howmanyis;
      howmanythis_i=count('This is a thistle? Yes, this is a thistle.',
          'this', 'i');
      put howmanythis_i;
      variable1='This is a thistle? Yes, this is a thistle.';
      variable2='is ';
      variable3='i';
      howmanyis_i=count(variable1,variable2,variable3);
      put howmanyis_i;
      expression1='This is a thistle? ' || 'Yes, this is a thistle.';
      expression2=kscan('This is',2) || '     ';
      expression3=compress('i     ' || '     t');
      howmanyis_it=count(expression1,expression2,expression3);
      put howmanyis_it;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
3
6
4
4
6
```

See Also

Functions:

- "COUNTC Function" on page 481
- "COUNTW Function" on page 484
- "FIND Function" on page 625
- "INDEX Function" on page 685

COUNTC Function

Counts the number of characters in a string that appear or do not appear in a list of characters.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

COUNTC(*string*, *charlist*[, *modifiers*])

Arguments

string
specifies a character constant, variable, or expression in which characters are counted.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

charlist
specifies a character constant, variable, or expression that initializes a list of characters. COUNTC counts characters in this list, provided that you do not specify the V modifier in the *modifiers* argument. If you specify the V modifier, then all characters that are not in this list are counted. You can add more characters to the list by using other modifiers.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tips: Enclose a literal string of characters in quotation marks.

If there are no characters in the list after processing the modifiers, COUNTC returns 0.

modifiers
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTC function. Blanks are ignored. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	scans *string* from right to left, instead of from left to right.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores case.
l or L	adds lowercase letters to the list of characters.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *charlist* and *modifier* arguments only once, at the first call to this instance of COUNTC. If you change the value of *charlist* or *modifier* in subsequent calls, the change might be ignored by COUNTC.
p or P	adds punctuation marks to the list of characters.
s or S	adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).
t or T	trims trailing blanks from *string* and *chars*. If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the COUNTC function with the T modifier.
u or U	adds uppercase letters to the list of characters.
v or V	counts characters that do not appear in the list of characters. If you do not specify this modifier, then COUNTC counts characters that do appear in the list of characters.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.
Data type	CHAR, VARCHAR
Tip	If *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks.

Details

The COUNTC function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. If there are no characters in the list of characters to be counted, COUNTC returns zero.

Note: Remember that strings with a CHAR data type are always padded out with blanks to the declared length. Strings with a VARCHAR data type return the length of the actual string instead of the declared length.

Comparisons

The COUNTC function counts individual characters in a character string, whereas the COUNT function counts substrings of characters in a character string.

Example

The following program uses the COUNTC function with and without modifiers to count the number of characters in a string.

```
data test;
   dcl char(24) string a b b_i abc_i abc_iv abc_ivt;
      method run();
         string  = 'Baboons Eat Bananas     ';
         a= countc(string, 'a');
         b= countc(string,'b');
         b_i= countc(string,'b','i');
         abc_i= countc(string,'abc','i');
      /* Scan string for characters that are not "a", "b", */
      /* and "c", ignore case, (and include blanks).       */
         abc_iv  = countc(string,'abc','iv');
      /* Scan string for characters that are not "a", "b", */
      /* and "c", ignore case, and trim trailing blanks.   */
         abc_ivt = countc(string,'abc','ivt');
      end;
enddata;
run;
```

Output 7.6 *Results from Using the COUNTC Function with and without Modifiers*

The SAS System

string	a	b	b_i	abc_i	abc_iv	abc_ivt
Baboons Eat Bananas	5	1	3	8	22	11

See Also

Functions:

- “ANYALNUM Function” on page 292
- “ANYALPHA Function” on page 295
- “ANYCNTRL Function” on page 298
- “ANYDIGIT Function” on page 300
- “ANYGRAPH Function” on page 304
- “ANYLOWER Function” on page 307
- “ANYPRINT Function” on page 311
- “ANYPUNCT Function” on page 314
- “ANYSPACE Function” on page 317
- “ANYUPPER Function” on page 319
- “ANYXDIGIT Function” on page 321
- “COUNT Function” on page 478
- “COUNTW Function” on page 484
- “FINDC Function” on page 628
- “INDEXC Function” on page 687
- “NOTALNUM Function” on page 845
- “NOTALPHA Function” on page 848
- “NOTCNTRL Function” on page 850
- “NOTDIGIT Function” on page 852
- “NOTGRAPH Function” on page 857
- “NOTLOWER Function” on page 860
- “NOTPRINT Function” on page 864
- “NOTPUNCT Function” on page 866
- “NOTSPACE Function” on page 869
- “NOTUPPER Function” on page 872
- “NOTXDIGIT Function” on page 874
- “VERIFY Function” on page 1164

COUNTW Function

Counts the number of words in a character string.

Categories: CAS

Character

Returned data type: DOUBLE

Syntax

COUNTW(*string*[, *chars*[, *modifiers*]])

Arguments

string
specifies a character constant, variable, or expression in which words are counted.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

chars
specifies an optional character constant, variable, or expression that initializes a list of characters. The characters in this list are the delimiters that separate words, provided that you do not use the K modifier in the *modifier* argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to the list by using other modifiers.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

modifiers
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTW function. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	counts from right to left instead of from left to right. Right-to-left counting makes a difference only when you use the Q modifier and the string contains unbalanced quotation marks.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters.
k or K	causes all characters that are not in the list of characters to be treated as delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.

l or L	adds lowercase letters to the list of characters.
m or M	specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear after the first character in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *chars* and *modifier* arguments only once, rather than every time the COUNTW function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make COUNTW run faster when you call it in a loop where *chars* and *modifier* arguments do not change.
p or P	adds punctuation marks to the list of characters.
q or Q	ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of *string* contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.
s or S	adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.
t or T	trims trailing blanks from the *string* and *chars* arguments.
u or U	adds uppercase letters to the list of characters.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

Data type CHAR, VARCHAR

Details

Definition of “Word”

In the COUNTW function, “word” refers to a substring that has one of the following characteristics:

- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters, except if you use the Q modifier and the delimiters are within substrings that have quotation marks

Note: The definition of “word” is the same in both the SCAN function and the COUNTW function.

Delimiter refers to any of several characters that you can specify to separate words.

Using the COUNTW Function in ASCII and EBCDIC Environments

If you use the COUNTW function with only two arguments, the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.

- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

Using Null Arguments

The COUNTW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Using the M Modifier

If you do not use the M modifier, then a word must contain at least one character. If you use the M modifier, then a word can have a length of zero. In this case, the number of words is one plus the number of delimiters in the string, not counting delimiters inside strings that are enclosed in quotation marks when you use the Q modifier.

Example

The following program shows how to use the COUNTW function with the M and P modifiers.

The explanation for the value of `mp` for each string is as follows:

- The period is the delimiter and the m modifier causes the period at the end to refer to a subsequent word with zero length, but still a word. So there is one word before the period and one word after the period for a total of two words.
- No delimiters, so there is only one word.
- The p modifier adds punctuation as a delimiter therefore 3 words.

- The p modifier adds punctuation, so / is a delimiter. The m modifier causes the leading / to refer to a word at beginning with zero length for a total of six words.
- The first \ is an escape character. The second \ is a delimiter, so there are six words.

```
data test;
   dcl char(60) string1 having informat $char60. format $char60.;
   method init();
      string1='The quick brown fox jumps over the lazy dog.'; output;
      string1='        Leading blanks'; output;
      string1='2+2=4'; output;
      string1='/unix/path/names/use/slashes'; output;
      string1='\Windows\Path\Names\Use\Backslashes'; output;
   end;
enddata;
run;

data test_out;
   dcl double default blanks mp;
   method run();
      set test;
      default = countw(string1);
      blanks = countw(string1, ' ');
      mp = countw(string1, '.', 'mp');
      put 'String= ' string1 'Default= ' default 'Blanks= ' blanks
'MP= ' mp;
   end;
enddata;
run;
```

Output 7.7 *Results from Using the COUNTW Function with the M and P Modifiers*

default	blanks	mp	string1
9	9	2	The quick brown fox jumps over the lazy dog.
2	2	1	Leading blanks
2	1	3	2+2=4
5	1	6	/unix/path/names/use/slashes
1	1	6	\Windows\Path\Names\Use\Backslashes

Note: Double backslashes are used in the Windows pathname.

See Also

Functions:

- “COUNT Function” on page 478
- “COUNTC Function” on page 481

- “FINDW Function” on page 637
- “SCAN Function” on page 1055

CSC Function

Returns the cosecant.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

CSC(*argument*)

Arguments

argument
specifies a numeric constant, variable, or expression and is expressed in radians.

Restriction *argument* cannot be 0 or a multiple of PI.

Data type DOUBLE

Comparisons

The CSC function is related to the SIN function in this way:

```
csc(x)=1/sin(x)
```

Example

```
data test(overwrite=yes);
   dcl double x y z;
   method run();
      x=csc(.5);
      y=csc(1);
      z=csc(3.14159/3);
      put x;
      put y;
      put z;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log.

```
2.08582964293348
1.18839510577812
1.15470112806662
```

Note: If you use `x=csc(0);`, then the CSC function returns a missing value, and a note is written to the log that indicates you entered an invalid argument to the function. This is the correct behavior.

See Also

Functions:

- “COS Function” on page 475
- “COT Function” on page 477
- “SEC Function” on page 1069
- “SIN Function” on page 1080
- “TAN Function” on page 1121

CSS Function

Returns the corrected sum of squares.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

CSS(*expression*[, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least one non-null or nonmissing expression is required.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the CSS function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3 x4;
   method run();
      x1=css(5,9,3,6);
      put x1=;
      x2=css(5,8,9,6,.);
      put x2=;
      x3=css(8,9,6,.);
      put x3=;
      x4=css(of x1-x3);
      put x4=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x1=18.75
x2=10
x3=4.66666666666666
x4=101.11574074074
```

See Also

Functions:

- "USS Function" on page 1160

CUMIPMT Function

Returns the cumulative interest paid on a loan between the start and end period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

CUMIPMT(*rate*, *number-of-periods*, *principal-amount*[, *start-period*][, *end-period*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods. *Number-of-periods* must be a positive, whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Data type DOUBLE

start-period
specifies the start period for the calculation.

Data type DOUBLE

end-period
specifies the end period for the calculation.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Data type DOUBLE

Example

- The cumulative interest that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

```
data test;
   dcl double TotalInterest having format dollar10.2;
   method run();
```

```
      TotalInterest= CUMIPMT(0.09/12, 360, 125000, 13, 24, 0);
      put 'Total Interest=' TotalInterest;
   end;
enddata;
run;
```

This computation returns a value of $11,135.23.

- The interest that is paid on the first period of the same loan is computed in the following way:

```
data test;
   dcl double first_period_interest having format dollar10.2;
   method run();
      first_period_interest= CUMIPMT(0.09/12, 360, 125000, 1, 1, 0);
      put 'Total Interest=' first_period_interest;
   end;

enddata;
run;
```

This computation returns a value of $937.50.

See Also

Functions:

- "CUMPRINC Function" on page 493

CUMPRINC Function

Returns the cumulative principal paid on a loan between the start and end period.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

CUMPRINC(*rate*, *number-of-periods*, *principal-amount*[, *start-period*] [, *end-period*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing or null value is specified.

start-period
specifies the start period for the calculation.

Data type DOUBLE

end-period
specifies the end period for the calculation.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Data type DOUBLE

Example

- The cumulative principal that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

```
data test;
   dcl double PrincipalYear2 having format dollar10.2;
   method run();
      PrincipalYear2=CUMPRINC(0.09/12, 360, 125000, 12, 24, 0);
      put 'Principal Year 2 EOP=' PrincipalYear2;
   end;
enddata;
run;
```

  This computation returns a value of $1008.23.

- The principal that is paid on the second year of the same loan with beginning-of-period payments is computed as follows:

```
data test;
   dcl double PrincipalYear2b having format dollar10.2;
   method run();
      PrincipalYear2b = CUMPRINC(0.09/12, 360, 125000, 12, 24, 1);
      put 'Principal Year 2 BOP=' PrincipalYear2b;
   end;
enddata;
run;
```

This computation returns a value of $1000.73.

See Also

Functions:

- "CUMIPMT Function" on page 491

CV Function

Returns the coefficient of variation.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

CV(*expression-1*, *expression-2* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the CV function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3 x4;
   method run();
      x1=cv(5, 9, 3, 6);
      put x1=;
      x2=cv(5, 8, 9, 6, .);
      put x2=;
      x3=cv(8, 9, 6, .);
      put x3=;
      x4=cv(of x1-x3);
      put x4=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x1=43.4782608695652
x2=26.082026547865
x3=19.9242421519819
x4=40.9535392160926
```

DAIRY Function

Returns the derivative of the AIRY function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

DAIRY(*x*)

Arguments

x
specifies a numeric constant, variable, or expression.

Details

The DAIRY function returns the value of the derivative of the AIRY function. (See a list of References.)

Example

The following program illustrates the DAIRY function:

```
data test(overwrite=yes);
   dcl double x y;
   method run();
      x=dairy(2.0);
      put x;
      y=dairy(-2.0);
      put y;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
-0.05309038443365
0.61825902074169
```

DATDIF Function

Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

DATDIF(*sdate*, *edate*, *basis*)

Arguments

sdate
specifies a SAS date value that identifies the starting date.

Data type DATE

Tip If *sdate* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

edate
specifies a SAS date value that identifies the ending date.

Data type DATE

Tip If *edate* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

basis
specifies a character string that represents the day count basis. The following values for *basis* are valid:

'30/360'
specifies a 30-day month and a 360-day year, regardless of the actual number of calendar days in a month or year.

A security that pays interest on the last day of a month will either always make its interest payments on the last day of the month, or it will always make its payments on the numerically same day of a month, unless that day is not a valid day of the month, such as February 30. For more information, see "Method of Calculation for Day Count Basis (30/360)" in *SAS Functions and CALL Routines: Reference*.

Alias '360'

'ACT/ACT'
uses the actual number of days between dates. Each month is considered to have the actual number of calendar days in that month, and each year is considered to have the actual number of calendar days in that year.

Alias *'Actual'*

'ACT/360'
uses the actual number of calendar days in a particular month, and 360 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/360* is used for short-term securities.

'ACT/365'
uses the actual number of calendar days in a particular month, and 365 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/365* is used for short-term securities.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

The DATDIF function has a specific meaning in the securities industry, and the method of calculation is not the same as the actual day count method. Calculations

can use months and years that contain the actual number of days. Calculations can also be based on a 30-day month or a 360-day year. For more information about standard securities calculation methods, see the References section at the bottom of this function.

Note: When counting the number of days in a month, DATDIF *always* includes the starting date and excludes the ending date.

Method of Calculation for Day Count Basis (30/360)

To calculate the number of days between two dates, use the following formula:

$$Number of days = [(Y2 - Y1) * 360] + [(M2 - M1) * 30] + (D2 - D1)$$

Arguments

Y2
: specifies the year of the later date.

Y1
: specifies the year of the earlier date.

M2
: specifies the month of the later date.

M1
: specifies the month of the earlier date.

D2
: specifies the day of the later date.

D1
: specifies the day of the earlier date.

Because all months can contain only 30 days, you must adjust for the months that do not contain 30 days. Do this before you calculate the number of days between the two dates.

The following rules apply:

- If the security follows the End-of-Month rule, and D2 is the last day of February (28 days in a non-leap year, 29 days in a leap year), and D1 is the last day of February, then change D2 to 30.
- If the security follows the End-of-Month rule, and D1 is the last day of February, then change D1 to 30.
- If the value of D2 is 31 and the value of D1 is 30 or 31, then change D2 to 30.
- If the value of D1 is 31, then change D1 to 30.

Example

In the following program, DATDIF returns the actual number of days between two dates, as well as the number of days based on a 30-day month and a 360-day year.

```
data test (overwrite=yes);
   dcl date sdate edate;
   dcl double actual days360;
   method run();
      sdate= date'1978-10-16';
      edate= date'1996-02-16';
      sassdate=to_double(sdate);
      sasedate=to_double(edate);
      actual=datdif(sassdate, sasedate, 'act/act');
      days360=datdif(sassdate, sasedate, '30/360');
      put  'Actual= ' actual;
      put 'Days 360 = ' days360;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
Actual= 6332
Days 360= 6240
```

See Also

Functions:

- “YRDIF Function” on page 1188

References

Securities Industry Association 1994. *Standard Securities Calculation Methods - Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, NY: Securities Industry Association.

DATE Function

Returns the current date as a SAS date value.

Categories:	CAS Date and Time
Alias:	TODAY
Returned data type:	DOUBLE

Syntax

DATE( )

Without Arguments

The DATE function has no arguments.

Comparisons

The DATE function does not take any arguments. The SAS date value returned is the number of days from January 1, 1960 to the current date.

For more information about how DS2 handles dates, see "DS2 Expressions" in *SAS DS2 Programmer's Guide*.

Example

The following program illustrates the DATE function:

```
data test(overwrite=yes);
   dcl double x having format nldate.;
   method run();
      x=date();
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
January 22, 2019
```

See Also

Functions:

- "TODAY Function" on page 1140

DATEJUL Function

Converts a Julian date to a SAS date value.

Categories: CAS

Date and Time

Returned data type: DOUBLE

Syntax

DATEJUL(*julian-date*)

Arguments

julian-date

specifies any valid expression that evaluates to a numeric value and that represents a Julian date. A Julian date is a date in the form *yyddd* or *yyyyddd*, where *yy* or *yyyy* is a two-digit or four-digit whole number that represents the year and *ddd* is the number of the day of the year. The value of *ddd* must be between 1 and 365 (or 366 for a leap year).

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

A SAS date value is the number of days from January 1, 1960 to a specified date. The DATEJUL function returns the number of days from January 1, 1960 to the Julian date specified in *julian-date*.

For more information about how dates are handled in DS2, see "DS2 Expressions" in *SAS DS2 Programmer's Guide* .

Example

The following program illustrates the DATEJUL function:

```
data test(overwrite=yes);
   dcl double xstart xend having format nldate.;
   method run();
      xstart=datejul(94365);
      put xstart;
      xend=datejul(2001001);
      put xend;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
December 31, 1994
January 01, 2001
```

See Also

Functions:

- "JULDATE Function" on page 765

DATEPART Function

Extracts the date from a SAS datetime value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

DATEPART(*datetime*)

Arguments

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

A SAS datetime value is the number of seconds between January 1, 1960 and the hour, minute, and seconds within a specific date. The DATEPART function determines the date portion of the SAS datetime value and returns the date as a SAS date value, which is the number of days from January 1, 1960.

Example

The following program illustrates the DATEPART function:

```
data test(overwrite=yes);
   dcl double dtvalue;
   dcl double dp having format date9.;
   method run();
      dtvalue=1652165417;
      dp=datepart(dtvalue);
      put dp;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
09MAY2012
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "DATETIME Function" on page 504
- "TIMEPART Function" on page 1125

DATETIME Function

Returns the current date and time of day as a SAS datetime value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

DATETIME( )

Comparisons

The DATETIME function does not take any arguments. The SAS datetime value returned is the number of seconds from January 1, 1960 to the current date and time.

Example

The following program illustrates the DATETIME function:

```
data test(overwrite=yes);
   dcl double dt ;
   method run();
      dt=datetime();
      put dt;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1865079581.31599
```

See Also

Concepts:

- "Dates and Times in DS2" in *SAS DS2 Programmer's Guide*

Functions:

- "DATE Function" on page 500
- "TIME Function" on page 1124

DAY Function

Returns the day of the month from a SAS date value.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

DAY(*date*)

Arguments

date

specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The DAY function produces a whole number from 1 to 31 that represents the day of the month.

A SAS date value is the number of days from January 1, 1960 to a specific date.

Example

The following program illustrates the DAY function where `dayvalue`, the SAS date value, has a value of 22541, which is the 18th day of the month:

```
data test(overwrite=yes);
   dcl double dv;
   method run();
      dv=day(22541);
      put dv;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
18
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "MONTH Function" on page 834
- "YEAR Function" on page 1185

DEQUOTE Function

Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

DEQUOTE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The value that is returned by the DEQUOTE function depends on the first character or the first two characters in *expression*:

- If the first character of *expression* is not a quotation mark, DEQUOTE returns a syntax error.
- If the first character of *expression* is a single quotation mark, the DEQUOTE function removes that single quotation mark from the result. DEQUOTE then scans *expression* from left to right, looking for more single quotation marks or double quotation marks.

  All paired single quotation marks are replaced with a single quotation mark.

  All paired double quotation marks are retained.

  If a double quotation mark is the second character, DEQUOTE removes the double quotation mark from the result. DEQUOTE then scans *expression* from left to right. If a matching double quotation mark is found, the text between the double quotation marks is returned. Any text to the right of the closing double quotation mark, to the end of *expression* is removed from the result.

The first non-paired single quotation mark in *expression* is the closing single quotation mark and is removed.

If a close parentheses follows the close single quotation mark, the function returns the dequoted string. If characters exist to the right of the close single quotation mark, the function results in a syntax error and the error is printed in the SAS log.

- If *expression* is enclosed in double quotation marks, the DEQUOTE function returns a null or missing value.

Note: If *expression* is a constant enclosed in quotation marks, those quotation marks are not part of the value of *expression*. Therefore, you do not need to use DEQUOTE to remove the quotation marks that denote a constant.

Examples

Example 1: No Quotation Marks

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(75) string;
   method run();
      string= dequote(Omitting quotation marks causes an error);
      put string;
   end;
enddata;
run;
```

SAS writes the following errors to the log.

```
ERROR: Compilation error.
ERROR: Parse encountered identifier when expecting ')'.
ERROR: Line 898: Parse failed: string= dequote(Omitting  >>> quotation <<<  marks causes an error
```

Example 2: No Leading Quotation Marks in the String

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(75) string;
   method run();
      string= dequote(No 'leading' quotation marks cause an error);
      put string;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
ERROR: Compilation error.
ERROR: Parse encountered constant when expecting ')'.
ERROR: Line 909: Parse failed: string= dequote(No  >>> 'leading' <<<  quotation marks caus
```

Example 3: Single Matched Quotation Marks

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(75) string;
   method run();
      string = dequote(' "Single matched quotation marks are removed"
');
      put string;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
"Single matched quotation marks are removed"
```

Example 4: Matched Double Quotation Marks

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(75) string;
   method run();
      string= dequote("Matched double quotation marks cause an
error.");
      put string;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
ERROR: Compilation error.
WARNING: Line 1001: No DECLARE for referenced variable "matched double quotation marks causes an
         error"; creating it as a global variable of type double.
ERROR: BASE driver, Column name Matched double quotation marks causes an error is too long for a
       SAS name
```

Example 5: Paired Single Quotation Marks

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(75) string;
   method run();
       string= dequote('Paired '' single '' quotation marks are
reduced to a
          single quotation mark');
       put string;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log.

```
Paired ' single ' quotation marks are reduced to a single quotation mark
```

Example 6: Double Quotation Marks within Single Quotation Marks with a Space before the Open Quotation Mark

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(100) string;
   method run();
       string= dequote(' "Double quotation marks within single
quotation marks,
          with space before open quotation mark" ');
       put string;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
 "Double quotation marks within single quotation marks, with space before open quotation mark"
```

Example 7: Double Quotation Marks within Single Quotation Marks without a Space before the Open Quotation Mark

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(110) string;
   method run();
       string= dequote(dequote('"Double quotation marks within single
quotation
          marks, without space before open quotation mark causes an
error"'););
       put string;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
ERROR: Compilation error.
ERROR: Parse encountered ';' when expecting ')'.
ERROR: Line 968: Parse failed:  quotation mark causes an error"') >>> ; <<< );
```

Example 8: Test after Closing Double Quotation Mark

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(75) string;
   method run();
```

```
        string= dequote(' "Text after closing double quotation mark"
           is removed ');
        put string;
   end;
enddata;
run;
```

SAS writes the following errors to the log.

```
"Text after closing double quotation mark" is removed
```

Example 9: No Matching Quotation Mark

The following program illustrates the DEQUOTE function:

```
data test(overwrite=yes);
   dcl char(75) string;
   method run();
        string= dequote(' No matching quotation mark);
        put string;
   end;
enddata;
run;
```

Statement execution does not complete. Submit the following characters to complete the execution: ');

DEVIANCE Function

Returns the deviance based on a probability distribution.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

DEVIANCE(*distribution*, *variable*, *shape-parameter(s)*[, *ε*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. Valid distributions are listed in the following table:

Distribution	Argument
Bernoulli (p. 512)	'BERNOULLI' | 'BERN'
Binomial (p. 513)	'BINOMIAL' | 'BINO'
Gamma (p. 513)	'GAMMA'
Inverse Gauss (Wald) (p. 514)	'IGAUSS' | 'WALD'
Normal (p. 514)	'NORMAL' | 'GAUSSIAN'
Poisson (p. 515)	'POISSON' | 'POIS'

variable
is a numeric constant, variable, or expression.

shape-parameter(s)
are one or more distribution-specific numeric parameters that characterize the shape of the distribution.

ε
is an optional numeric small value used for all of the distributions, except for the normal distribution.

Details

The Bernoulli Distribution

DEVIANCE('BERNOULLI', *variable*, *p*[, *ε*])

Arguments

variable
is a binary numeric random variable that has the value of 1 for success and 0 for failure.

p
is a numeric probability of success with $\varepsilon \le p \le 1-\varepsilon$.

ε
is an optional positive numeric value that is used to bound *p*. Any value of *p* in the interval $0 \le p \le \varepsilon$ is replaced by *ε*. Any value of *p* in the interval $1 - \varepsilon \le p \le 1$ is replaced by $1 - \varepsilon$.

The DEVIANCE function returns the deviance from a Bernoulli distribution with a probability of success *p*, where success is defined as a random variable value of 1. The equation follows:

$$\text{DEVIANCE}('BERN', variable, p, \varepsilon) = \begin{cases} -2\log(1-p) & x = 0 \\ -2\log(p) & x = 1 \\ . & otherwise \end{cases}$$

The Binomial Distribution

DEVIANCE('BINO', *variable*, *μ*, *n*[, *ε*])

Arguments

variable
is a numeric random variable that contains the number of successes.

Range 0 ≤ *variable* ≤ 1

μ
is a numeric mean parameter.

Range $n\varepsilon \le \mu \le n(1-\varepsilon)$

n
is a whole number of Bernoulli trials parameter

Range $n \ge 0$

ε
is an optional positive numeric value that is used to bound *μ*. Any value of *μ* in the interval $0 \le \mu \le n\varepsilon$ is replaced by *nε*. Any value of *μ* in the interval $n(1-\varepsilon) \le \mu \le n$ is replaced by $n(1-\varepsilon)$.

The DEVIANCE function returns the deviance from a binomial distribution, with a probability of success *p*, and a number of independent Bernoulli trials *n*. The following equation describes the DEVIANCE function for the Binomial distribution, where *x* is the random variable:

$$\text{DEVIANCE}('BINO', x, \mu, n) = \begin{cases} . & x < 0 \\ 2\left(x\log\left(\frac{x}{\mu}\right) + (n-x)\log\left(\frac{n-x}{n-\mu}\right)\right) & 0 \le x \le n \\ . & x > n \end{cases}$$

The Gamma Distribution

DEVIANCE('GAMMA', *variable*, *μ*[, *ε*])

Arguments

variable
is a numeric random variable.

Range *variable* ≥ *ε*

μ
is a numeric mean parameter.

Range $\mu \ge \varepsilon$

ε
is an optional positive numeric value that is used to bound *variable* and *μ*. Any value of *variable* in the interval $0 \le variable \le \varepsilon$ is replaced by *ε*. Any value of *μ* in the interval $0 \le \mu \le \varepsilon$ is replaced by *ε*.

The DEVIANCE function returns the deviance from a gamma distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the gamma distribution, where x is the random variable:

$$\text{DEVIANCE}('GAMMA', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(-\log\left(\frac{x}{\mu}\right) + \frac{x-\mu}{\mu}\right) & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

The Inverse Gauss (Wald) Distribution

DEVIANCE('IGAUSS' | 'WALD', *variable*, μ[, ε])

Arguments

variable
is a numeric random variable.

Range *variable* ≥ ε

μ
is a numeric mean parameter.

Range $\mu \geq \varepsilon$

ε
is an optional positive numeric value that is used to bound *variable* and μ. Any value of *variable* in the interval 0 ≤ *variable* ≤ ε is replaced by ε. Any value of μ in the interval 0 ≤ μ ≤ ε is replaced by ε.

The DEVIANCE function returns the deviance from an inverse Gaussian distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the inverse Gaussian distribution, where x is the random variable:

$$\text{DEVIANCE}('IGAUSS', x, \mu) = \begin{cases} . & x < 0 \\ \frac{(x-\mu)^2}{\mu^2 x} & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

The Normal Distribution

DEVIANCE('NORMAL' | 'GAUSSIAN', *variable*, μ)

Arguments

variable
is a numeric random variable.

μ
is a numeric mean parameter.

The DEVIANCE function returns the deviance from a normal distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the normal distribution, where x is the random variable:

$$\text{DEVIANCE}('NORMAL', x, \mu) = (x-\mu)^2$$

The Poisson Distribution

DEVIANCE('POISSON', *variable*, *μ*[, *ε*])

Arguments

variable
is a numeric random variable.

Range *variable* ≥ 0

μ
is a numeric mean parameter.

Range $\mu \geq \varepsilon$

ε
is an optional positive numeric value that is used to bound *μ*. Any value of *μ* in the interval $0 \leq \mu \leq \varepsilon$ is replaced by *ε*.

The DEVIANCE function returns the deviance from a Poisson distribution with a mean parameter *μ*. The following equation describes the DEVIANCE function for the Poisson distribution, where *x* is the random variable:

$$\text{DEVIANCE}('POISSON', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(x log\left(\frac{x}{\mu}\right) - (x - \mu)\right) & x \geq 0, \mu \geq \varepsilon \end{cases}$$

DHMS Function

Returns a SAS datetime value from date, hour, minute, and second values.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

DHMS(*date*, *hour*, *minute*, *second*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

hour
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

minute
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

second
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

Examples

Example 1: Using the DHMS Function

The following program illustrates the DHMS function:

```
data test(overwrite=yes);
   dcl double dtvalue;
   method run();
      dtvalue=dhms(mdy(12,31,2018),12,01,01);
      put dtvalue;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1861876861
```

Example 2: Combining Date and Time Values

The following program illustrates how to combine a SAS date value with a SAS time value into a SAS datetime value, using the current day and time.

```
data test(overwrite=yes);
   dcl double day tme dt;
   method run();
      day=date();
      tme=time();
      dt=dhms(day,0,0,tme);
      put dt;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1863788289.77899
```

Example 3: Using Datetime Values

The following program illustrates how to use the DHMS function with datetime values.

```
data test(overwrite=yes);
   dcl double dtid dtid1 dtid2 having format datetime.;
   method run();
      dtid=dhms(mdy(01, 03, 2018), 15, 30, 15);
      put dtid;
      dtid1=dhms(mdy(01, 03, 2018), 15, 30, 61);
      put dtid1;
      dtid2=dhms(mdy(01, 03, 2018), 15, .5, 15);
      put dtid2;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
03JAN18:15:30:15
03JAN18:15:31:01
03JAN18:15:00:45
```

See Also

Concepts:

- "Dates and Times in DS2" in *SAS DS2 Programmer's Guide*

Functions:

- "HMS Function" on page 676

DIF Function

Returns differences between an argument and its *n*th lag.

Categories: CAS
Special

Returned data type: Same as the expression's data type

Syntax

DIF[*n*] (*expression*)

Arguments

n
specifies the number of lags.

Range 1 to 100

Data type INTEGER

expression
specifies any valid expression that evaluates to a numeric value.

Data type BIGINT, DECIMAL, DOUBLE, INTEGER, TINYINT

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The DIF functions, DIF1, DIF2, ..., DIF*n*, return the differences between the expression value and its *n*th lag. DIF1 can also be written as DIF. DIF*n* is defined as DIF*n*(*x*)=*x*-LAG*n*(*x*).

Note: To avoid unexpected behavior with data underflow or overflow, if expression is a SMALLINT or a TINYINT, DS2 converts expression to INTEGER before processing the DIF function. After processing, DS2 converts expression back to SMALLINT and TINYINT and handle the underflow or overflow in the same way it is done for all other processing.

For details about storing and returning values from the LAG*n* queue, see the LAG function.

Comparisons

The function DIF2(X) is not equivalent to the difference DIF(DIF(X)).

Examples

Example 1

The following program illustrates the difference between the LAG and DIF functions.

```
proc ds2;
data one (overwrite=yes);
    declare int x z d;
    method run();
       do x=1,2,6,4,7;
          z=lag(x);
          d=dif(x);
          output;
       end;
    end;
enddata;
run;
quit;

proc print data=one;
run;
```

Output 7.8 *Difference between the DIF and LAG Functions*

The SAS System

Obs	X	Z	D
1	1	.	.
2	2	1	1
3	6	2	4
4	4	6	-2
5	7	4	3

Example 2: Using Expressions as Arguments

The following program illustrates how string expressions are converted to doubles.

```
data _null_;
   dcl char(8) i difi;
   method init();
      do i=' 2', '  4', '   6', '10  ', 'abc';
         difi=dif(i);
         put i= difi=;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
i= 2       difi=.
i=  4      difi=2
i=   6     difi=2
i=10       difi=4
i=abc      difi=.
```

See Also

Functions:

- "LAG Function" on page 770

DIGAMMA Function

Returns the value of the digamma function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

DIGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction	Zero and negative integers are not valid.
Data type	DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The DIGAMMA function returns the ratio that is given by the following equation.

$$\Psi(x) = \Gamma'(x)/\Gamma(x)$$

$\Gamma(.)$ and $\Gamma'(.)$ denote the gamma function and its derivative, respectively. For *expression*>0, the DIGAMMA function is the derivative of the lgamma function.

Example

The following program illustrates the DIGAMMA function:

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=digamma(1.0);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
-0.57721566490153
```

DIM Function

Returns the number of elements in an array.

Categories:	Array CAS
Returned data type:	INTEGER

Syntax

DIM(*array-name*[, *bound-n*])

Arguments

array-name
specifies the name of a temporary or a variable array.

bound-n
is a numeric constant, variable, or expression that specifies the dimension, in a multidimensional array, for which you want to know the number of elements. If no *bound-n* value is specified, the DIM function returns the number of elements in the first dimension of the array.

Bound-n evaluates to an integral value.

Data type INTEGER

Details

The DIM function returns the number of elements in a one-dimensional array, or the number of elements in a specified dimension of a multidimensional array.

If the DIM function is called with a *bound-n* dimension value that is outside the dimension of the array, then a run-time error occurs and the function returns a NULL integer value.

Comparisons

- DIM returns the number of elements in an array dimension.
- HBOUND returns the value of the upper bound of an array dimension.
- LBOUND returns the value of the lower bound of an array dimension.
- NDIMS returns the number of dimensions in an array.

Example

The following program shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions:

```
data _null_;

   method init();
      declare char(15) a1[4];
      declare double   a2[2,3,4] sum;
```

```
        a1 := ('red' 'yellow' 'green' 'blue');
        a2 := (24*2.0);

        do i = 1 to dim(a1);
           put a1[i];
        end;

        numelems = 0;
        do i = 1 to ndims(a2);
           numelems = numelems + dim(a2, i);
        end;

        sum = 0;

        do i = lbound(a2, 1) to hbound(a2, 1);
          do j = lbound(a2, 2) to hbound(a2, 2);
            do k = lbound(a2, 3) to hbound(a2, 3);
               sum = sum + a2[i,j,k];
            end;
          end;
        end;

        put sum=;
     end;
  enddata;
  run;
```

SAS writes the following output to the log:

```
red
yellow
green
blue
sum=48
```

See Also

Functions:

- “HBOUND Function” on page 674
- “LBOUND Function” on page 776
- “NDIMS Function” on page 838

DIVIDE Function

Returns the result of a division that handles special missing values for ODS output.

Categories: Arithmetic
CAS

Returned data type: DOUBLE

Syntax

DIVIDE(*x*, *y*)

Arguments

x

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

y

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The DIVIDE function divides two numbers and returns a result that is compatible with ODS conventions. The function handles special missing values for ODS output. The following list shows how certain special missing values are interpreted in ODS:

- .I as infinity
- .M as minus infinity
- ._ as a blank

The following table shows the values that are returned by the DIVIDE function, based on the values of *x* and *y*.

Figure 7.1 *Values That Are Returned by the DIVIDE Function*

y \ x	positive	zero	negative	.I	.M	._	other
positive	x/y or .I	0	x/y or .M	.I	.M	._	x
zero	.I	.	.M	.I	.M	._	x
negative	x/y or .M	0	x/y or .I	.M	.I	._	x
.I	0	0	0	.	.	._	x
.M	0	0	0	.	.	._	x
._	._	._	._	._	._	._	._
other	y	y	y	y	y	._	x

Note: The DIVIDE function never writes a note to the SAS log regarding missing values, division by zero, or overflow.

Example

The following program shows the results of using the DIVIDE function.

```
data test (overwrite=yes);
   dcl double a b c d;
   method run();
      a=divide(1, 0);
      put a='(infinity)';
      b=divide(2, .I);
      put b=;
      c=divide(.I, -1);
      put c='(minus infinity)';
      d=divide(constant('big'), constant('small'));
      put d='(infinity because of overflow)';
   end;
enddata;
run;
```

The following lines are written to the SAS log:

```
a=I (infinity)
b=0
c=M (minus infinity)
d=I (infinity because of overflow)
```

DUR Function

Returns the modified duration for an enumerated cash flow.

Categories: CAS

Financial

Returned data type: DOUBLE

Syntax

DUR(*y*, *f*, *c(1), ..., c(k)*)

Arguments

y

specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Data type DOUBLE

f

specifies the frequency of cash flows per period.

Range $f > 0$

Data type DOUBLE

c(1), ... ,c(k)

specifies a list of cash flows.

Data type DOUBLE

Details

The DUR function returns the value from the following equation.

$$C = \sum_{k=1}^{K} \frac{k\left(\frac{c(k)}{(1+y)^{\frac{k}{f}}}\right)}{(P(1+y)f)}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example: Using the DUR Function

```
data test;
   dcl double d;
```

```
   method run();
      d=dur(.05,1,.33,.44,.55,.49,.50,.22,.4,.8,.01,.36,.2,.4);
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
5.28402498798216
```

See Also

Functions:

- "DURP Function" on page 527

DURP Function

Returns the modified duration for a periodic cash flow stream, such as a bond.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

DURP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal per-period coupon rate, expressed as a fraction.

Range $0 \le c < 1$

Data type DOUBLE

n

specifies the number of coupons per period.

Range *n* > 0 and is a whole number

Data type DOUBLE

K

specifies the number of remaining coupons.

Range *K* > 0 and is a whole number

Data type DOUBLE

k_o

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_o \leq 1/n$

Data type DOUBLE

y

specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range *y* > 0

Data type DOUBLE

Details

The DURP function returns the value from the following equation.

$$D = \frac{1}{n} \frac{\sum_{k=1}^{K} t_k \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$
- $c(K) = \left(1+\frac{c}{n}\right)A$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}$$

Example: Using the DURP Function

```
data test;
   dcl double d;
   method run();
      d=durp(1000,1/100,4,14,.33/2,.10);
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
3.33170731707317
```

See Also

Functions:

- "DUR Function" on page 525

EFFRATE Function

Returns the effective annual interest rate.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

EFFRATE(*compounding-interval*, *rate*)

Arguments

compounding-interval
is a SAS interval. This value represents how often *rate* compounds.

Data type CHAR

rate
is numeric. *Rate* is a nominal annual interest rate (expressed as a percentage) that is compounded at each compounding interval.

Data type DOUBLE

Details

The EFFRATE function returns the effective annual interest rate. The function computes the effective annual interest rate that corresponds to a nominal annual interest rate.

The following details apply to the EFFRATE function:

- The values for rates must be at least –99.
- In considering a nominal interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by EFFRATE equals $e^{rate/100}-1$.

  If *compounding-interval* is not 'CONTINUOUS', and *m* compounding intervals occur in a year, the value that is returned by EFFRATE equals $(1+[rate/100\ m])^{m-1}$.
- The following values are valid for *compounding-interval*:
  - 'CONTINUOUS'
  - 'DAY'
  - 'SEMIMONTH'
  - 'MONTH'
  - 'QUARTER'
  - 'SEMIYEAR'
  - 'YEAR'
- If the interval is 'DAY', then *m*=365.

Example

The following programs show how the effective rate is calculated:

- If a nominal rate is 10%, this program shows the corresponding effective rate when interest is compounded monthly.

```
data test(overwrite=yes);
   dcl double effectiverate1;
   method run();
      effectiverate1=effrate('month', 10);
      put effectiverate1;
   end;
enddata;
run;
```

  The effective rate is 10.4713067441296.

- If a nominal rate is 10%, this program shows the corresponding effective rate when interest is compounded quarterly.

```
data test(overwrite=yes);
   dcl double effectiverate1;
   method run();
      effectiverate1=effrate('quarter', 10);
      put effectiverate1;
   end;
enddata;
run;
```

The effective rate is 10.3812890624999.

ERF Function

Returns the value of the (normal) error function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

ERF(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ERF function returns the integral, given by the following:

$$\mathrm{ERF}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \varepsilon^{-z^2} dz$$

You can use the ERF function to find the probability (p) that a normally distributed random variable with mean 0 and standard deviation takes on a value less than X.

For example, the quantity that is given by the following statement is equivalent to PROBNORM(X):

```
p=.5+.5*erf(x/sqrt(2));
```

Example

The following program illustrates the ERF function:

```
data test(overwrite=yes);
   dcl double x y;
   method run();
      x=erf(1);
      y=erf(-1);
      put x=;
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x=0.84270079294971
y=-0.84270079294971
```

See Also

Functions:

- “ERFC Function” on page 532
- “PROBNORM Function” on page 975

ERFC Function

Returns the value of the complementary (normal) error function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

ERFC(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ERFC function returns the complement to the ERF function (that is, 1 – ERF(*argument*)).

Example

The following program illustrates the ERFC function:

```
data test(overwrite=yes);
   dcl double x y;
   method run();
      x=erfc(1);
      y=erfc(-1);
      put x=;
      put y=;
  end;
enddata;
run;
```

SAS writes the following output to the log.

```
x=0.15729920705028
y=1.84270079294971
```

See Also

Functions:

- "ERF Function" on page 531

EXP Function

Returns the value of the e constant raised to a specified power.

Categories: CAS

Mathematical

Returned data type: DOUBLE

Syntax

EXP(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The EXP function raises the constant **e** to the power that is supplied by the argument. **e** is approximately given by 2.71828. The result is limited by the maximum value of a double decimal value on the computer.

Examples

Example 1: Using the EXP Function

The following program illustrates the EXP function:

```
data test(overwrite=yes);
   dcl double x y;
   method run();
      x=exp(1.0);
      y=exp(0);
      put x=;
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x=2.71828182845904
y=1
```

Example 2: Filling a Matrix Package Array

The following program creates two matrix packages and fills them with values of the EXP function:

```
data _null_;
   dcl double a[3, 3];
   dcl double c[3, 3];

   method run();
      dcl package matrix m;
      dcl package matrix r;
      dcl double i j;

      a := (1, 2, 3, 1, 2, 3, 1, 2, 3);

      m=_new_ matrix(a, 3, 3);
      r=m.exp();
      r.toarray(c);

      do i=1 to 3;
         do j=1 to 3;
            put c[i, j];
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2.71828182845904
7.38905609893065
20.0855369231876
2.71828182845904
7.38905609893065
20.0855369231876
2.71828182845904
7.38905609893065
20.0855369231876
```

FACT Function

Computes a factorial.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

FACT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The mathematical representation of the FACT function is given by the following equation:

$$FACT(n) = n!$$

In this equation, $n \geq 0$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the FACT function.

Example

The following program illustrates the FACT function:

```
data;
  dcl double x y z;
  method run();
      x=fact(5);
      y=fact(-27);
      /* decimal #s do not work */
      z=fact(7.3);
      put x=;
      put y=;
      put z=;
  end;
enddata;
run;
```

/* negative #s don't work */ SAS writes the following output to the log.

```
x=120
y=.
z=.
ERROR: Invalid argument to function fact.
ERROR: Invalid argument to function fact.
```

See Also

Functions:

- "COMB Function" on page 435
- "PERM Function" on page 935

FINANCE Function

Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.

Categories: CAS
Financial

Syntax

FINANCE(*string-identifier*, *parameter–1*, *parameter–2*, ...)

Arguments

string-identifier
specifies a character constant, variable, or expression. Valid values for *string-identifier* are listed in the following table.

string-identifier	**Description**
'ACCRINT'	computes the accrued interest for a security that pays periodic interest.
'ACCRINTM'	computes the accrued interest for a security that pays interest at maturity.
' AMORDEGRC'	computes the depreciation for each accounting period by using a depreciation coefficient.
'AMORLINC'	computes the depreciation for each accounting period.
'COUPDAYBS'	computes the number of days from the beginning of the coupon period to the settlement date.
'COUPDAYS'	computes the number of days in the coupon period that contains the settlement date.

string-identifier	Description
'COUPDAYSNC'	computes the number of days from the settlement date to the next coupon date.
'COUPNCD'	computes the next coupon date after the settlement date.
'COUPNUM'	computes the number of coupons that are payable between the settlement date and the maturity date.
'COUPPCD'	computes the previous coupon date before the settlement date.
'CUMIPMT'	computes the cumulative interest that is paid between two periods.
'CUMPRINC'	computes the cumulative principal that is paid on a loan between two periods.
'DB'	computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
'DDB'	computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.
'DISC'	computes the discount rate for a security.
'DOLLARDE'	converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
'DOLLARFR'	converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.
'DURATION'	computes the annual duration of a security with periodic interest payments.
'EFFECT'	computes the effective annual interest rate.
'FV'	computes the future value of an investment.
'FVSCHEDULE'	computes the future value of an initial principal after applying a series of compound interest rates.
'INTRATE'	computes the interest rate for a fully invested security.
'IPMT'	computes the interest payment for an investment for a given period.

string-identifier	Description
'IRR'	computes the internal rate of return for a series of cash flows.
'ISPMT'	calculates the interest paid during a specific period of an investment.
'MDURATION'	computes the Macaulay modified duration for a security with an assumed face value of $100.
'MIRR'	computes the internal rate of return where positive and negative cash flows are financed at different rates.
'NOMINAL'	computes the annual nominal interest rate.
'NPER'	computes the number of periods for an investment.
'NPV'	computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
'ODDFPRICE'	computes the price per $100 face value of a security with an odd first period.
'ODDFYIELD'	computes the yield of a security with an odd first period.
'ODDLPRICE'	computes the price per $100 face value of a security with an odd last period.
'ODDLYIELD'	computes the yield of a security with an odd last period.
'PMT'	computes the periodic payment for an annuity.
'PPMT'	computes the payment on the principal for an investment for a given period.
'PRICE'	computes the price per $100 face value of a security that pays periodic interest.
'PRICEDISC'	computes the price per $100 face value of a discounted security.
'PRICEMAT'	computes the price per $100 face value of a security that pays interest at maturity.
'PV'	computes the present value of an investment.

string-identifier	Description
'RATE'	computes the interest rate per period of an annuity.
'RECEIVED'	computes the amount received at maturity for a fully invested security.
'SLN'	computes the straight-line depreciation of an asset for one period.
'SYD'	computes the sum-of-years digits depreciation of an asset for a specified period.
'TBILLEQ'	computes the bond-equivalent yield for a treasury bill.
'TBILLPRICE'	computes the price per $100 face value for a treasury bill.
'TBILLYIELD'	computes the yield for a treasury bill.
'VDB'	computes the depreciation of an asset for a specified or partial period by using a declining balance method.
'XIRR'	computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
'XNPV'	computes the net present value for a schedule of cash flows that is not necessarily periodic.
'YIELD'	computes the yield on a security that pays periodic interest.
'YIELDDISC'	computes the annual yield for a discounted security (for example, a treasury bill).
'YIELDMAT'	computes the annual yield of a security that pays interest at maturity.

parameter
specifies a parameter that is associated with each *string-identifier*. The following parameters are available:

basis
is an optional parameter that specifies a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360

Numeric Value	String Value	Day Count Method
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

interest-rates
specifies rates that are provided as numeric values and not as percentages.

dates
specifies that all dates in the financial functions are SAS dates.

sign-of-cash-values
for all the arguments, specifies that the cash that you pay out, such as deposits to savings or other withdrawals, is represented by negative numbers. It also specifies that the cash that you receive, such as dividend checks and other deposits, is represented by positive numbers.

FINANCE ACCRINT Function

Computes the accrued interest for a security that pays periodic interest.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ACCRINT', *issue*, *first-interest*, *settlement*, *rate*, *par-value*, *frequency*, [*basis*]);

Arguments

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

first-interest
specifies the first interest date of the security.

Requirement *First-interest* is a SAS date.

Data type DOUBLE

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

par-value
specifies the par value of the security. If you omit *par-value*, SAS uses the value `$1000`.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Accrued Interest: ACCRINT

The following program computes the accrued interest for a security that pays periodic interest.

```
data test(overwrite=yes);
   dcl double issue firstinterest settlement rate;
   dcl double par frequency basis r;
   method init();
      issue=mdy(2, 28, 2016);
      firstinterest=mdy(8, 31, 2016);
      settlement=mdy(5, 1, 2016);
      rate=0.1;
      par=1000;
      frequency=2;
      basis=1;
      r=finance('accrint', issue, firstinterest,
         settlement, rate, par, frequency, basis);
     put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=17.1225513616818
```

FINANCE ACCRINTM Function

Computes the accrued interest for a security that pays interest at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ACCRINTM', *issue*, *settlement*, *rate*, *par-value*, [*basis*]);

Arguments

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

par-value
specifies the par value of the security. If you omit *par-value*, SAS uses the value `$1000`.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Accrued Interest: ACCRINTM

program

```
data test(overwrite=yes);
   dcl double issue maturity rate;
   dcl double par basis r;
   method init();
      issue=mdy(2, 28, 2015);
```

```
        maturity=mdy(8, 31, 2015);
        rate=0.1;
        par=1000;
        basis=0;
        r=finance('accrintm', issue, maturity, rate, par, basis);
        put r=;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=50.5555555555555
```

FINANCE AMORDEGRC Function

Computes the depreciation for each accounting period by using a depreciation coefficient.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('AMORDEGRC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, [*basis*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

date-purchased
specifies the date of the purchase of the asset.

Requirement *Date-purchased* is a SAS date.

Data type DOUBLE

first-period
specifies the date of the end of the first period.

Requirement *First-period* is a SAS date.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

period
specifies the depreciation period.

Data type DOUBLE

rate
specifies the rate of depreciation.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

TIP When the first argument of the FINANCE function is AMORDEGRC and the value of *basis* is 2, the function returns a missing value.

Data type DOUBLE

Example: Computing Depreciation: AMORDEGRC

The following program computes the depreciation for each accounting period by using a depreciation coefficient.

```
data test(overwrite=yes);
   dcl double cost datepurchased firstperiod salvage;
   dcl double period rate basis r;
```

```
   method init();
      cost=2400;
      datepurchased=mdy(8, 19, 2016);
      firstperiod=mdy(12, 31, 2016);
      salvage=300;
      period=1;
      rate=0.15;
      basis=1;
      r=finance('amordegrc', cost, datepurchased,
         firstperiod, salvage, period, rate, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=776
```

FINANCE AMORLINC Function

Computes the depreciation for each accounting period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('AMORLINC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, [*basis*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

date-purchased
specifies the date of the purchase of the asset.

Requirement *Date-purchased* is a SAS date.

Data type DOUBLE

first-period
specifies the date of the end of the first period.

Requirement *First-period* is a SAS date.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

period
specifies the depreciation period.

Data type DOUBLE

rate
specifies the rate of depreciation.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

TIP When the first argument of the FINANCE function is AMORLINC and the value of *basis* is 2, the function returns a missing value.

Data type DOUBLE

Example: Computing Description: AMORLINC

The following program computes the depreciation for each accounting period.

```
data test(overwrite=yes);
   dcl double cost dp fp salvage;
   dcl double period rate basis r;
   method init();
      cost=2400;
      dp=mdy(9, 30, 2016);
      fp=mdy(12, 31, 2016);
      salvage=245;
      period=0;
      rate=0.115;
      basis=0;
      r=finance('amorlinc', cost, dp, fp, salvage, period, rate,
basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=69
```

FINANCE COUPDAYBS Function

Computes the number of days from the beginning of the coupon period to the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPDAYBS', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date of the security. The security settlement date is the date after the issue date when the security is traded to the buyer.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date of the security. The maturity date is the date on which the security expires.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPDAYBS

The following program computes the number of days from the beginning of the coupon period to the settlement date.

```
data test(overwrite=yes);
   dcl double settlement maturity frequency basis r;
   method init();
      settlement=mdy(12,30,2013);
      maturity=mdy(11,29,2016);
      frequency=4;
      basis=2;
      r=finance('coupdaybs', settlement, maturity, frequency, basis);
      put r=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=31
```

Note: Dates should be entered using the DATE function, or as results of other formulas or functions. For example, use DATE(2016,5,23) for the 23rd day of May 2016. Problems can occur if dates are entered as text.

FINANCE COUPDAYS Function

Computes the number of days in the coupon period that contains the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPDAYS', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPDAYS

The following program computes the number of days in the coupon period that contains the settlement date.

```
data test(overwrite=yes);
   dcl double settlement maturity frequency basis r;
   method init();
      settlement=mdy(1,25,2015);
      maturity=mdy(11,15,2016);
      frequency=2;
      basis=1;
      r=finance('coupdays', settlement, maturity, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=181
```

FINANCE COUPDAYSNC Function

Computes the number of days from the settlement date to the next coupon date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPDAYSNC', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPDAYSNC

The following program computes the number of days from the settlement date to the next coupon date.

```
data test(overwrite=yes);
   dcl double settlement maturity frequency basis r;
   method init();
      settlement=mdy(1,25,2007);
      maturity=mdy(11,15,2008);
      frequency=2;
      basis=1;
      r=finance('coupdaysnc', settlement, maturity, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=110
```

FINANCE COUPNCD Function

Computes the next coupon date after the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPNCD', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPNCD

The following program computes the next coupon date after the settlement date.

```
data test(overwrite=yes);
   dcl double settlement maturity frequency basis r;
   method init();
      settlement=mdy(1, 25, 2007);
      maturity=mdy(11, 15, 2008);
      frequency=2;
      basis=1;
      r=finance('coupncd', settlement, maturity, frequency, basis);
      put r date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
15MAY07
```

Note: *r* is a numeric SAS value and can be printed using the DATE7 format.

FINANCE COUPNUM Function

Computes the number of coupons that are payable between the settlement date and the maturity date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPNUM', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPNUM

The following program computes the number of coupons that are payable between the settlement date and the maturity date.

```
data test(overwrite=yes);
   dcl double settlement maturity frequency basis r;
   method init();
      settlement=mdy(1,25,2015);
      maturity=mdy(11,15,2016);
      frequency=2;
      basis=1;
      r=finance('coupnum', settlement, maturity, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=4
```

FINANCE COUPPCD Function

Computes the previous coupon date before the settlement date.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('COUPPCD', *settlement*, *maturity*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: COUPPCD

The following program computes the previous coupon date before the settlement date.

```
data test(overwrite=yes);
   dcl double settlement maturity frequency basis r;
   method init();
      settlement=mdy(1, 25, 2007);
      maturity=mdy(11, 15, 2008);
      frequency=2;
      basis=1;
      r=finance('couppcd', settlement, maturity, frequency, basis);
      put r date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
15NOV06
```

FINANCE CUMIPMT Function

Computes the cumulative interest paid between two periods.

Categories:	CAS Financial
Returned data type:	DOUBLE

Syntax

FINANCE('CUMIPMT', *rate*, *nper*, *pv*, *start-period*, *end-period*, [*type*]);

Arguments

rate
specifies the interest rate.

Requirement	*Rate* is provided as a numeric value and not as a percentage.
Data type	DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

Data type DOUBLE

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

Data type DOUBLE

end-period
specifies the last period in the calculation.

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: CUMIPMT

The following program computes the cumulative interest that is paid between two periods.

```
data test(overwrite=yes);
   dcl double rate nper pv startperiod endperiod type r;
   method init();
      rate=0.09;
      nper=30;
      pv=125000;
      startperiod=13;
      endperiod=24;
      type=0;
      r=finance('cumipmt', rate, nper, pv, startperiod, endperiod,
type);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=-94054.8203251095
```

FINANCE CUMPRINC Function

Computes the cumulative principal that is paid on a loan between two periods.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('CUMPRINC', *rate*, *nper*, *pv*, *start-period*, *end-period*, [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

Data type DOUBLE

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

Data type DOUBLE

end-period
specifies the last period in the calculation.

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: CUMPRINC

The following program computes the cumulative principal that is paid on a loan between two periods.

```
data test(overwrite=yes);
   dcl double rate nper pv startperiod endperiod type r;
   method init();
      rate=0.09;
      nper=30;
      pv=125000;
      startperiod=13;
      endperiod=24;
      type=0;
      r=finance('cumprinc', rate, nper, pv, startperiod, endperiod,
type);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=-51949.7067612251
```

FINANCE DB Function

Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DB', *cost*, *salvage*, *life*, *period*, [*month*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

Data type DOUBLE

month
specifies the number of months (month is an optional numeric argument). If month is omitted, it defaults to a value of 12.

Data type DOUBLE

Example: Computing Description: DB

The following program computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

```
data test(overwrite=yes);
   dcl double cost salvage life period month r;
   method init();
      cost=1000000;
      salvage=100000;
      life=6;
      period=2;
      month=7;
      r=finance('db', cost, salvage, life, period, month);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=259639.416666666
```

FINANCE DDB Function

Computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DDB', *cost*, *salvage*, *life*, *period*, [*factor*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

Data type DOUBLE

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

Data type DOUBLE

Example: Computing Description: DDB

The following program computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.

```
data test(overwrite=yes);
   dcl double cost salvage life period factor r;
   method init();
      cost=2400;
      salvage=300;
      life=10*365;
      period=1;
      factor=.;
      r=finance('ddb', cost, salvage, life, period, factor);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=1.31506849315068
```

FINANCE DISC Function

Computes the discount rate for a security.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DISC', *settlement*, *maturity*, *price*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

price
specifies the price of security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: DISC

The following program computes the discount rate for a security.

```
data test(overwrite=yes);
   dcl double settlement maturity price redemption basis r;
   method init();
      settlement=mdy(1, 25, 2007);
      maturity=mdy(6, 15, 2007);
      price=97.975;
      redemption=100;
      basis=1;
      r=finance('disc', settlement, maturity, price, redemption,
basis);
      put r=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.05242021276595
```

FINANCE DOLLARDE Function

Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DOLLARDE', *fractional-dollar*, *fraction*);

Arguments

fractional-dollar
specifies the number expressed as a fraction.

Data type DOUBLE

fraction
specifies the whole number to use in the denominator of a fraction.

Data type DOUBLE

Example: Computing Description: DOLLARDE

The following program converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

```
data test(overwrite=yes);
   dcl double fractionaldollar fraction r;
   method init();
      fractionaldollar=1.125;
      fraction=16;
      r=finance('dollarde', fractionaldollar, fraction);
      put r=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=1.78125
```

FINANCE DOLLARFR Function

Converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DOLLARFR', *decimal-dollar*, *fraction*);

Arguments

decimal-dollar
specifies a decimal number.

Data type DOUBLE

fraction
specifies the whole number to use in the denominator of a fraction.

Data type DOUBLE

Example: Computing Description: DOLLARFR

The following program converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

```
data test(overwrite=yes);
   dcl double decimaldollar fraction r;
   method init();
      decimaldollar=1.125;
      fraction=16;
      r=finance('dollarfr', decimaldollar, fraction);
      put r=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=1.02
```

In fraction form, the value of *r* is read as $1\frac{2}{16}$.

FINANCE DURATION Function

Computes the annual duration of a security with periodic interest payments.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('DURATION', *settlement*, *maturity*, *coupon*, *yield*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

coupon
specifies the annual coupon rate of the security.

Requirement *Coupon* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: DURATION

The following program computes the annual duration of a security with periodic interest payments.

```
data test(overwrite=yes);
   dcl double settlement maturity couponrate yield
      frequency basis r;
      method init();
      settlement=mdy(1, 1, 2008);
      maturity=mdy(1, 1, 2016);
      couponrate=0.08;
      yield=0.09;
      frequency=2;
      basis=1;
      r=finance('duration', settlement, maturity, couponrate,
         yield, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=5.99377495554518
```

FINANCE EFFECT Function

Computes the effective annual interest rate.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('EFFECT', *nominal-rate*, *npery*);

Arguments

nominal-rate
specifies the nominal interest rate.

Data type DOUBLE

npery
specifies the number of compounding periods per year.

Data type DOUBLE

Example: Computing Description: EFFECT

The following program computes the effective annual interest rate.

```
data test(overwrite=yes);
   dcl double nominalrate npery r;
   method init();
      nominalrate=0.0525;
      npery=4;
      r=finance('effect', nominalrate, npery);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.05354266737075
```

FINANCE FV Function

Computes the future value of an investment.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('FV', *rate*, *nper*, [*payment*], [*present-value*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

payment
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no fees and taxes. If *payment* is omitted, you must include the *present-value* argument.

Data type DOUBLE

present-value
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *present-value* is omitted, it is assumed to be 0 (zero), and you must include the *payment* argument.

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: FV

The following program computes the future value of an investment.

```
data test(overwrite=yes);
   dcl double rate nper payment present_value type r;
   method init();
      rate=0.06/12;
      nper=10;
      payment=-200;
      present_value=-500;
      type=1;
      r=finance('fv', rate, nper, payment, present_value, type);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=2581.40337406018
```

FINANCE FVSCHEDULE Function

Computes the future value of the initial principal after applying a series of compound interest rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('FVSCHEDULE', *principal*, *schedule-1*, *schedule-2* ...);

Arguments

principal
specifies the present value.

Data type DOUBLE

schedule
specifies the sequence of interest rates to apply.

Requirement *Schedule* rates are provided as numeric values and not as percentages.

Data type DOUBLE

Example: Computing Description: FVSCHEDULE

The following program computes the future value of the initial principal after applying a series of compound interest rates.

```
data test(overwrite=yes);
   dcl double principal r1 r2 r3 r;
   method init();
      principal=1;
      r1=0.09;
      r2=0.11;
      r3=0.1;
      r=finance('fvschedule', principal, r1, r2, r3);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=1.33089
```

FINANCE INTRATE Function

Computes the interest rate for a fully invested security.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('INTRATE', *settlement*, *maturity*, *investment*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

investment
specifies the amount that is invested in the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: INTRATE

The following program computes the interest rate for a fully invested security.

```
data test(overwrite=yes);
   dcl double settlement maturity investment redemption basis r;
```

```
      method init();
         settlement=mdy(2, 15, 2008);
         maturity=mdy(5, 15, 2008);
         investment=1000000;
         redemption=1014420;
         basis=2;
         r=finance('intrate', settlement, maturity, investment,
   redemption, basis);
         put r=;
      end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
r=0.05768
```

FINANCE IPMT Function

Computes the interest payment for an investment for a specified period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('IPMT', *rate*, *period*, *nper*, *pv*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same units as *life*.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: IPMT

The following program computes the interest payment for an investment for a specified period.

```
data test(overwrite=yes);
   dcl double rate per nper pv fv type r;
   method init();
      rate=0.1/12;
      per=2;
      nper=3;
      pv=100;
      fv=.;
      type=.;
      r=finance('ipmt', rate, per, nper, pv, fv, type);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=-0.5578575637229
```

FINANCE IRR Function

Computes the internal rate of return for a series of cash flows.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('IRR', *value-1*, *value-2*, ..., *value-n*);

Arguments

value
specifies a list of numeric arguments that contain numbers for which you want to calculate the internal rate of return.

Data type DOUBLE

Example: Computing Description: IRR

The following program computes the internal rate of return for a series of cash flows.

```
data test(overwrite=yes);
   dcl double v1 v2 v3 v4 v5 v6 r;
   method init();
      v1=-70000;
      v2=12000;
      v3=15000;
      v4=18000;
      v5=21000;
      v6=26000;
      r=finance('irr', v1, v2, v3, v4, v5, v6);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.08663094803653
```

FINANCE ISPMT Function

Calculates the interest paid during a specific period of an investment.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ISPMT', *interest-rate*, *period*, *number-payments*, *pv*);

Arguments

interest-rate
specifies the interest rate for the investment.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

period
specifies the period for which you want to calculate the interest rate. *Period* must be a value between 1 and *number-payments*.

Data type DOUBLE

number-payments
specifies the number of payments for the annuity.

Data type DOUBLE

pv
specifies the loan amount or present value of the payments.

Data type DOUBLE

Example: Computing Description: ISPMT

The following program computes the interest payment for a $5,000 investment that earns 7.5% annually for two years. The interest payment is calculated for the eighth month.

```
data test(overwrite=yes);
```

```
   dcl double interest;
   method init();
      interest=finance('ispmt', 0.075/12, 8, 2*12, 5000);
      put interest=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
interest=-20.8333333333333
```

FINANCE MDURATION Function

Computes the Macaulay modified duration for a security with an assumed face value of $100.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('MDURATION', *settlement*, *maturity*, *coupon*, *yield*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

coupon
specifies the annual coupon rate of the security.

Requirement *Coupon* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: MDURATION

The following program computes the Macaulay modified duration for a security with an assumed face value of $100.

```
data test(overwrite=yes);
   dcl double settlement maturity couponrate yield frequency basis r;
   method init();
      settlement=mdy(1, 1, 2008);
      maturity=mdy(1, 1, 2016);
      couponrate=0.08;
      yield=0.09;
      frequency=2;
      basis=1;
      r=finance('mduration', settlement, maturity, couponrate, yield,
         frequency, basis);
      put r=;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log:

```
r=5.73566981391883
```

FINANCE MIRR Function

Computes the internal rate of return where positive and negative cash flows are financed at different rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('MIRR', *value-1*, ..., *value-n*, *finance-rate*, *reinvest-rate*);

Arguments

value
specifies a list of numeric arguments that contain numbers. These numbers represent a series of payments (negative values) and income (positive values) that occur at regular periods. *Value* must contain at least one positive value and one negative value to calculate the modified internal rate of return.

Data type DOUBLE

finance-rate
specifies the interest rate that you pay on the money that is used in the cash flows.

Requirement *Finance-rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

reinvest-rate
specifies the interest rate that you receive on the cash flows as you reinvest them.

Requirement *Reinvest-rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

Example: Computing Description: MIRR

The following program computes the internal rate of return where positive and negative cash flows are financed at different rates.

```
data test(overwrite=yes);
   dcl double v1 v2 v3 v4 financerate reinvestrate r;
   method init();
      v1=-1000;
      v2=3000;
      v3=4000;
      v4=5000;
      financerate=0.08;
      reinvestrate=0.10;
      r=finance('mirr', v1, v2, v3, v4, financerate, reinvestrate);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=1.35314201717001
```

FINANCE NOMINAL Function

Computes the annual nominal interest rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('NOMINAL', *effective-rate*, *npery*);

Arguments

effective-rate
specifies the effective interest rate.

Requirement *Effective-rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

npery
specifies the number of compounding periods per year.

Data type DOUBLE

Example: Computing Description: NOMINAL

The following program computes the annual nominal interest rate.

```
data test(overwrite=yes);
   dcl double effectrate npery r;
   method init();
      effectrate=0.08;
      npery=4;
      r= finance('nominal', effectrate, npery);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.07770618763309
```

FINANCE NPER Function

Computes the number of periods for an investment.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('NPER', *rate*, *payment*, *pv*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

payment
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no other fees or taxes. If *payment* is omitted, you must include the *pv* argument.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *payment* argument.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: NPER

The following program computes the number of periods for an investment.

```
data test(overwrite=yes);
   dcl double rate payment pv fv type r;
   method init();
      rate=0.08;
      payment=200;
      pv=1000;
      fv=0;
      type=0;
      r=finance('nper', rate, payment, pv, fv, type);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=-4.37198135126657
```

FINANCE NPV Function

Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('NPV', *rate*, *value-1*, [, ... *value-n*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

value
represents the sequence of the cash flows.

Data type DOUBLE

Example: Computing Description: NPV

The following program computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

```
data test(overwrite=yes);
   dcl double rate v1 v2 v3 r;
   method init();
      rate=0.08;
      v1=200;
      v2=1000;
      v3=0.;
      r=finance('npv', rate, v1, v2, v3);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=1042.52400548696
```

FINANCE ODDFPRICE Function

Computes the price of a security per $100 face value with an odd first period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDFPRICE', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *yield*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

first-coupon
specifies the first coupon date of the security.

Requirement *First-coupon* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDFPRICE

The following program computes the price of a security per $100 face value with an odd first period.

```
data test(overwrite=yes);
   dcl double settlement maturity issue firstcoupon
```

```
   rate yield redemption frequency basis r;
   method init();
      settlement=mdy(1, 15, 93);
      maturity=mdy(1, 1, 98);
      issue=mdy(1, 1, 93);
      firstcoupon=mdy(7, 1, 94);
      rate=0.07;
      yield=0.06;
      redemption=100;
      frequency=2;
      basis=0;
      r=finance('oddfprice', settlement, maturity, issue, firstcoupon,
         rate, yield, redemption, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=103.941039839766
```

FINANCE ODDFYIELD Function

Computes the yield of a security with an odd first period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDFYIELD', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *price*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

first-coupon
specifies the first coupon date of the security.

Requirement *First-coupon* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360

Numeric Value	String Value	Day Count Method
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDFYIELD

The following program computes the interest of a yield with an odd first period.

```
data test(overwrite=yes);
   dcl double settlement maturity issue firstcoupon
      rate price redemption frequency basis r;
      method init();
      settlement=mdy(1, 15, 93);
      maturity=mdy(1, 1, 98);
      issue=mdy(1, 1, 93);
      firstcoupon=mdy(7, 1, 94);
      rate=0.07;
      price=103.94103984;
      redemption=100;
      frequency=2;
      basis=0;
      r=finance('oddfyield', settlement, maturity, issue, firstcoupon,
         rate, price, redemption, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.05999999999946
```

FINANCE ODDLPRICE Function

Computes the price of a security per $100 face value with an odd last period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDLPRICE', *settlement*, *maturity*, *last-interest*, *rate*, *yield*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

last-interest
specifies the last coupon date of the security.

Requirement *Last-interest* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDLPRICE

The following program computes the price of a security per $100 face value with an odd last period.

```
data;
   dcl double settlement maturity lastinterest
      rate yield redemption frequency basis r;
   method init();
      settlement=mdy(2, 7, 2008);
      maturity=mdy(6, 15, 2008);
      lastinterest=mdy(10, 15, 2007);
      rate=0.0375;
      yield=0.0405;
      redemption=100;
      frequency=2;
      basis=0;
      r=finance('oddlprice', settlement, maturity, lastinterest,
         rate, yield, redemption, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=99.8782860147213
```

FINANCE ODDLYIELD Function

Computes the yield of a security with an odd last period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('ODDLYIELD', *settlement*, *maturity*, *last-interest*, *rate*, *price*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

last-interest
specifies the last coupon date of the security.

Requirement *Last-interest* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: ODDLYIELD

The following program computes the yield of a security with an odd last period.

```
data test(overwrite=yes);
   dcl double settlement maturity lastinterest
      rate price redemption frequency basis r;
   method init();
      settlement=mdy(2, 7, 2008);
      maturity=mdy(6, 15, 2008);
      lastinterest=mdy(10, 15, 2007);
      rate=0.0375;
      price=99.878286015;
      redemption=100;
      frequency=2;
      basis=0;
      r=finance('oddlyield', settlement, maturity, lastinterest,
         rate, price, redemption, frequency, basis);
      put r=;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.04049999999213
```

FINANCE PMT Function

Computes the periodic payment of an annuity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PMT', *rate*, *nper*, *pv*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: PMT

The following program computes the periodic payment for an annuity.

```
data test(overwrite=yes);
   dcl double rate nper pv fv type r;
   method init();
      rate=0.08;
      nper=5;
      pv=91;
      fv=3;
      type=0;
      r=finance('pmt', rate, nper, pv, fv, type);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=-23.3029067292826
```

FINANCE PPMT Function

Computes the payment on the principal for an investment for a specified period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PPMT', *rate*, *period*, *nper*, *pv*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

period
specifies the period.

Range: 1–*nper*

Data type DOUBLE

nper
specifies the number of payment periods.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: PPMT

The following program computes the payment on the principal for an investment for a specified period.

```
data test(overwrite=yes);
   dcl double rate period nper pv fv type r;
   method init();
```

```
        rate=0.08;
        period=10;
        nper=10;
        pv=200000;
        fv=0;
        type=0;
        r=finance('ppmt', rate, period, nper, pv, fv, type);
        put r=;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=-27598.0534624213
```

FINANCE PRICE Function

Computes the price of a security per $100 face value that pays periodic interest.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PRICE', *settlement*, *maturity*, *rate*, *yield*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: PRICE

The following program computes the price of a security per $100 face value that pays periodic interest.

```
data test(overwrite=yes);
   dcl double settlement maturity rate yield redemption
      frequency basis r;
   method init();
      settlement=mdy(2,15,2008);
```

```
      maturity=mdy(11,15,2017);
      rate=0.0575;
      yield=0.065;
      redemption=100;
      frequency=2;
      basis=0;
      r=finance('price', settlement, maturity, rate, yield,
         redemption, frequency, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=94.6343616213221
```

FINANCE PRICEDISC Function

Computes the price of a discounted security per $100 face value.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PRICEDISC', *settlement*, *maturity*, *discount*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: PRICEDISC

The following program computes the price of a discounted security per $100 face value.

```
data test(overwrite=yes);
   dcl double settlement maturity discount redemption basis r;
   method init();
      settlement=mdy(2, 15, 2008);
      maturity=mdy(11, 15, 2017);
      discount=0.0525;
      redemption=100;
      basis=0;
      r=finance('pricedisc', settlement, maturity, discount,
redemption,
         basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=48.8125
```

FINANCE PRICEMAT Function

Computes the price of a security per $100 face value that pays interest at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PRICEMAT', *settlement*, *maturity*, *issue*, *rate*, *yield*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

yield
specifies the annual yield of the security.

Data type DOUBLE

basis

specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: PRICEMAT

The following program computes the price of a security per $100 face value that pays interest at maturity.

```
data test(overwrite=yes);
   dcl double settlement maturity issue rate yield basis r;
   method init();
      settlement=mdy(2, 15, 2008);
      maturity=mdy(4, 13, 2008);
      issue=mdy(11, 11, 2007);
      rate=0.061;
      yield=0.061;
      basis=0;
      r=finance('pricemat', settlement, maturity, issue, rate, yield,
basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=99.9844988755569
```

FINANCE PV Function

Computes the present value of an investment.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('PV', *rate*, *nper*, *payment*, [*fv*], [*type*]);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

nper
specifies the total number of payment periods.

Data type DOUBLE

payment
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no other fees or taxes.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: PV

The following program computes the present value of an investment.

```
data test(overwrite=yes);
   dcl double rate nper payment fv type r;
   method init();
      rate=0.05;
      nper=10;
      payment=1000;
      fv=200;
      type=0;
      r=finance('pv', rate, nper, payment, fv, type);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=-7844.51757989297
```

FINANCE RATE Function

Computes the interest rate per period of an annuity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('RATE', *nper*, *payment*, *pv*, [*fv*], [*type*]);

Arguments

nper
specifies the total number of payment periods.

Data type DOUBLE

payment
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *payment* contains principal and interest but no other fees or taxes. If *payment* is omitted, you must include the *pv* argument.

Data type DOUBLE

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

Data type DOUBLE

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

Data type DOUBLE

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

If payments are due at the end of the period, then either omit the *type* argument or set it to 0. If payments are due at the beginning of the period, then set *type* to 1.

Data type DOUBLE

Example: Computing Description: RATE

The following program computes the interest rate per period of an annuity.

```
data test(overwrite=yes);
   dcl double nper payment pv r;
   method init();
      nper=4;
      payment=-2481;
      pv=8000;
      r=finance('rate', nper, payment, pv);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.09214768406875
```

FINANCE RECEIVED Function

Computes the amount that is received at maturity for a fully invested security.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('RECEIVED', *settlement*, *maturity*, *investment*, *discount*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

investment
specifies the amount that is invested in the security.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: RECEIVED

The following program computes the amount that is received at maturity for a fully invested security.

```
data test(overwrite=yes);
   dcl double settlement maturity investment discount basis r;
   method init();
      settlement=mdy(2, 15, 2008);
      maturity=mdy(5, 15, 2008);
      investment=1000000;
      discount=0.0575;
      basis=2;
      r=finance('received', settlement, maturity, investment,
discount, basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=1014584.6544071
```

FINANCE SLN Function

Computes the straight-line depreciation of an asset for one period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('SLN', *cost*, *salvage*, *life*);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

Example: Computing Description: SLN

The following program computes the straight-line depreciation of an asset for one period.

```
data test(overwrite=yes);
   dcl double cost salvage life r;
   method init();
      cost=2000;
      salvage=200;
      life=11;
      r=finance('sln', cost, salvage, life);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=163.636363636363
```

FINANCE SYD Function

Computes the sum-of-years digits depreciation of an asset for a specified period.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('SYD', *cost*, *salvage*, *life*, *period*);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

period
specifies a period in the same time units that are used for the argument *life*.

Data type DOUBLE

Example: Computing Description: SYD

The following program computes the sum-of-years digits depreciation of an asset for a specified period.

```
data test(overwrite=yes);
   dcl double cost salvage life period r;
   method init();
```

```
        cost=2000;
        salvage=200;
        life=11;
        period=1;
        r=finance('syd', cost, salvage, life, period);
        put r=;
      end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
r=300
```

FINANCE TBILLEQ Function

Computes the bond-equivalent yield for a treasury bill.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('TBILLEQ', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

Example: Computing Description: TBILLEQ

The following program computes the bond-equivalent yield for a treasury bill.

```
data test(overwrite=yes);
   dcl double settlement maturity discount r;
   method init();
      settlement=mdy(3, 31, 2008);
      maturity=mdy(6, 1, 2008);
      discount=0.0914;
      r=finance('tbilleq', settlement, maturity, discount);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.09415149356594
```

FINANCE TBILLPRICE Function

Computes the price of a treasury bill per $100 face value.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('TBILLPRICE', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

discount
specifies the discount rate of the security.

Requirement *Discount* is provided as a numeric value and not as a percentage.

Data type DOUBLE

Example: Computing Description: TBILLPRICE

The following program computes the price of a treasury bill per $100 face value.

```
data test(overwrite=yes);
   dcl double settlement maturity discount r;
   method init();
      settlement=mdy(3, 31, 2008);
      maturity=mdy(6, 1, 2008);
      discount=0.09;
      r=finance('tbillprice', settlement, maturity, discount' ');
      put r=;
   end;
enddata;
run;
```

The value of *r* that is returned is 98.45.

FINANCE TBILLYIELD Function

Computes the yield for a treasury bill.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('TBILLYIELD', *settlement*, *maturity*, *price*);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

Example: Computing Description: TBILLYIELD

The following program computes the yield for a treasury bill.

```
data test(overwrite=yes);
   dcl double settlement maturity price r;
   method init();
      settlement=mdy(3, 31, 2008);
      maturity=mdy(6, 1, 2008);
      price=98;
      r=finance('tbillyield', settlement, maturity, price);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.11849901250822
```

FINANCE VDB Function

Computes the depreciation of an asset for a specified or partial period by using a declining balance method.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('VDB', *cost*, *salvage*, *life*, *start-period*, *end-period*, [*factor*], [*noswitch*]);

Arguments

cost
specifies the initial cost of the asset.

Data type DOUBLE

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

Data type DOUBLE

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Data type DOUBLE

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

Data type DOUBLE

end-period
specifies the last period in the calculation.

Data type DOUBLE

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

Data type DOUBLE

noswitch
specifies a logical value that determines whether to switch to straight-line depreciation when the depreciation is greater than the declining balance calculation. If *noswitch* is omitted, it is assumed to be 1.

Data type DOUBLE

Example: Computing Description: VDB

The following program computes the depreciation of an asset for a specified or partial period by using a declining balance method.

```
data test(overwrite=yes);
   dcl double cost salvage life startperiod endperiod factor r;
   method init();
      cost=2400;
      salvage=300;
      life=10;
      startperiod=0;
      endperiod=1;
      factor=1.5;
      r=finance('vdb', cost, salvage, life, startperiod, endperiod,
factor);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=360
```

FINANCE XIRR Function

Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('XIRR', *values*, *dates*, [*guess*]);

Arguments

values
specifies a series of cash flows that corresponds to a schedule of payments in dates. The first payment is optional and corresponds to a cost or payment that occurs at the beginning of the investment. If the first value is a cost or payment, it must be a negative value. All succeeding payments are discounted based on a 365-day year. The series of values must contain at least one positive value and one negative value.

Data type DOUBLE

dates
specifies a schedule of payment dates that corresponds to the cash flow payments. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they can occur in any order.

Requirement *Dates* are SAS dates.

Data type DOUBLE

guess
specifies an optional number that you guess is close to the result of XIRR.

Data type DOUBLE

Example: Computing Description: XIRR

The following program computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

```
data test(overwrite=yes);
   dcl double v1 v2 v3 v4 v5 d1 d2 d3 d4 d5 r;
   method init();
      v1=-10000; d1=mdy(1, 1, 2008);
      v2=2750; d2=mdy(3, 1, 2008);
      v3=4250; d3=mdy(10, 30, 2008);
      v4=3250; d4=mdy(2, 15, 2009);
      v5=2750; d5=mdy(4, 1, 2009);
      r=finance('xirr', v1, v2, v3, v4, v5, d1, d2, d3, d4, d5, 0.1);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.37336253351883
```

FINANCE XNPV Function

Computes the net present value for a schedule of cash flows that is not necessarily periodic.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('XNPV', *rate*, *values*, *dates*);

Arguments

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

values
specifies a series of cash flows that corresponds to a schedule of payments in dates. The first payment is optional and corresponds to a cost or payment that occurs at the beginning of the investment. If the first value is a cost or payment, it must be a negative value. All succeeding payments are discounted based on a 365-day year. The series of values must contain at least one positive value and one negative value.

Data type DOUBLE

dates
specifies a schedule of payment dates that corresponds to the cash flow payments. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they can occur in any order.

Requirement *Dates* are SAS dates.

Data type DOUBLE

Example: Computing Description: XNPV

The following program computes the net present value for a schedule of cash flows that is not necessarily periodic.

```
data test(overwrite=yes);
   dcl double rate v1 v2 v3 v4 v5 d1 d2 d3 d4 d5 r;
   method init();
      rate=.09;
      v1=-10000; d1=mdy(1, 1, 2008);
      v2=2750; d2=mdy(3, 1, 2008);
      v3=4250; d3=mdy(10, 30, 2008);
      v4=3250; d4=mdy(2, 15, 2009);
      v5=2750; d5=mdy(4, 1, 2009);
```

```
        r=finance('xnpv', rate, v1, v2, v3, v4, v5, d1, d2, d3, d4, d5);
        put r=;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=2086.64760203153
```

FINANCE YIELD Function

Computes the yield on a security that pays periodic interest.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('YIELD', *settlement*, *maturity*, *rate*, *price*, *redemption*, *frequency*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: YIELD

The following program computes the yield on a security that pays periodic interest.

```
data test(overwrite=yes);
   dcl double settlement maturity rate pr redemption frequency basis r;
   method init();
      settlement=mdy(2, 15, 2008);
      maturity=mdy(11, 15, 2016);
      rate=0.0575;
      pr=95.04287;
      redemption=100;
      frequency=2;
      basis=0;
      r=finance('yield', settlement, maturity, rate, pr, redemption,
         frequency, basis);
      put r=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.06500000688075
```

FINANCE YIELDDISC Function

Computes the annual yield for a discounted security (for example, a treasury bill).

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('YIELDDISC', *settlement*, *maturity*, *price*, *redemption*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

redemption
specifies the amount to be received at maturity.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: YIELDDISC

The following program computes the annual yield for a discounted security (for example, a treasury bill).

```
data test(overwrite=yes);
   dcl double settlement maturity pr redemption basis r;
   method init();
      settlement=mdy(2, 15, 2008);
      maturity=mdy(11, 15, 2016);
      pr=95.04287;
      redemption=100;
      basis=0;
      r=finance('yielddisc', settlement, maturity, pr, redemption,
basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.0059607747836
```

FINANCE YIELDMAT Function

Computes the annual yield of a security that pays interest at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

FINANCE('YIELDMAT', *settlement*, *maturity*, *issue*, *rate*, *price*, [*basis*]);

Arguments

settlement
specifies the settlement date.

Requirement *Settlement* is a SAS date.

Data type DOUBLE

maturity
specifies the maturity date.

Requirement *Maturity* is a SAS date.

Data type DOUBLE

issue
specifies the issue date of the security.

Requirement *Issue* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate.

Requirement *Rate* is provided as a numeric value and not as a percentage.

Data type DOUBLE

price
specifies the price of the security per $100 face value.

Data type DOUBLE

basis
specifies an optional parameter as a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360

Numeric Value	String Value	Day Count Method
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

Data type DOUBLE

Example: Computing Description: YIELDMAT

The following program computes the annual yield of a security that pays interest at maturity.

```
data test(overwrite=yes);
   dcl double settlement maturity issue rate pr basis r;
   method init();
      settlement=mdy(3, 15, 2008);
      maturity=mdy(11, 3, 2008);
      issue=mdy(11, 8, 2007);
      rate=0.0625;
      pr=100.0123;
      basis=0;
      r=finance('yieldmat', settlement, maturity, issue, rate, pr,
basis);
      put r=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
r=0.06095433369153
```

FIND Function

Searches for a specific substring of characters within a character string.

Categories: CAS
Character

Returned data type: DOUBLE

Tip: Use the "KINDEX Function" in *SAS National Language Support (NLS): Reference Guide* instead to write encoding independent code.

Syntax

FIND(*string*, *substring*[, *modifier(s)*][, *startpos*])

FIND(*string*, *substring*[, *startpos*][, *modifier(s)*])

Arguments

string

specifies a character constant, variable, or expression that evaluates or can be coerced to a character string and that will be searched for substrings.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

substring

is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the substring of characters to search for in *string*.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

modifier(s)

is a character constant, variable, or expression that specifies one or more modifiers. The following characters, in uppercase or lowercase, can be used as modifiers:

i or I

ignores character case during the search. If this modifier is not specified, FIND searches only for character substrings with the same case as the characters in *substring*.

t or T

trims trailing blanks from *string* and *substring*.

Note: If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the FIND function with the T modifier.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: If *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

startpos
is a numeric constant, variable, or expression which is a whole number that specifies the position at which the search should start and the direction of the search.

Data type DOUBLE

Details

The FIND function searches *string* for the first occurrence of the specified *substring*, and returns the position of that substring. If the substring is not found in *string*, FIND returns a value of 0.

If *startpos* is not specified, FIND starts the search at the beginning of the *string* and searches the *string* from left to right. If *startpos* is specified, the absolute value of *startpos* determines the position at which to start the search. The sign of *startpos* determines the direction of the search.

Value of *startpos*	**Action**
greater than 0	starts the search at position *startpos* and the direction of the search is to the right. If *startpos* is greater than the length of *string*, FIND returns a value of 0.
less than 0	starts the search at position –*startpos* and the direction of the search is to the left. If –*startpos* is greater than the length of *string*, the search starts at the end of *string*.
equal to 0	returns a value of 0.

Comparisons

- The FIND function searches for substrings of characters in a character string, whereas the FINDC function searches for individual characters in a character string.
- The FIND function and the INDEX function both search for substrings of characters in a character string. However, the INDEX function does not have the *modifiers* nor the *startpos* arguments.

Example

The following program illustrates the FIND function using different data types:

```
data _null_;
   dcl char(35) char1 char2 char3;
```

```
      dcl varchar(35) varchar1 varchar2 varchar3;
      dcl double i;
      method run();
         i=find('She sells seashells? Yes, she does.', 'she ', 'i');
         put 'literal' i=;
         char1='She sells seashells? Yes, she does.';
         char2='she ';
         char3='i';
         i=find(char1, char2, char3);
         put 'char' i=;
         varchar1='She sells seashells? Yes, she does.';
         varchar2='she ';
         varchar3='i';
         i=find(varchar1, varchar2, varchar3);
         put 'varchar' i=;
         char1='She sells seashells? Yes, she does.';
         char2='she ';
         char3='i';
         i=find(char1, trim(char2), char3);
         put 'trim char' i=;
      end;
   enddata;
   run;
```

SAS writes the following output to the log.

```
literal i=1
char i=0
varchar i=1
trim char i=1
```

See Also

Functions:

- “COUNT Function” on page 478
- “FINDC Function” on page 628
- “FINDW Function” on page 637
- “INDEX Function” on page 685

FINDC Function

Searches a string for any character in a list of characters.

Categories:	CAS Character
Returned data type:	DOUBLE

Tip: Use the "KINDEXC Function" in *SAS National Language Support (NLS): Reference Guide* instead to write encoding independent code.

Syntax

FINDC(*string*[, *charlist*])

FINDC(*string*, *charlist*[, *modifier*])

FINDC(*string*, *charlist*, *modifier*[, *startpos*])

FINDC(*string*, *charlist*[, *startpos*][, *modifier*])

Arguments

string
is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the character string to be searched.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

charlist
is a constant, variable, or character expression that initializes a list of characters. FINDC searches for the characters in this list provided that you do not specify the K modifier in the *modifier* argument. If you specify the K modifier, FINDC searches for all characters that are not in this list of characters. You can add more characters to the list by using other modifiers.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

modifier
is a character constant, variable, or expression in which each character modifies the action of the FINDC function. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	searches from right to left, instead of from left to right, regardless of the sign of the *startpos* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.

h or H adds a horizontal tab to the list of characters.

i or I ignores character case during the search.

k or K searches for any character that does not appear in the list of characters. If you do not specify this modifier, then FINDC searches for any character that appears in the list of characters. The V and K modifiers perform the same function.

l or L adds lowercase letters to the list of characters.

n or N adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O processes the *charlist* and the *modifier* arguments only once, rather than every time the FINDC function is called. Using the O modifier in DS2 (excluding WHERE clauses) can make FINDC run faster when you call it in a loop where the *charlist* and the *modifier* arguments do not change.

p or P adds punctuation marks to the list of characters.

s or S adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T trims trailing blanks from the *string* and *charlist* arguments. Note that if you want to remove trailing blanks from just one character argument instead of both (or all) character arguments, use the TRIM function instead of the FINDC function with the T modifier.

u or U adds uppercase letters to the list of characters.

v or V causes all character that are not in the list of characters to be treated as delimiters. If V is not specified, then all characters that are in the list of characters are treated as delimiters. The V and K modifiers perform the same function.

w or W adds printable characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip If *modifier* is a constant, then enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

startpos

is an optional numeric constant, variable, or expression which is a whole number that specifies the position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The FINDC function searches *string* for the first occurrence of the specified characters, and returns the position of the first character found. If no characters are found in *string*, then FINDC returns a value of 0.

The FINDC function allows character arguments to be null. Null arguments are treated as character strings that have a length of zero. Numeric arguments cannot be null.

If *startpos* is not specified, FINDC begins the search at the end of the string if you use the B modifier, or at the beginning of the string if you do not use the B modifier.

If *startpos* is specified, the absolute value of *startpos* specifies the position at which to begin the search. If you use the B modifier, the search always proceeds from right to left. If you do not use the B modifier, the sign of *startpos* specifies the direction in which to search. The following table summarizes the search directions:

Value of *startpos*	**Action**
greater than 0	search begins at position *startpos* and proceeds to the right. If *startpos* is greater than the length of the string, FINDC returns a value of 0.
less than 0	search begins at position –*startpos* and proceeds to the left. If *startpos* is less than the negative of the length of the string, the search begins at the end of the string.
equal to 0	returns a value of 0.

Comparisons

- The FINDC function searches for individual characters in a character string, whereas the FIND function searches for substrings of characters in a character string.
- The FINDC function and the INDEXC function both search for individual characters in a character string. However, the INDEXC function does not have the *modifier* nor the *startpos* arguments.
- The FINDC function searches for individual characters in a character string, whereas the VERIFY function searches for the first character that is unique to an expression. The VERIFY function does not have the *modifier* nor the *startpos* arguments.

Examples

Example 1: Searching for Characters in a String

The following program searches a character string and returns the characters that are found.

```
data test (overwrite=yes);
   dcl double j i;
   dcl char c;
   method run();
      j=0;
      do until(j=0);
         j = findc('Hi, ho!','hi',j+1);
         if j= 0 then put 'The End';
         else do;
            c = substr('Hi, ho!', j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=2 c=i
j=5 c=h
The End
```

Example 2: Searching for Characters in a String and Ignoring Case

The following program searches a character string and returns the characters that are found. The I modifier is used to ignore the case of the characters.

```
data test (overwrite=yes);
   dcl char string charlist c;
   dcl double j i;
   method run();
      string='Hi, ho!';
      charlist='ho';
      j=0;
      do until(j=0);
         j=findc(string,charlist,j+1,'i');
         if j=0 then put 'The End';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=H
j=5 c=h
j=6 c=o
The End
```

Example 3: Searching for Characters and Using the K Modifier

The following program searches a character string and returns the characters that do not appear in the character list.

```
data test (overwrite=yes);
   dcl double j;
   dcl char c string charlist;
   method run();
      string='Hi, ho!';
      charlist='hi';
      j=0;
      do until(j = 0);
      j = findc(string,charlist,'k',j+1);
      if j=0 then put 'The End';
         else do;
            c = substr(string,j,1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=H
j=3 c=,
j=4 c=
j=6 c=o
j=7 c=!
The End
```

Example 4: Searching for the Characters h, i, and Blank

The following program searches for the three characters h, i, and blank. The characters h and i are in lowercase. The uppercase characters H and I are ignored in this search.

```
data test (overwrite=yes);
   dcl double whereishi;
   dcl char whatfound;
   method run();
      whereishi=0;
      do until(whereishi=0);
         whereishi=findc('Hi there, Ian!','hi ',whereishi+1);
         if whereishi=0 then put 'The End';
         else do;
            whatfound=substr('Hi there, Ian!',whereishi,1);
            put whereishi= whatfound=;
         end;
     end;
```

```
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
whereishi=2 whatfound=i
whereishi=3 whatfound=
whereishi=5 whatfound=h
whereishi=10 whatfound=
The End
```

Example 5: Searching for the Characters h and i While Ignoring Case

The following program searches for the four characters h, i, H, and I. FINDC with the i modifier ignores character case during the search.

```
data test (overwrite=yes);
   dcl double whereishi_i;
   dcl char whatfound variable1 variable2 variable3;
   method run();
      whereishi=0;
      do until(whereishi=0);
         whereishi=findc('Hi there, Ian!','hi ',whereishi+1);
         if whereishi=0 then put 'The End';
            else do;
                whatfound=substr('Hi there, Ian!',whereishi,1);
                put whereishi= whatfound=;
            end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
whereishi_i=1 whatfound=H
whereishi_i=2 whatfound=i
whereishi_i=5 whatfound=h
whereishi_i=11 whatfound=I
The End
```

Example 6: Searching for the Characters h and i with Trailing Blanks Trimmed

The following program searches for the two characters h and i. FINDC with the t modifier trims trailing blanks from the string argument and the characters argument.

```
data test (overwrite=yes);
dcl char(14) expression1 expression2 expression3 whatfound;
dcl double whereishi_t;
method run();
   whereishi_t=0;
      do until(whereishi_t=0);
         expression1='Hi there, '||'Ian!';
```

```
         expression2=kscan('bye or hi',3)||'  ';
         expression3=trim('t   ');

whereishi_t=findc(expression1,expression2,expression3,whereishi_t+1);
         if whereishi_t=0 then put 'The End';
            else do;
               whatfound=substr(expression1,whereishi_t,1);
               put whereishi_t= whatfound=;
            end;
      end;
   end;
enddata;
run;
```

SAS writes the following lines output to the log:

```
whereishi_t=2 whatfound=i
whereishi_t=5 whatfound=h
The End
```

Example 7: Searching for All Characters, Excluding h, i, H, and I

The following program searches for all of the characters in the string, excluding the characters h, i, H, and I. FINDC with the v modifier counts only the characters that do not appear in the characters argument. This example also includes the i modifier and therefore ignores character case during the search.

```
data test (overwrite=yes);
   dcl char(14) xyz;
   dcl double whereishi_iv;
   method run();
      whereishi_iv=0;
      do until(whereishi_iv=0);
         xyz='Hi there, Ian!';
         whereishi_iv=findc(xyz,'hi',whereishi_iv+1,'iv');
         if whereishi_iv=0 then put 'The End';
         else do;
            whatfound=substr(xyz,whereishi_iv,1);
            put whereishi_iv= whatfound=;
         end;
      end;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
whereishi_iv=3 whatfound=
whereishi_iv=4 whatfound=t
whereishi_iv=6 whatfound=e
whereishi_iv=7 whatfound=r
whereishi_iv=8 whatfound=e
whereishi_iv=9 whatfound=,
whereishi_iv=10 whatfound=
whereishi_iv=12 whatfound=a
whereishi_iv=13 whatfound=n
whereishi_iv=14 whatfound=!
The End
```

See Also

Functions:

- “ANYALNUM Function” on page 292
- “ANYALPHA Function” on page 295
- “ANYCNTRL Function” on page 298
- “ANYDIGIT Function” on page 300
- “ANYGRAPH Function” on page 304
- “ANYLOWER Function” on page 307
- “ANYPRINT Function” on page 311
- “ANYPUNCT Function” on page 314
- “ANYSPACE Function” on page 317
- “ANYUPPER Function” on page 319
- “ANYXDIGIT Function” on page 321
- “COUNTC Function” on page 481
- “INDEXC Function” on page 687
- “NOTALNUM Function” on page 845
- “NOTALPHA Function” on page 848
- “NOTCNTRL Function” on page 850
- “NOTDIGIT Function” on page 852
- “NOTGRAPH Function” on page 857
- “NOTLOWER Function” on page 860
- “NOTPRINT Function” on page 864
- “NOTPUNCT Function” on page 866
- “NOTSPACE Function” on page 869
- “NOTUPPER Function” on page 872
- “NOTXDIGIT Function” on page 874
- “VERIFY Function” on page 1164

FINDW Function

Returns the character position of a word in a string, or returns the number of the word in a string.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

FINDW(*string*, *word*[, *chars*])

FINDW(*string*, *word*, *chars*, *modifier(s)*[, *startpos*])

FINDW(*string*, *word*, *chars*, *startpos*[, *modifier(s)*])

FINDW(*string*, *word*, *startpos*[, *chars*[, *modifier(s)*]])

Arguments

string
is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the character string to be searched.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

word
is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the word to be searched.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

chars
is an optional character constant, variable, or expression that initializes a list of characters.

The characters in this list are the delimiters that separate words, provided that you do not specify the K modifier in the *modifier* argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to this list by using other modifiers.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

startpos
is an optional numeric constant, variable, or expression which is a whole number that specifies the position at which the search should begin and the direction in which to search.

Data type DOUBLE

'modifier(s)'
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the FINDW function.

You can use the following characters as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	searches from right to left, instead of from left to right, regardless of the sign of the *startpos* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
e or E	counts the words that are scanned until the specified word is found, instead of determining the character position of the specified word in the string. Fragments of words are not counted.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters.
k or K	causes all character that are not in the list of characters to be treated as delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters. The K and V modifiers perform the same function.
l or L	adds lowercase letters to the list of characters.
m or M	specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *chars* and the *modifier* arguments only once, rather than every time the FINDW function is called. Using the O modifier in DS2 (excluding WHERE clauses) can make FINDW run faster when you call it in a loop where the *chars* and the *modifier* arguments do not change.
p or P	adds punctuation marks to the list of characters.

q or Q	ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.
r or R	removes leading and trailing delimiters from the *word* argument.
s or S	adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.
t or T	trims trailing blanks from the *string*, *word*, and *chars* arguments.
u or U	adds uppercase letters to the list of characters.
v or V	causes all character that are not in the list of characters to be treated as delimiters. If V is not specified, then all characters that are in the list of characters are treated as delimiters. The V and K modifiers perform the same function.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.
Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
Tip	If you use the *modifier* argument, then it must be positioned after the *chars* argument.

Details

Definition of "Delimiter"

"Delimiter" refers to any of several characters that are used to separate words. You can specify the delimiters by using the *chars* argument, the *modifier* argument, or both. If you specify the Q modifier, then the characters inside substrings that are enclosed in quotation marks are not treated as delimiters.

Definition of "Word"

"Word" refers to a substring that has both of the following characteristics:

- bounded on the left by a delimiter or the beginning of the string
- bounded on the right by a delimiter or the end of the string

Note: A word can contain delimiters. In this case, the FINDW function differs from the SCAN function, in which words are defined as not containing delimiters.

Searching for a String

If the FINDW function fails to find a substring that both matches the specified word and satisfies the definition of a word, then FINDW returns a value of 0.

If the FINDW function finds a substring that both matches the specified word and satisfies the definition of a word, the value that is returned by FINDW depends on whether the E modifier is specified:

- If you specify the E modifier, then FINDW returns the number of complete words that were scanned while searching for the specified word. If *startpos* specifies a position in the middle of a word, then that word is not counted.
- If you do not specify the E modifier, then FINDW returns the character position of the substring that is found.

If you specify the *startpos* argument, then the absolute value of *startpos* specifies the position at which to begin the search. The sign of *startpos* specifies the direction in which to search:

Value of *startpos*	**Action**
greater than 0	search begins at position *startpos* and proceeds to the right. If *startpos* is greater than the length of the string, then FINDW returns a value of 0.
less than 0	search begins at position –*startpos* and proceeds to the left. If *startpos* is less than the negative of the length of the string, then the search begins at the end of the string.
equal to 0	FINDW returns a value of 0.

If you do not specify the *startpos* argument or the B modifier, then FINDW searches from left to right starting at the beginning of the string. If you specify the B modifier, but do not use the *startpos* argument, then FINDW searches from right to left starting at the end of the string.

Using the FINDW Function in ASCII and EBCDIC Environments

If you use the FINDW function with only two arguments, the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the FINDW function uses the ~ character instead.
- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

Using Null Arguments

The FINDW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Examples

Example 1: Searching a Character String for a Word

The following program searches a character string for the word “she”, and returns the position of the beginning of the word.

```
data _null_;
   dcl double whereisshe;
   method run();
      whereisshe=findw('She sells sea shells? Yes, she does.','she');
      put whereisshe=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
whereisshe=28
```

Example 2: Searching a Character String and Using the Chars and Startpos Arguments

The following program contains two occurrences of the word “rain.” Only the second occurrence is found by FINDW because the search begins in position 25. The *chars* argument specifies a space as the delimiter.

```
data _null_;
   dcl double result;
   method run();
      result = findw('At least 2.5 meters of rain falls in a rain
forest.',
         'rain', ' ', 25);
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=40
```

Example 3: Searching a Character String and Using the I Modifier and the Startpos Argument

The following program uses the I modifier and returns the position of the beginning of the word. The I modifier disregards case, and the *startpos* argument identifies the starting position from which to search.

```
data _null_;
   dcl double result;
   dcl char(75) string;
   method run();
      string='Artists from around the country display their art at
         an art festival.';
      result=findw(string, 'Art',' ', 'i', 10);
      put result=;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
result=47
```

Example 4: Searching a Character String and Using the E Modifier

The following program uses the E modifier and returns the number of complete words that are scanned while searching for the word "art."

```
data _null_;
   dcl double result;
   dcl char(75) string;
   method run();
      string='Artists from around the country display their art at
         an art festival.';
      result=findw(string,'art',' ','E');
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=8
```

Example 5: Searching a Character String and Using the E Modifier and the Startpos Argument

The following program uses the E modifier to count words in a character string. The word count begins at position 50 in the string. The result is 3 because "art" is the third word after the 50th character position.

```
data _null_;
   dcl double result;
```

```
   dcl char(75) string;
   method run();
      string='Artists from around the country display their art at
         an art festival.';
      result=findw(string, 'art',' ','E',50);
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=3
```

Example 6: Searching a Character String and Using Two Modifiers

The following program uses the I and the E modifiers to find a word in a string.

```
data _null_;
   dcl double result;
   dcl char(130) string;
   method run();
      string='The Great Himalayan National Park was created in 1984.
Because
         of its terrain and altitude, the park supports a diversity
         of wildlife and vegetation.';
      result=findw(string,'park',' ','I E');
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=5
```

Example 7: Searching a Character String and Using the R Modifier

The following program uses the R modifier to remove leading and trailing delimiters from a word.

```
data _null_;
   dcl double result;
   dcl char(75) string;
   dcl char word;
   method run();
      string='Artists from around the country display their art at
         an art festival.';
      word='  art  ';
      result=findw(string, word, ' ', 'R');
      put result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
result=47
```

See Also

Functions:

- “COUNTW Function” on page 484
- “FIND Function” on page 625
- “FINDC Function” on page 628
- “INDEXW Function” on page 689
- “SCAN Function” on page 1055

FLOOR Function

Returns the largest integer less than or equal to a numeric value expression.

Categories: CAS
Truncation

Returned data type: DECIMAL, DOUBLE, NUMERIC

Syntax

FLOOR(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DECIMAL, DOUBLE, NUMERIC

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

If *expression* is within 1E-12 of an integer, the function returns that integer. If the result is a number that does not fit into the range of a DOUBLE, the FLOOR function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

The FLOOR function fuzzes the results so that if the results are within 1E-12 of an integer, the FLOOR function returns that integer. The FLOORZ function uses zero fuzzing. Therefore, with the FLOORZ function, you might get unexpected results.

Examples

Example 1: FLOOR Function Example

The following program illustrates the FLOOR function:

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=floor(1.95);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1
```

Example 2: FLOOR Function with Matrix Package Example

The following program illustrates the FLOOR function:

```
data _null_;
   dcl double a[2, 2];
   dcl double c[2, 2];

   method run();
      dcl package matrix m;
      dcl package matrix r;
      dcl double i j;

      a := (1323.43, -.72, 3.38, 45);

      m=_new_ matrix(a, 2, 2);
```

```
            r=m.floor();
            r.toarray(c);

            do i=1 to 2;
               do j=1 to 2;
                  put c[i, j];
               end;
            end;
         end;
      enddata;
      run;
      quit;
```

SAS writes the following output to the log.

```
1323
0
3
45
```

See Also

Functions:

- “CEIL Function” on page 419
- “CEILZ Function” on page 421
- “FLOORZ Function” on page 646

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

FLOORZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Comparisons

Unlike the FLOOR function, the FLOORZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function, you might get unexpected results.

Example

The following program illustrates the FLOORZ function:

```
data test(overwrite=yes);
   dcl double a b c d e f var1 var2 var6;
   method run();
      var1=2.1;
      a=floorz(var1);
      put a=;

      var2=-2.4;
      b=floorz(var2);
      put b=;

      c=floorz(-1.6);
      put c=;

      var6=(1.-1.e-13);
      d=floorz(1-1.e-13);
      put d=;

      e=floorz(763);
      put e=;

      f=floorz(-223.456);
      put f=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=2
b=-3
c=-2
d=0
e=763
f=-224
```

See Also

Functions:

- “CEIL Function” on page 419
- “CEILZ Function” on page 421
- “FLOOR Function” on page 644

FMTINFO Function

Returns information about a SAS format or informat.

Categories: CAS
Special

Restriction: This function returns information about formats that are supplied by SAS. It cannot be used for user-defined formats that are created with the FORMAT procedure.

Syntax

FMTINFO('*format-name*', '*information-type*');

Arguments

'format-name'
specifies the name of a SAS format or informat.

Requirement *format-name* must be enclosed in single quotation marks.

information-type
specifies the type of information that is returned. *format-information* can be one of the following values:

'CAT'
returns the function category.

See For a complete list, see “Function Categories” on page 244.

'TYPE'
returns whether the *format-name* is a format, an informat, or both.

'DESC'
returns a short description of the format or informat.

'MIND'
returns the minimum number of digits to the right of the decimal place in the format or informat.

'MAXD'
returns the maximum number of digits to the right of the decimal place in the format or informat.

'DEFD'
returns the default number of digits to the right of the decimal place in the format or informat.

'MINW'
returns the minimum width value of the format or informat.

'MAXW'
returns the maximum width value of the format or informat.

'DEFW'
returns the default width value of the format or informat.

Restriction You can specify only one *information-type* argument.

Requirement *information-type* must be enclosed in single quotation marks.

Details

The FMTINFO function returns information about a format or informat. You can return information about a format or informat's category, the type of language element, a description of the language element, and the minimum, maximum, and default decimal and width values.

You cannot specify multiple arguments with the FMTINFO function.

The FMTINFO function returns a character string for all data values, including the numeric value arguments MIND, MAXD, DEFD, MINW, MAXW, and DEFW.

Example

The following program returns information about the DATE format.

```
data _null_;
   dcl char(30) fdesc fcat ftype;
   dcl double fmind fmaxd fdefd fminw fmaxw fdefw;
   method run();
      ftype=fmtinfo('date','type');
      fcat= fmtinfo('date','cat');
      fdesc= fmtinfo('date','desc');
      fmind= fmtinfo('date','mind');
      fmaxd= fmtinfo('date','maxd');
      fdefd= fmtinfo('date','defd');
      fminw= fmtinfo('date','minw');
      fmaxw= fmtinfo('date','maxw');
      fdefw= fmtinfo('date','defw');
      put ftype=;
      put fcat=;
```

```
      put fdesc= ;
      put fmind= ;
      put fmaxd= ;
      put fdefd= ;
      put fminw= ;
      put fmaxw= ;
      put fdefw= ;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
ftype=BOTH
fcat=date
fdesc=date value
fmind=0
fmaxd=8
fdefd=0
fminw=5
fmaxw=11
fdefw=7
```

FNONCT Function

Returns the value of the noncentrality parameter of an F distribution.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

FNONCT(*x*, *ndf*, *ddf*, *probability*)

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

ndf
is a numeric numerator degree of freedom parameter.

Range $ndf > 0$

Data type DOUBLE

ddf
is a numeric denominator degree of freedom parameter.

Range *ddf* > 0

Data type DOUBLE

probability
is a probability.

Range 0 < *probability* < 1

Data type DOUBLE

Details

The FNONCT function returns the nonnegative noncentrality parameter from a noncentral *F* distribution whose parameters are *x*, *ndf*, *ddf*, and *nc*. If *probability* is greater than the probability from the central *F* distribution whose parameters are *x*, *ndf*, and *ddf*, a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root *nc* of the equation

$$P_f(x|ndf, ddf, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_f(x|ndf, ddf, nc) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} I_{\frac{(ndf)x}{ddf+(ndf)x}}\left(\frac{ddf}{2}+j, \frac{ddf}{2}\right)$$

In the equation, *I* (. . .) is the probability from the beta distribution that is given by the following equation:

$$I_x(a,b) = \frac{}{\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}} \int_0^x t^{a-1}(1-t)^{b-1}dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

```
proc ds2;
data work /overwrite=yes;
   dcl double x df ddf nc prob ncc;
   method init();
      x=2;
      df=4;
      ddf=5;
```

```
      do nc=1 to 3 by .5;
         prob=probf(x, df, ddf, nc);
         ncc=fnonct(x, df, ddf, prob);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=work;
run;
quit;
```

Output 7.9 *Output from the FNONCT Function*

The SAS System

Obs	x	df	ddf	nc	prob	ncc
1	2	4	5	1.0	0.69277	1.00000
2	2	4	5	1.5	0.65701	1.50000
3	2	4	5	2.0	0.62232	2.00000
4	2	4	5	2.5	0.58878	2.50000
5	2	4	5	3.0	0.55642	3.00000

See Also

Functions:

- “CNONCT Function” on page 430
- “TNONCT Function” on page 1131

FUZZ Function

Returns the nearest whole number if the argument is within 1E-12 of that number.

Categories:	CAS Truncation
Returned data type:	DOUBLE

Syntax

FUZZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The FUZZ function returns the nearest whole number if the expression is within 1E-12 of the number (that is, if the absolute difference between the whole number and argument is less than 1E-12). Otherwise, the expression is returned.

Example

The following programs illustrate the FUZZ function:

```
/**** returns nearest whole number ******/
data test (overwrite=yes);
   dcl double var1 x;
   method run();
      var1=5.9999999999999;
      x=put(fuzz(var1),16.14);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
6
```

```
/**** returns the original expression ******/
data test (overwrite=yes);
   dcl double var1 x;
   method run();
      var1=5.99999999;
      x=put(fuzz(var1),16.14);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
5.99999999
```

GAMINV Function

Returns a quantile from the gamma distribution.

Categories: CAS
Quantile

Returned data type: DOUBLE

Syntax

GAMINV(*p*, *a*)

Arguments

p

specifies any valid expression that evaluates to a numeric probability.

Range $0 \le p < 1$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

a

specifies any valid expression that evaluates to a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The GAMINV function returns the pth quantile from the gamma distribution, with shape parameter *a*. The probability that a row from a gamma distribution is less than or equal to the returned quantile is *p*.

Note: GAMINV is the inverse of the PROBGAM function.

Example

The following program illustrates the GAMINV function:

```
data test(overwrite=yes);
  dcl double x y;
  method run();
      x=gaminv(0.5, 9);
      y=gaminv(.1, 2.1);
      put x;
      put y;
  end;
enddata;
run;
```

SAS writes the following output to the log.

```
8.66895118437038
0.5841932368992
```

See Also

Functions:

- "PROBGAM Function" on page 957

GAMMA Function

Returns the value of the gamma function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

GAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction Nonpositive integers are invalid.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The GAMMA function returns the integral, which is given by the following equation.

$$\text{GAMMA}(x) = \int_0^\infty t^{x-1} e^{-t}\, dt.$$

For positive integers, GAMMA(x) is (x – 1)!. This function is commonly denoted by $\Gamma(x)$.

Example

The following program illustrates the GAMMA function:

```
data _null_;
   dcl double x;
   method run();
      x=gamma(6);
      put x=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
120
```

GARKHCLPRC Function

Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

GARKHCLPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

R_d

is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period *t*.

Requirement Specify a value for R_d for the same time period as the unit of *t*.

Data type DOUBLE

R_f

is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period *t*.

Requirement Specify a value for R_f for the same time period as the unit of *t*.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

Data type DOUBLE

Details

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

$$\text{CALL} = SN(d_1)\left(\varepsilon^{-R_f t}\right) - EN(d_2)\left(\varepsilon^{-R_d t}\right)$$

Arguments

S
specifies the spot currency price.

N
specifies the cumulative normal density function.

E
specifies the exercise price of the option.

t
specifies the time to expiration.

R_d
specifies the risk-free domestic interest rate for period *t*.

R_f
specifies the risk-free foreign interest rate for period *t*.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(R_d - R_f + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{CALL} = \max((S - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

Example

The following program illustrates the GARKHCLPRC function:

```
data test(overwrite=yes);
   dcl double a b;
   method run();
      a=garkhclprc(40, .5, 38, .06, .04, .2);
      b=garkhclprc(19, .25, 20, .05, .03, .09);
      put a=;
      put b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1.44942510595479
1.1304209447635
```

See Also

Functions:

- "GARKHPTPRC Function" on page 659

GARKHPTPRC Function

Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

GARKHPTPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

R_d

is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period *t*.

Requirement Specify a value for R_d for the same time period as the unit of *t*.

Data type DOUBLE

R_f

is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period *t*.

Requirement Specify a value for R_t for the same time period as the unit of *t*.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Data type DOUBLE

Details

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} - S\left(\varepsilon^{-R_f t}\right) + E\left(\varepsilon^{-R_d t}\right)$$

Arguments

S

specifies the spot currency price.

E

specifies the exercise price of the option.

t
specifies the time to expiration, in years.

R_d
specifies the risk-free domestic interest rate for period *t*.

R_f
specifies the risk-free foreign interest rate for period *t*.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(R_d - R_f + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{PUT} = \max((E - S), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

Example

The following program illustrates the GARKHPTPRC function:

```
data _null_;
   method run();
      a=garkhptprc(50, .7, 55, .05, .04, .2);
      b=garkhptprc(32, .3, 33, .05, .03, .3);
      put a=;
      put b=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1.4050880944848
1.56473205137371
```

See Also

Functions:

- "GARKHCLPRC Function" on page 656

GCD Function

Returns the greatest common divisor for a set of integers.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

GCD(*expression-1*, *expression-2* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The GCD (greatest common divisor) function returns the greatest common divisor of one or more integers. For example, the greatest common divisor for 30 and 42 is 6. The greatest common divisor is also called the highest common factor.

Example

The following program illustrates the GCD function:

```
data test(overwrite=yes);
   dcl double x y;
   method run();
      x=gcd(10, 15);
      y=gcd(36, 45);
      put x;
      put y;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
5
9
```

See Also

Functions:

- "LCM Function" on page 778

GEODIST Function

Returns the geodetic distance between two latitude and longitude coordinates.

Categories: CAS
Distance

Returned data type: DOUBLE

Syntax

GEODIST(*latitude-1*, *longitude-1*, *latitude-2*, *longitude-2* [,*options*])

Arguments

latitude
is a numeric constant, variable, or expression that specifies the coordinate of a given position north or south of the equator. Coordinates that are located north

of the equator have positive values; coordinates that are located south of the equator have negative values.

Restriction: If the value is expressed in degrees, it must be between 90 and -90. If the value is expressed in radians, it must be between pi/2 and -pi/2.

Data type: DOUBLE

longitude

is a numeric constant, variable, or expression that specifies the coordinate of a given position east or west of the prime meridian, which runs through Greenwich, England. Coordinates that are located east of the prime meridian have positive values; coordinates that are located west of the prime meridian have negative values.

Restriction: If the value is expressed in degrees, it must be between 540 and -540. If the value is expressed in radians, it must be between 3*pi and -3*pi.

Data type: DOUBLE

options

specifies a character constant, variable, or expression that contains any of the following characters:

M specifies distance in miles.

K specifies distance in kilometers. K is the default value for distance.

D specifies that input values are expressed in degrees. D is the default for input values.

R specifies that input values are expressed in radians.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The GEODIST function computes the geodetic distance between any two arbitrary latitude and longitude coordinates. Input values can be expressed in degrees or in radians.

Examples

Example 1: Calculating the Geodetic Distance in Kilometers

The following program shows the geodetic distance in kilometers between Mobile, AL (latitude 30.68 N, longitude 88.25 W), and Asheville, NC (latitude 35.43 N, longitude 82.55 W). The program uses the default K option.

```
data _null_;
   dcl double distance;
   method run();
      distance=geodist(30.68, -88.25, 35.43, -82.55);
      put 'Distance= ' distance 'kilometers';
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Distance= 748.652914703183 kilometers
```

Example 2: Calculating the Geodetic Distance in Miles

The following program uses the M option to compute the geodetic distance in miles between Mobile, AL (latitude 30.68 N, longitude 88.25 W), and Asheville, NC (latitude 35.43 N, longitude 82.55 W).

```
data _null_;
   dcl double distance;
   method run();
      distance=geodist(30.68, -88.25, 35.43, -82.55, 'M');
      put 'Distance = ' distance 'miles';
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Distance = 465.191354181072 miles
```

Example 3: Calculating the Geodetic Distance with Input Measured in Degrees

The following program uses latitude and longitude values that are expressed in degrees to compute the geodetic distance between two locations. Both the D and the M options are specified in the program.

```
data _null_;
   dcl double distance lat1 long1 lat2 long2;
   method run();
      lat1=35.2;
      long1=-78.1;
      lat2=37.6;
      long2=-79.8;
      Distance = geodist(lat1,long1,lat2,long2,'DM');
      put 'Distance= ' Distance 'miles';
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Distance= 190.683975083577 miles
```

Example 4: Calculating the Geodetic Distance with Input Measured in Radians

The following program uses latitude and longitude values that are expressed in radians to compute the geodetic distance between two locations. The program converts degrees to radians before executing the GEODIST function. Both the R and the M options are specified in this program.

```
data _null_;
   dcl double distance pi lat1 long1 lat2 long2;
   method run();
      lat1=35.2;
      long1=-78.1;
      lat2=37.6;
      long2=-79.8;
      pi = constant('pi');
      lat1 = (pi*lat1)/180;
      long1 = (pi*long1)/180;
      lat2 = (pi*lat2)/180;
      long2 = (pi*long2)/180;
      Distance = geodist(lat1,long1,lat2,long2,'RM');
      put 'Distance= ' Distance 'miles';
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Distance= 190.683975083578 miles
```

References

Vincenty, T. 1975. “Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations.” *Survey Review* 22: 99–93.

GEOMEAN Function

Returns the geometric mean.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

GEOMEAN(*expression* [, ...*expression*])

Arguments

expression
is any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the nth root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, GEOMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the GEOMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEAN function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following program illustrates the GEOMEAN function:

```
data _null_;
   dcl double x1 x2;
   method run();
      x1=geomean(1,2,4,4);
      x2=geomean(.,4,12,24);
      put x1=;
      put x2=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x1=2.37841423000544
x2=10.4829655768355
```

See Also

Functions:

- “GEOMEANZ Function” on page 668
- “HARMEAN Function” on page 670
- “HARMEANZ Function” on page 672
- “MEAN Function” on page 820

GEOMEANZ Function

Returns the geometric mean, using zero fuzzing.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

GEOMEANZ(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the nth root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEANZ function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following program illustrates the GEOMEANZ function:

```
data _null_;
   dcl double x1 x2;
   method run();
      x1=geomeanz(1,2,4,4);
      x2=geomeanz(.,4,12,24);
      put x1=;
      put x2=;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log.

```
x1=2.37841423000544
x2=10.4829655768355
```

See Also

Functions:

- “GEOMEAN Function” on page 666
- “HARMEAN Function” on page 670
- “HARMEANZ Function” on page 672
- “MEAN Function” on page 820

HARMEAN Function

Returns the harmonic mean.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

HARMEAN(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1}+\frac{1}{x_2}+\ldots+\frac{1}{x_n}}$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, HARMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the HARMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEAN function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEANZ, HARMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following program illustrates the HARMEAN function:

```
data _null_;
   dcl double x1 x2;
   method run();
      x1=harmean(1,2,4,4);
      x2=harmean(.,4,12,24);
      put x1=;
      put x2=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x1=2
x2=8
```

See Also

Functions:

- “GEOMEAN Function” on page 666
- “GEOMEANZ Function” on page 668
- “HARMEANZ Function” on page 672
- “MEAN Function” on page 820

HARMEANZ Function

Returns the harmonic mean, using zero fuzzing.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

HARMEANZ(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

If any argument is negative, then the result is a null or value. A message appears in the log that the negative argument is invalid. If all the arguments are null or values, then the result is a null or value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1}+\frac{1}{x_2}+\ldots+\frac{1}{x_n}}$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEANZ function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEAN, HARMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following program illustrates the HARMEANZ function:

```
data _null_;
   dcl double x1 x2;
   method run();
      x1=harmeanz(1,2,4,4);
      x2=harmeanz(.,4,12,24);
      put x1= x2=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x1=2 x2=8
```

See Also

Functions:

- “GEOMEAN Function” on page 666
- “GEOMEANZ Function” on page 668
- “HARMEAN Function” on page 670
- “MEAN Function” on page 820

HBOUND Function

Returns the upper bound of an array.

Categories: Array
CAS

Returned data type: INTEGER

Syntax

HBOUND(*array-name*[, *bound-n*])

Arguments

array-name
specifies the name of a temporary or a variable array.

Data type CHAR

bound-n
is a numeric constant, variable, or expression that specifies the dimension, in a multidimensional array, for which you want to know the upper bound. If no *bound-n* value is specified, the HBOUND function returns the upper bound of the first dimension of the array.

Bound-n evaluates to an integral value.

Data type INTEGER

Details

The HBOUND function returns the upper bound of a one-dimensional array, or the upper bound of a specified dimension of a multidimensional array.

HBOUND and LBOUND can be used together to return the values of the upper and lower bounds of an array dimension.

If the HBOUND function is called with a dimension value that is outside the dimension of the array, then a run-time error occurs and the function returns a NULL integer value.

Comparisons

- DIM returns the number of elements in an array dimension.
- HBOUND returns the value of the upper bound of an array dimension.
- LBOUND returns the value of the lower bound of an array dimension.
- NDIMS returns the number of dimensions in an array.

Example

The following program shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions:

```
data _null_;
   dcl char(15) a1[4];
   dcl double a2[2,3,4] sum;
   method init();
      declare char(15) a1[4];
      declare double   a2[2,3,4] sum;
      a1 := ('red' 'yellow' 'green' 'blue');
      a2 := (24*2.0);
      do i = 1 to dim(a1);
         put a1[i];
      end;
      numelems = 0;
      do i = 1 to ndims(a2);
         numelems = numelems + dim(a2, i);
      end;
      sum = 0;
      do i = lbound(a2, 1) to hbound(a2, 1);
         do j = lbound(a2, 2) to hbound(a2, 2);
            do k = lbound(a2, 3) to hbound(a2, 3);
               sum = sum + a2[i,j,k];
            end;
         end;
      end;
      put sum=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
red
yellow
green
blue
sum=48
```

See Also

Functions:

- “DIM Function” on page 521
- “LBOUND Function” on page 776
- “NDIMS Function” on page 838

HMS Function

Returns a SAS time value from hour, minute, and second values.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

HMS(*hour*, *minute*, *second*)

Arguments

hour
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

minute
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

second
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

The HMS function returns a numeric value that represents a SAS time value. A SAS time value is a number that represents the number of seconds since midnight of the current day.

Example

The following program illustrates the HMS function:

```
data test(overwrite=yes);
   dcl double a;
   dcl varchar(10) b;
   method run();
      a=hms(12,45,10);
      b=put(a, time.);
      put a;
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
45910
12:45:10
```

See Also

Concepts:

- "Dates and Times in DS2" in *SAS DS2 Programmer's Guide*

Functions:

- "DHMS Function" on page 515
- "HOUR Function" on page 682
- "MINUTE Function" on page 824
- "SECOND Function" on page 1071

HOLIDAY Function

Returns a SAS date value of a specified holiday for a specified year.

Categories: CAS

Date and Time

Returned data type: DOUBLE

Syntax

HOLIDAY('*holiday*', *year*)

Arguments

'holiday'

is a character constant, variable, or expression that specifies one of the values listed in the following table.

Values for *holiday* can be in uppercase or lowercase.

Table 7.3 *Holiday Values and Their Descriptions*

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canadian Independence Day	July 1
CANADAOBSERVED	Canadian Independence Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25
COLUMBUS	Columbus Day	2nd Monday in October
EASTER	Easter Sunday	date varies
FATHERS	Father's Day	3rd Sunday in June
HALLOWEEN	Halloween	October 31
LABOR	Labor Day	1st Monday in September
MLK	Martin Luther King, Jr. 's birthday	3rd Monday in January beginning in 1986
MEMORIAL	Memorial Day	last Monday in May (since 1971)

Holiday Value	Description	Date Celebrated
MOTHERS	Mother's Day	2nd Sunday in May
NEWYEAR	New Year's Day	January 1
THANKSGIVING	U.S. Thanksgiving Day	4th Thursday in November
THANKSGIVINGCANADA	Canadian Thanksgiving Day	2nd Monday in October
USINDEPENDENCE	U.S. Independence Day	July 4
USPRESIDENTS	Abraham Lincoln's and George Washington's birthdays observed	3rd Monday in February (since 1971)
VALENTINES	Valentine's Day	February 14
VETERANS	Veterans Day	November 11
VETERANSUSG	Veterans Day - U.S. government-observed	U.S. government-observed date for Monday–Friday schedule
VETERANSUSPS	Veterans Day - U.S. post office observed	U.S. government-observed date for Monday–Saturday schedule (U.S. Post Office)
VICTORIA	Victoria Day	Monday on or preceding May 24

Data type CHAR

year
is a numeric constant, variable, or expression that specifies a four-digit year. If you use a two-digit year, then you must specify the YEARCUTOFF= system option.

Data type DOUBLE

Details

The HOLIDAY function computes the date on which a specific holiday occurs in a specified year. Only certain common U.S. and Canadian holidays are defined for use with this function. (See "Holiday Values and Their Descriptions" in *SAS Functions and CALL Routines: Reference* for a list of valid holidays.)

The definition of many holidays has changed over the years. In the U.S., Executive Order 11582, issued on February 11, 1971, fixed the observance of many U.S. federal holidays.

The current holiday definition is extended indefinitely into the past and future, although many holidays have a fixed date at which they were established. Some holidays have not had a consistent definition in the past.

The HOLIDAY function returns a SAS date value. To convert the SAS date value to a calendar date, use any valid SAS date format, such as the DATE9. format.

Comparisons

In some cases, the HOLIDAY function and the NWKDOM function return the same result. For example, the statement `holiday('thanksgiving', 2012);` returns the same value as `nwkdom(4, 5, 11, 2012);`.

In other cases, the HOLIDAY function and the MDY function return the same result. For example, the statement `holiday('christmas', 2012);` returns the same value as `mdy(12, 25, 2012);`.

Example

The following programs illustrate the HOLIDAY function:

```
/*** Thanksgiving ***/

data _null_;
   dcl double thanks having format date9.;
   method init();
      thanks = holiday('thanksgiving', 2019);
      put thanks;
   end;
enddata;
run;
quit;

/*** Boxing ***/
data _null_;
   method run();
      dcl double boxing having format date9.;
      boxing = holiday('boxing', 2019);
      put boxing;
   end;
```

```
enddata;
run;

/*** Easter ***/
data _null_;
   method run();
      dcl double easter having format date9.;
      easter = holiday('easter', 2019);
      put easter;
   end;
enddata;
run;

/*** Canada Day ***/
data _null_;
   method run();
      dcl double canada having format date9.;
      canada = holiday('canada', 2019;
      put canada;
   end;
enddata;
run;

/*** Father's Day ***/
proc ds2;
data _null_;
   method run();
     dcl double fathers having format date9.;
      fathers = holiday('fathers', 2019);
      put fathers;
   end;
enddata;
run;

/*** Valentine's Day ***/
data _null_;
   method run();
      dcl double valentines having format date9.;
      valentines = holiday('valentines', 2019);
      put valentines;
   end;
enddata;
run;

/*** Victoria Day ***/
data _null_;
   method run();
      dcl double victoria having format date9.;
      victoria = holiday('victoria', 2019);
      put victoria;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
28NOV2019

26DEC2019

21APR2019

01JUL2019

16JUN2019

14FEB2019

20MAY2019
```

See Also

Functions:

- “MDY Function” on page 818
- “NWKDOM Function” on page 880

HOUR Function

Returns the hour from a SAS time or datetime value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

HOUR(*time* | *datetime*)

Arguments

time
specifies any valid expression that represents a SAS time value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

The HOUR function returns a numeric value that represents the hour from a SAS time or datetime value. Numeric values can range from 0 through 23. HOUR always returns a positive number.

Example

The following program illustrates the HOUR function:

```
data _null_;
   dcl double a;
   method run();
      a=hour(time());
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
14
```

See Also

- “Dates and Times in DS2” in *SAS DS2 Programmer’s Guide*

Functions:

- “MINUTE Function” on page 824
- “SECOND Function” on page 1071

IBESSEL Function

Returns the value of the modified Bessel function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

IBESSEL(*nu*, *x*, *kode*)

Arguments

nu

specifies a numeric constant, variable, or expression.

Range *nu* ≥ 0

Data type DOUBLE

x

specifies a numeric constant, variable, or expression.

Range *x* ≥ 0

Data type DOUBLE

kode

is a numeric constant, variable, or expression that specifies a nonnegative whole number.

Data type DOUBLE

Details

The IBESSEL function returns the value of the modified Bessel function of order *nu* evaluated at *x* (Abramowitz, Stegun 1964; Amos, Daniel, Weston 1977). When *kode* equals 0, the Bessel function is returned. Otherwise, the value of the following function is returned:

$$\varepsilon^{-x} I_{nu}(x)$$

Example

The following program illustrates the IBESSEL function:

```
data _null_;
   dcl double x;
   method run();
      x=ibessel(2, 2, 0);
      put x=;
      x=ibessel(2, 2, 1);
      put x=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x=0.68894844769873
x=0.09323903330473
```

See Also

Functions:

- "JBESSEL Function" on page 764

INDEX Function

Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.

Categories: CAS
Character

Returned data type: INTEGER

Syntax

INDEX(*target-expression*, *search-expression*)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

search-expression
specifies any valid expression that evaluates or can be coerced to a character string that is used to search for in *target-expression*.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in single quotation marks.

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The INDEX function searches *target-expression*, from left to right, for the first occurrence of the string specified in *search-expression*, and returns the position in *target-expression* of the string's first character. If the string is not found in *target-expression*, INDEX returns a value of 0. If there are multiple occurrences of the string, INDEX returns only the position of the first occurrence.

Note: When either the *target-expression* or the *search-expression* has a length of 0, the INDEX function returns a value of 0. It should also be noted that an undeclared VARCHAR variable has a length of 0.

Comparisons

The VERIFY function returns the position of the first character in *target-expression* that does not contain *search-expression* where the INDEX function returns the position of the first occurrence of *search-expression* that is present in *target-expression*.

Example

The following program illustrates the INDEX statement:

```
data _null_;
   dcl double c;
   dcl varchar(20) a b;
   method run();
      a='ABC.DEF (X=Y)';
      b='X=Y';
      c=index(a,b);
      put c=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
c=10
```

See Also

Functions:

- "INDEXC Function" on page 687
- "INDEXW Function" on page 689

- “VERIFY Function” on page 1164

INDEXC Function

Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

INDEXC(*target-expression*, *search-expression*[, ...*search-expression*])

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string that is searched.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

search-expression
specifies the characters to search for in *target-expression*.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in single quotation marks.

See: “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

The INDEXC function searches *target-expression*, from left to right, for the first occurrence of any character present in the search expressions and returns the position in *target-expression* of that character. If none of the characters in the search expressions are found in *target-expression*, INDEXC returns a value of 0.

Comparisons

The INDEXC function searches for the first occurrence of any individual character that is present within the search expression, whereas the INDEX function searches for the first occurrence of the search expression as a pattern.

Example

The following program illustrates the INDEXC function:

```
data _null_;
   dcl double x y;
   dcl varchar(20) a b c;
   method run();
      a='ABC.DEP (X2=Y1)';
      b='()';
      c='=.';
      x=indexc(a,'0123',';()=.');
      put x=;
      x=indexc(a,b);
      put x=;
      x=indexc(a,b,c);
      put x=;
      c='have a good day';
      x=indexc(c,'pleasant','very');
      put x=;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
x=4
x=9
x=4
x=2
```

See Also

Functions:

- “INDEX Function” on page 685
- “INDEXW Function” on page 689

INDEXW Function

Searches a character expression for a string that is specified as a word, and returns the position of the first character in the word.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

INDEXW(*target-expression*, *search-expression* [, *delimiter*])

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string and that is searched.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

search-expression
specifies any valid expression that evaluates or can be coerced to a character string and that is searched for in *target-expression*. SAS removes the leading and trailing delimiters from *search-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

delimiter
specifies a character expression that you want INDEXW to use as a word separator in the character strings. The default delimiter is the blank character.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip If the blank character is a delimiter, order it so that it is not the last character in *delimiter*. Trailing blanks are ignored because *delimiter* is trimmed of trailing blanks.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The INDEXW function searches *target-expression*, from left to right, for the first occurrence of *search-expression* and returns the position in *target-expression* of the substring's first character. If the substring is not found in *target-expression*, then INDEXW returns a value of 0. If there are multiple occurrences of the string, then INDEXW returns only the position of the first occurrence.

The substring pattern must begin and end on a word boundary. For INDEXW, word boundaries are delimiters, the beginning of *target-expression*, and the end of *target-expression*.

TIP INDEXW has the following behavior when *search-expression* contains blank spaces or has a length of 0:

- If both *target-expression* and *search-expression* contain only blank spaces or have a length of 0, then INDEXW returns a value of 1.
- If *search-expression* contains only blank spaces or has a length of 0, and *target-expression* contains character or numeric data, then INDEXW returns a value of 0.

Comparisons

The INDEXW function searches for strings that are words, whereas the INDEX function searches for patterns as separate words or as parts of other words. INDEXC searches for any characters that are present in the excerpts.

Example

The following program illustrates the INDEXW function:

```
data _null_;
   dcl double c b;
   dcl varchar(20) a word x xyz;
   method run();
      a='The power to know.';
      word='power';
      c=indexw(a,word);
      put c=;
      a='The power to know.';
      b=indexw(a,'know');
      put b=;
      a='The power to know.';
      b=indexw(a,'know','. ');
      put b=;
      a='abc,def@ xyz';
      b=indexw(a,',', '@');
```

```
        put b=;
        a='abc,def@ xyz';
        b=indexw(a,'def', '@,');
        put b=;
        x='abc,def@ xyz';
        xyz=indexw(x, ' xyz', '@');
        put xyz=;
    end;
enddata;
run;
```

SAS writes the following output to the log.

```
c=5
b=0
b=14
b=0
b=5
xyz=9
```

See Also

Functions:

- “INDEX Function” on page 685
- “INDEXC Function” on page 687

INPUTC Function

Enables you to specify a character informat at run time.

Categories: CAS
Special

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INPUTC(*source*, *informat*[, *w*])

Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

informat
is a character constant, variable, or expression that contains the character informat that you want to apply to *source*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

w
specifies any valid expression that evaluates to a numeric width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If the INPUTC function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

Comparisons

The INPUTN function enables you to specify a numeric informat at run time.

Example

The following program illustrates how to specify character informats.

```
data _null_;
   dcl char(10) type type2;
   method init();
      type=inputc('positive', '$upcase15.');
      type2=inputc('positive', '$upcase15.', 3);
      put type=;
      put type2=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
type=POSITIVE
type2=POS
```

See Also

Functions:

- “INPUTN Function” on page 693
- “PUT Function” on page 992

INPUTN Function

Enables you to specify a numeric informat at run time.

Categories: CAS
Special

Returned data type: DOUBLE

Syntax

INPUTN(*source*, *informat*[, *w*[, *d*]])

Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR

informat
is a character constant, variable, or expression that contains the numeric informat that you want to apply to *source*.

Data type CHAR

w
is a numeric constant, variable, or expression that specifies a width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type DOUBLE

d

is a numeric constant, variable, or expression that specifies the number of decimal places to use.

Interaction: If you specify a number here, it overrides any decimal-place specification in the informat.

Data type: DOUBLE

Comparisons

The INPUTC function enables you to specify a character informat at run time. Using the PUT function is faster because you specify the informat at compile time.

Example

The following program illustrates how to specify numeric informats.

```
data _null_;
   method init();
      declare double salary;
      salary = inputn('20,000.00', 'comma10.2');
      put salary=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
20000
```

See Also

Functions:

- “INPUTC Function” on page 691
- “PUT Function” on page 992

INT Function

Returns the whole number, fuzzed to avoid unexpected floating-point results.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

INT(*expression*)

Arguments

expression
specifies any expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The INT function returns the whole number portion of the argument (truncates the decimal portion). If the argument's value is within 1E-12 of a whole number, the function results in that whole number. If the value of *expression* is positive, the INT function has the same result as the FLOOR function. If the value of *expression* is negative, the INT function has the same result as the CEIL function.

Comparisons

Unlike the INTZ function, the INT function fuzzes the result. If the argument is within 1E-12 of a whole number, the INT function fuzzes the result to be equal to that whole number. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

Example

The following program illustrates the INT function:

```
data test(overwrite=yes);
   dcl double var1 a b c d;
   method run();
      var1=2.1;
      a=int(var1);
      put a;
      b=int(-2.4);
      put b;
      c=int(1+1.e-11);
      put c;
```

```
        d=int(-1.6);
        put d;
    end;
enddata;
run;
```

SAS writes the following output to the log.

```
2
-2
1
-1
```

See Also

Functions:

- “CEIL Function” on page 419
- “FLOOR Function” on page 644
- “INTZ Function” on page 757
- “MOD Function” on page 829
- “MODZ Function” on page 831
- “ROUND Function” on page 1035

INTCINDEX Function

Returns the cycle index when a date, time, or timestamp interval and value are specified.

Categories:	CAS Date and Time
Restriction:	DS2 does not support custom date or time intervals.
Returned data type:	DOUBLE

Syntax

INTCINDEX({*interval*[*multiple*][.*shift-index*]}, *date-time-value*)

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

date-time-value
specifies a date, time, or timestamp value that represents a time period of a specified interval.

Data type DOUBLE

Details

The Basics

The INTCINDEX function returns the index of the seasonal cycle when you specify an interval and a DATE, TIME, or TIMESTAMP value. For example, if the interval is MONTH, each observation in the data corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12. WEEK is the seasonal cycle for an interval that is equal to DAY. This example returns a value of 36 because September 1, 2013, is the sixth day of the 35th week of the year.

```
sasdate1=to_double(date'2013-09-01');
cycle_index1 = intcindex('day', sasdate1);
```

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

The INTCINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For a list of these intervals, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][*.shift-index*]

For more information, see "Arguments" on page 697.

Comparisons

The INTCINDEX function returns the cycle index, whereas the INTINDEX function returns the seasonal index.

In this example, the INTCINDEX function returns the week of the year.

```
sasdate1=to_double(date'04apr2013');
cycle_index = intcindex('day', sasdate1);
```

In this example, the INTINDEX function returns the day of the week.

```
sasdate1=to_double(date'04apr2013');
index = intindex('day','04APR2013'd);
```

In this example, the INTCINDEX function returns the hour of the day.

```
sasts=to_double(timestamp '2012-09-01 00:00:00');
a= intcindex('minute', sasts);
```

In this example, the INTINDEX function returns the minute of the hour.

```
sasts=to_double(timestamp'2012-09-01 00:00:00');
a= intindex('minute', sasts);
```

In the example `intseas(intcycle('interval'));`, the INTSEAS function returns the maximum number that could be returned by `intcindex('interval', date);`.

Example

The following programs illustrate the INTCINDEX function:

```
data test (overwrite=yes);
   dcl double sasdate1 cycle_index1;
   method run();
      sasdate1=as_double(date'2013-09-01');
      cycle_index1 = intcindex('day', sasdate1);
      put cycle_index1;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
36
```

```
data test (overwrite=yes);
   dcl double sascydate cycle_index1;
   method run();
      sascydate=as_double(timestamp '2013-05-23 05:03:01');
      cycle_index1 = intcindex('dtqtr', sascydate);
      put cycle_index1;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1
```

```
data test (overwrite=yes);
   dcl double sascydate cycle_index1;
```

```
   method run();
      sascydate=as_double(date'2013-12-13');
      cycle_index1 = intcindex('tenday', sascydate);
      put cycle_index1;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1
```

```
data test (overwrite=yes);
   dcl double sascydate cycle_index1;
   method run();
      sascydate=as_double(time '23:13:02');
      cycle_index1 = intcindex('minute', sascydate);
      put cycle_index1;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
24
```

```
data test (overwrite=yes);
   dcl double sascydate cycle_index1;
   dcl char(10) var1;
   method run();
      sascydate=as_double(timestamp '2013-05-05 10:54:03');
      var1='semimonth';
      cycle_index1 = intcindex(var1, sascydate);
      put cycle_index1;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1
```

See Also

Functions:

- “INTCYCLE Function” on page 710
- “INTINDEX Function” on page 722
- “INTSEAS Function” on page 744

INTCK Function

Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: DOUBLE

Syntax

INTCK({*interval*[*multiple*][*.shift-index*]}, *start-date*, *end-date*[, '*method*'])

INTCK(*start-date*, *end-date*[, '*method*'])

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Note: The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type: DOUBLE

See: "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example: YEAR2 specifies a two-year, or biennial, interval type.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

start-date
specifies an expression that represents the starting SAS date, time, or timestamp value.

Data type DOUBLE

end-date
specifies an expression that represents the ending SAS date, time, or timestamp value.

Data type DOUBLE

'method'
specifies that intervals are counted using either a discrete or a continuous method.

You must enclose *method* in quotation marks. *Method* can be one of these values:

CONTINUOUS
specifies that continuous time is measured. The interval is shifted based on the starting date.

For example, the distance in months between January 15, 2013, and February 15, 2013, is one month.

Alias C or CONT

DISCRETE
specifies that discrete time is measured. The discrete method counts interval boundaries (for example, end of month).

The default discrete method is useful to sort time series observations into bins for processing. For example, daily data can be accumulated to monthly data for processing as a monthly series.

For the DISCRETE method, the distance in months between January 31, 2013, and February 1, 2013, is one month.

Alias D or DISC

Default DISCRETE

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see “Arguments” on page 701.

Calendar Interval Calculations

All values within a discrete time interval are interpreted as being equivalent. This means that the dates of January 1, 2013 and January 15, 2013 are equivalent when you specify a monthly interval. Both of these dates represent the interval that begins on January 1, 2013 and ends on January 31, 2013. You can use the date for the beginning of the interval (January 1, 2013) or the date for the end of the interval (January 31, 2013) to identify the interval. These dates represent all of the dates within the monthly interval.

In the following example, the *start-date* (Jan. 14, 2013) is equivalent to the first quarter of 2013.

```
sasdate1=to_double(date'2013-01-14');
sasdate2 = to_double(date'2013-09-02');
qtr=intck('qtr', sasdate1, sasdate2);
```

The *end-date* (September 2, 2013) is equivalent to the third quarter of 2013. The interval count, that is, the number of times the beginning of an interval is reached in moving from the *start-date* to the *end-date* is 2.

The INTCK function using the default discrete method counts the number of times the beginning of an interval is reached in moving from the first date to the second. It does not count the number of complete intervals between two dates:

- The following example returns 0, because the two dates are within the same month.

```
sasdate1=to_double(date'2013-01-01');
sasdate2 = to_double(date'2013-01-31');
month=intck(month, sasdate1, sasdate2);
put month;
```

- The following example returns 1, because the two dates lie in different months that are one month apart.

```
sasdate1=to_double(date'2013-01-31');
sasdate2 = to_double(date'2013-02-01');
month=intck(month, sasdate1, sasdate2);
put month;
```

- The following example returns –1 because the first date is in a later discrete interval than the second date. (INTCK returns a negative value whenever the first date is later than the second date and the two dates are not in the same discrete interval.)

```
sasdate1=to_double(date'2013-02-01');
sasdate2 = to_double(date'2013-01-31');
month=intck('month', sasdate1, sasdate2);
put month;
```

Using the discrete method, WEEK intervals are determined by the number of Sundays, the default first day of the week, that occur between the *start-date* and the *end-date*, and not by how many seven-day periods fall between those dates. To count the number of seven-day periods between *start-date* and *end-date*, use the continuous method.

Both the *multiple* and the *shift-index* arguments are optional and default to 1. For example, YEAR, YEAR1, YEAR.1, and YEAR1.1 are all equivalent ways of specifying ordinary calendar years.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

Intervals by Category

Table 7.4 *Intervals Used with Date and Time Functions*

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
Date	DAY	Daily intervals	Each day	Days	DAY3	Three-day intervals

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
						starting on Sunday
	WEEK	Weekly intervals of seven days	Each Sunday	Days (1=Sunday ... 7=Saturday)	WEEK.7	Weekly with Saturday as the first day of the week
	WEEKDAY <*days*W>	Daily intervals with Friday-Saturday-Sunday	Each day	Days	WEEKDAY1W	Six-day week with Sunday as a weekend day
		counted as the same day (five-day work week with a Saturday-Sunday weekend). *days* identifies the weekend days by number (1=Sunday ... 7=Saturday). By default, *days*=17.			WEEKDAY35W	Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)
	TENDAY	Ten-day intervals (a U.S. automobile industry convention)	First, 11th, and 21st of each month	Ten-day periods	TENDAY4.2	Four ten-day periods starting at the second TENDAY period

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
	SEMIMONTH	Half-month intervals	First and 16th of each month	Semi-monthly periods	SEMIMONTH2.2	Intervals from the 16th of one month through the 15th of the next month
	MONTH	Monthly intervals	First of each month	Months	MONTH2.2	February-March, April-May, June-July, August-September, October-November, and December-January of the following year
	QTR	Quarterly (three-month) intervals	January 1 April 1 July 1 October 1	Months	QTR3.2	Three-month intervals starting on April 1, July 1, October 1, and January 1
	SEMIYEAR	Semiannual (six-month) intervals	January 1 July 1	Months	SEMIYEAR.3	Six-month intervals, March-August, and September-February
	YEAR	Yearly intervals	January 1	Months		
Datetime	Add DT to any of the date intervals	Interval that corresponds to the associated	Midnight of January 1, 1960		DTMONTH DTWEEKDAY	

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
		date interval				
Time	SECOND	Second intervals	Start of the day (midnight)	Seconds		
	MINUTE	Minute intervals	Start of the day (midnight)	Minutes		
	HOUR	Hourly intervals	Start of the day (midnight)	Hours		

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Example

The following programs illustrate the INTCK function:

In the second example, INTCK returns a value of 1 even though only one day has elapsed. This result is returned because the interval from December 31, 2012, to January 1, 2013, contains the starting point for the YEAR interval. However, in the third example, a value of 0 is returned even though 364 days have elapsed. This result occurs because the period between January 1, 2013, and December 31, 2013, does not contain the starting point for the interval.

In the fourth example, SAS returns a value of 6 because the period of January 1, 2010, through January 1, 2013, contains six semiyearly intervals. (Note that if the ending date were December 31, 2012, SAS would count five intervals.) In the fifth example, SAS returns a value of 6 because there are six two-week intervals beginning on a first Monday during the period of January 7, 2013, through April 1, 2013. In the sixth example, SAS returns the value 27. That indicates that beginning with January 1, 2013, and counting only Saturdays as weekend days through February 1, 2013, the period contains 27 weekdays.

In the seventh example, the use of variables for the arguments is illustrated.

```
data _null_;
```

```
   dcl double sasdate1 sasdate2 qtr;
   method run();
      sasdate1 = to_double(date'2013-01-10');
      sasdate2 = to_double(date'2013-07-01');
      qtr=intck('qtr', sasdate1, sasdate2);
      put qtr;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2
```

```
data test (overwrite=yes);
   dcl double sasdate1 sasdate2 year;
   method run();
      sasdate1 = to_double(date'2012-12-31');
      sasdate2 = to_double(date'2013-01-01');
      year=intck('year', sasdate1, sasdate2);
      put year;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1
```

```
data test (overwrite=yes);
   dcl double sasdate1 sasdate2 year;
   method run();
      sasdate1 = to_double(date'2013-01-01');
      sasdate2 = to_double(date'2013-12-31');
      year=intck('year', sasdate1, sasdate2);
      put year;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
0
```

```
data test (overwrite=yes);
   dcl double sasdate1 sasdate2 semiyear;
   method run();
      sasdate1 = to_double(date'2010-01-01');
      sasdate2 = to_double(date'2013-01-01');
      semiyear=intck('semiyear', sasdate1, sasdate2);
      put semiyear;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
6
```

```
data test (overwrite=yes);
   dcl double sasdate1 sasdate2 weekvar;
   method run();
      sasdate1 = to_double(date'2013-01-07');
      sasdate2 = to_double(date'2013-04-01');
      weekvar=intck('week2.2', sasdate1, sasdate2);
      put weekvar;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
6
```

```
data test (overwrite=yes);
   dcl double sasdate1 sasdate2 wdvar;
   method run();
      sasdate1 = to_double(date'2013-01-01');
      sasdate2 = to_double(date'2013-02-01');
      wdvar=intck('weekday7w', sasdate1, sasdate2);
      put wdvar;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
27
```

```
data test (overwrite=yes);
   dcl double sasdate1 sasdate2 newyears;
   dcl char y;
   method run();
      y='year';
      sasdate1 = to_double(date'2003-09-01');
      sasdate2 = to_double(date'2013-09-01');
      newyears=intck(y, sasdate1, sasdate2);
      put newyears;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
10
```

See Also

Functions:

- "INTDT Function" on page 715
- "INTNX Function" on page 732
- "INTTS Function" on page 755

Other References:

- "Dates and Times in DS2" in *SAS DS2 Programmer's Guide*

INTCYCLE Function

Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTCYCLE({*interval*[*multiple*][.*shift-index*]}[, *seasonality*])

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Note: The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type: DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

seasonality

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default 52

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example The *seasonality* argument in the following example

```
INTCYCLE('MONTH', 3);
```

causes the function call to return the value QTR. The function call

```
INTCYCLE('MONTH');
```

does not have a *seasonality* argument and returns the value YEAR.

Details

The Basics

The INTCYCLE function returns the interval of the seasonal cycle, depending on a date, time, or datetime interval. For example, `INTCYCLE('MONTH');` returns the value YEAR because the months from January through December constitute a yearly cycle. `INTCYCLE('DAY');` returns the value WEEK because the days from Sunday through Saturday constitute a weekly cycle.

For information about multipliers and shift indexes, see "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference*. For information about how intervals are calculated, see "Commonly Used Time Intervals" in *SAS Functions and CALL Routines: Reference*.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*..

The INTCYCLE function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Functions and CALL Routines: Reference*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][.*shift-index*]

For more information, see "Arguments" on page 710.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTCYCLE function uses the concept of seasonality and returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the *SAS/ETS User's Guide*.

Example

The following programs illustrate the INTCYCLE function:

```
data _null_;
   dcl char(10) cycle_year;
   method init();
      cycle_year=intcycle('year');
      put cycle_year;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
YEAR
```

```
data _null_;
   dcl char(10) cycle_quarter;
   method init();
      cycle_quarter=intcycle('qtr');
      put cycle_quarter;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
YEAR
```

```
data test(overwrite=yes);
   dcl char(10) cycle_3;
   method init();
      cycle_3=intcycle('month', 3);
      put cycle_3;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
QTR
```

```
data test(overwrite=yes);
   dcl char(10) cycle_month;
   method init();
      cycle_month=intcycle('month');
      put cycle_month;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
YEAR
```

```
data test(overwrite=yes);
   dcl char(10) cycle_weekday;
   method init();
      cycle_weekday=intcycle('weekday');
```

```
      put cycle_weekday;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
WEEK
```

```
data  test(overwrite=yes);
   dcl char(10) cycle_weekday2;
   method init();
      cycle_weekday2=intcycle('weekday', 5);
      put cycle_weekday2;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log.

```
WEEK
```

```
data test(overwrite=yes);
   dcl char(10) cycle_day;
   method init();
      cycle_day=intcycle('day');
      put cycle_day;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
WEEK
```

```
data  test(overwrite=yes);
   dcl char(10) cycle_day2;
   method init();
      cycle_day2=intcycle('day', 10);
      put cycle_day2;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
TENDAY
```

```
data test(overwrite=yes);
   dcl char(10) cycle_second;
   dcl char(6) var1;
   method init();
      var1='second';
      cycle_second=intcycle(var1);
      put cycle_second;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
DTMINUTE
```

See Also

Functions:

- “INTCINDEX Function” on page 696
- “INTINDEX Function” on page 722
- “INTSEAS Function” on page 744

Other References:

- *SAS/ETS User's Guide*

INTDT Function

Specifies the number of days to add to a DATE value.

Categories: CAS
Date and Time

Returned data type: DATE

Syntax

INTDT(*expression*, *increment*)

Arguments

expression
specifies any valid expression that represents a DATE value.

Data type DATE

See “DS2 Expressions” in *SAS DS2 Programmer's Guide*

increment
specifies a negative, positive, or zero whole number that represents the number of days to add to the date.

Data type DOUBLE

Details

The INTDT function increments a DATE value by the number of days that you specify.

Comparisons

The INTNX function increments a SAS date, time, or datetime value that is encoded as a DOUBLE value.

Example

The following program illustrates the INTDT function:

```
data test(overwrite=yes);
   dcl date a b c d;
   method run();
      a= date '2019-05-01';
      b=intdt(a, 25);
      put a;
      put b;
      c= date '2019-05-01';
      d=intdt(c, -25);
      put c;
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2019-05-01
2019-05-26
2019-05-01
2019-04-06
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "INTCK Function" on page 701

- “INTNX Function” on page 732
- “INTTS Function” on page 755

INTFIT Function

Returns a time interval that is aligned between two dates.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTFIT(*expresssion-1*, *expression-2*, '*type*')

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string and that represents a SAS date or datetime value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

'type'
specifies whether the arguments are SAS date values, datetime values, or a row.

The following values for *type* are valid:

d specifies that *expression-1* and *expression-2* are date values.

dt specifies that *expression-1* and *expression-2* are datetime values.

obs specifies that *expression-1* and *expression-2* are rows.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The INTFIT function returns the most likely time interval based on two dates, datetime values, or rows that have been aligned within an interval. INTFIT assumes that the alignment value is SAME, which specifies that the date is aligned to the

same calendar date with the corresponding interval increment. For more information about the *alignment* argument, see "INTNX Function" in *SAS Functions and CALL Routines: Reference*.

If the arguments that are used with INTFIT are rows, you can determine the cycle of an occurrence by using row numbers. In the following example, the first two arguments of INTFIT are row numbers, and the *type* argument is `obs`. If Jason used the gym the first time and the 25th time that a researcher recorded data, you could determine the interval by using the following statement: `interval=intfit(1, 25, 'obs');`. In this case, the value of interval is OBS24.2.

For information about time series, see the *SAS/ETS 9.3 User's Guide*.

The INTFIT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Example

The following program illustrates the interval that is aligned between two dates. The *type* argument in this program identifies the input as date values.

```
data test;
   dcl char(10) c;
   dcl double sasdate1 sasdate2;
   method run();
      sasdate1=to_double(date'2013-08-01');
      sasdate2=to_double(date'2013-09-01');
      c=intfit(sasdate1, sasdate2, 'd');
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
MONTH
```

See Also

Functions:

- "INTCK Function" on page 701
- "INTNX Function" on page 732

INTGET Function

Returns a time interval based on three date or datetime values.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTGET(*date-1*, *date-2*, *date-3*)

Argument

date
specifies any valid expression that evaluates to a SAS date or datetime value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

INTGET Function Intervals

The INTGET function returns a time interval based on three date or datetime values. The function first determines all possible intervals between the first two dates, and then determines all possible intervals between the second and third dates. If the intervals are the same, INTGET returns that interval. If the intervals for the first and second dates differ, and the intervals for the second and third dates differ, INTGET compares the intervals. If one interval is a multiple of the other, then INTGET returns the smaller of the two intervals. Otherwise, INTGET returns a missing value. INTGET works best with dates generated by the INTNX function whose alignment value is BEGIN.

In the following example, INTGET returns the interval DAY2:

```
interval=intget('01mar00'd, '03mar00'd, '09mar00'd);
```

The interval between the first and second dates is DAY2, because the number of days between March 1, 2000, and March 3, 2000, is two. The interval between the second and third dates is DAY6, because the number of days between March 3,

2000, and March 9, 2000, is six. DAY6 is a multiple of DAY2. INTGET returns the smaller of the two intervals.

In the following example, INTGET returns the interval MONTH4:

```
interval=intget('01jan00'd, '01may00'd, '01may01'd);
```

The interval between the first two dates is MONTH4, because the number of months between January 1, 2000, and May 1, 2000, is four. The interval between the second and third dates is YEAR. INTGET determines that YEAR is a multiple of MONTH4 (there are three MONTH4 intervals in YEAR), and returns the smaller of the two intervals.

In the following example, INTGET returns a missing value:

```
interval=intget('01Jan2006'd, '01Apr2006'd, '01Dec2006'd);
```

The interval between the first two dates is MONTH3, and the interval between the second and third dates is MONTH8. INTGET determines that MONTH8 is not a multiple of MONTH3, and returns a missing value.

The intervals that are returned are valid SAS intervals, including multiples of the intervals and shift intervals. Valid SAS intervals are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts* .

Note: If INTGET cannot determine a matching interval, then the function returns a missing value. No message is written to the SAS log.

Retail Calendar Intervals

The INTGET function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Example

The following programs illustrate the INTGET function:

```
data test (overwrite=yes);
   dcl char(10) c;
   dcl double sasdate1 sasdate2 sasdate3;
   method run();
      sasdate1=to_double(date'2013-01-01');
      sasdate2=to_double(date'2014-01-01');
      sasdate3=to_double(date'2014-05-01');
      c=intget(sasdate1, sasdate2, sasdate3);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
MONTH4
```

```
data test (overwrite=yes);
   dcl char(10) c;
   dcl double sasdate1 sasdate2 sasdate3;
   method run();
      sasdate1=to_double(date'2012-02-29');
      sasdate2=to_double(date'2014-02-28');
      sasdate3=to_double(date'2016-02-29');
      c=intget(sasdate1, sasdate2, sasdate3);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
YEAR2.2
```

```
data test (overwrite=yes);
   dcl char(10) c;
   dcl double sasdate1 sasdate2 sasdate3;
   method run();
      sasdate1=to_double(date'2013-02-01');
      sasdate2=to_double(date'2013-02-16');
      sasdate3=to_double(date'2013-03-01');
      c=intget(sasdate1, sasdate2, sasdate3);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
SEMIMONTH
```

```
data test (overwrite=yes);
   dcl char(10) c;
   dcl double sasdate1 sasdate2 sasdate3;
   method run();
      sasdate1=to_double(date'2013-01-02');
      sasdate2=to_double(date'2014-02-02');
      sasdate3=to_double(date'2015-03-02');
      c=intget(sasdate1, sasdate2, sasdate3);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
MONTH13.13
```

```
data test (overwrite=yes);
   dcl char(10) c;
   dcl double sasdate1 sasdate2 sasdate3;
   method run();
      sasdate1=to_double(date'2013-02-10');
```

```
      sasdate2=to_double(date'2013-02-19');
      sasdate3=to_double(date'2013-02-28');
      c=intget(sasdate1, sasdate2, sasdate3);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
DAY9.5
```

```
data test (overwrite=yes);
   dcl char(10) c;
   dcl double sasdate1 sasdate2 sasdate3;
   method run();
      sasdate1=to_double(timestamp'2014-04-01 01:03:00.0000');
      sasdate2=to_double(timestamp'2014-04-01 01:04:00.0000');
      sasdate3=to_double(timestamp'2014-04-01 01:05:00.0000');
      c=intget(sasdate1, sasdate2, sasdate3);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
MINUTE
```

See Also

Functions:

- “INTFIT Function” on page 717
- “INTNX Function” on page 732

INTINDEX Function

Returns the seasonal index when a date, time, or timestamp interval and value are specified.

Categories:	CAS Date and Time
Restriction:	DS2 does not support custom date or time intervals.
Returned data type:	DOUBLE

Syntax

INTINDEX({*interval*[*multiple*][.*shift-index*]}, *date-value*[, *seasonality*])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

date-value
specifies a date, time, or timestamp value that represents a time period of the given interval.

Data type DOUBLE

seasonality
specifies a number or a cycle.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example In this example, the following functions produce the same result.

```
INTINDEX('MONTH', sasdate, 3);
INTINDEX('MONTH', sasdate, 'QTR');
```

Seasonality in the first example is a number (the number of months), and in the second example *seasonality* is a cycle (QTR).

Details

The Basics

The INTINDEX function returns the seasonal index when you supply an interval and an appropriate date, time, or timestamp value. The seasonal index is a number that represents the position of the date, time, or timestamp value in the seasonal cycle of the specified interval. This example returns a value of 12 because there are 12 months in a yearly cycle and December is the 12th month of the year.

```
sasdate=to_double(date'2012-12-01');
x=intindex('month', sasdate);
put x;
```

In the following examples, INTINDEX returns the same value (1) because both statements have dates that occur in the first quarter of the year 2013.

```
sasdate=to_double(date'2013-01-01');
x=intindex('qtr', sasdate);
put x;

sasdate=to_double(date'2013-03-31');
y=intindex('qtr', sasdate);
```

```
put y;
```

The following example returns a value of 6 because daily data is weekly periodic and December 7, 2012, is a Friday, the sixth day of the week.

```
sasdate=to_double(date'2012-12-07');
x=intindex('day', sasdate);
put x;
```

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][*.shift-index*]

For more information, see "Arguments" on page 723.

How *Interval* and *Date-Time-Value* Are Related

To correctly identify the seasonal index, the interval should agree with the date, time, or timestamp value. For example, `intindex('month', '01DEC2012'd);` returns a value of 12 because there are 12 months in a yearly interval and December is the 12th month of the year. The MONTH interval requires a SAS date value. The following example, returns a value of 6 because there are seven days in a weekly interval and December 7, 2012, is a Friday, the sixth day of the week.

```
sasdate=to_double(date'2012-12-07');
x=intindex('day', sasdate);
put x;
```

The DAY interval requires a SAS date value.

```
select intget('day', date'2012-12-07');
```

This example returns a missing value because the QTR interval expects the date to be a SAS date value rather than a TIMESTAMP value.

```
sasdate=to_double(timestamp'2013-01-01 00:00:00');
x=intindex('qtr', sasdate);
put x;
```

This example returns a value of 12. The DTMONTH interval requires a TIMESTAMP value.

```
sasdate=to_double(timestamp'2013-12-01 00:00:00');
x=intindex('dtmonth', sasdate);
put x;
```

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

Retail Calendar Intervals

The INTINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Functions and CALL Routines: Reference*.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTINDEX function uses the concept of seasonality and returns the seasonal index when a date, time, or timestamp interval and value are specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the *SAS/ETS User's Guide*.

Comparisons

The INTINDEX function returns the seasonal index whereas the INTCINDEX function returns the cycle index.

In the following example, the INTINDEX function returns 5 because April 4, 2013 is on a Thursday, the fifth day of the week.

```
sasdate=to_double(date'2013-04-04');
x = intindex('day', sasdate);
put x;
```

Using the same date, the INTCINDEX function returns 14 because April 4, 2013 is the 14th week of the year.

```
sasdate=to_double(date'2013-04-04');
x = intcindex('day', sasdate);
put x;
```

In this example, the INTINDEX function returns the minute of the hour.

```
sasdate=to_double(timestamp'2012-09-01 06:05:04');
x = intindex('minute', sasdate);
put x;
```

Using the same date and time, the INTCINDEX function returns the hour of the day.

```
sasdate=to_double(timestamp'2012-09-01 06:05:04');
y = intcindex('minute', sasdate);
put y;
```

In the example `intseas('interval');`, INTSEAS returns the maximum number that could be returned by `intindex('interval', date);`.

Example

The following programs illustrate the INTINDEX function:

```
data test (overwrite=yes);
   dcl double sasdate1 interval1;
   method run();
      sasdate1=to_double(date'2013-08-14');
      interval1 = intindex('qtr', sasdate1);
      put interval1;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
3
```

```
data test (overwrite=yes);
   dcl double sasts1 interval2;
   method run();
      sasts1=to_double(timestamp'2013-12-23 15:09:19');
      interval2 = intindex('dtqtr', sasts1);
      put interval2;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
4
```

```
data test (overwrite=yes);
   dcl double sastime1 interval3;
   method run();
      sastime1=to_double(time'09:05:15');
      interval3 = intindex('hour', sastime1);
      put interval3;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
10
```

```
data test (overwrite=yes);
   dcl double sasdate1 interval4;
   method run();
      sasdate1=to_double(date '2013-02-26');
      interval4 = intindex('month', sasdate1);
      put interval4;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2
```

```
data test (overwrite=yes);
   dcl double sasts1 interval5;
   method run();
      sasts1=to_double(timestamp'2013-05-28 05:15:00');
      interval5 = intindex('dtmonth', sasts1);
      put interval5;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
5
```

```
data test (overwrite=yes);
   dcl double sasdate1 interval6;
   method run();
      sasdate1=to_double(date '2013-09-09');
      interval6 = intindex('week', sasdate1);
      put interval6;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
37
```

```
data test (overwrite=yes);
   dcl double sasdate1 interval7;
   method run();
      sasdate1=to_double(date '2013-04-16');
      interval7 = intindex('tenday', sasdate1);
      put interval7;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
11
```

See Also

Functions:

- “INTCINDEX Function” on page 696
- “INTCYCLE Function” on page 710

- “INTSEAS Function” on page 744

Other References:

- *SAS/ETS User's Guide*

INTNEST Function

Calculates the number of whole periods of the smaller interval that will fit into the period of the larger interval.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: DOUBLE

Syntax

INTNEST(*interval1*, *interval2*)

Arguments

interval1
specifies the first interval.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

interval2
specifies the second interval.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

An interval nests within another interval when a whole number of the first interval spans the same time period as the second interval for all time periods. In order to nest, the two intervals must generate beginning and ending dates that align.

If the first interval, *interval1*, spans a larger time period than the second interval, *interval2*, then the returned number is positive. If the second interval spans a larger period than the first interval, then the returned number is negative.

The following time series tasks are related to the INTNEST function:

accumulation
: if one interval nests into another interval, even with a variable number, accumulation from the smaller time periods into the larger time periods is accomplished with a simple rule. If the intervals do not nest, consider transforming a time series from one frequency to another with a more complex rule (for example, an interpolation).

seasonality
: many seasonal models require that the higher frequency interval nest into the lower frequency seasonal interval with a fixed number of periods.

time reconciliation
: time reconciliation requires that the higher frequency interval nest into the lower frequency interval.

The following table contains the types and descriptions of results that are returned by the INTNEST function:

Table 7.5 *INTNEST Function Results and Descriptions*

Result	Description	Explanation or Example
0	Same	The two input intervals define the same time periods for all time periods. `intnest('month12','year')`
1	Variable Number	The first interval contains a whole number of periods of the second interval, but the number varies over time. `intnest('month','day')`
-1	Variable Number	The second interval contains a whole number of periods of the first interval, but the number varies over time. `intnest('day',''year')`
n > 1	Fixed Number	The first interval contains a whole number *n* periods of the second interval, and that is fixed for all time. `intnest('week', 'day')`
n < -1	Fixed Number	The second interval contains a whole number *n* periods of the first interval, and that is fixed for all time. `intnest('dthour', 'day')`
Missing value of M	Multiple Mismatch	Both intervals cannot nest into the other interval. However, intervals of these types can nest for some multiple values. `intnest('semimonth3', 'month')`

Result	Description	Explanation or Example
Missing value of S	Shift Mismatch	Both intervals cannot nest into the other interval. However, if a shift value is changed, then either the intervals would be the same or one interval would nest into the other. `intnest('semimonth2.2', 'month')`
Missing value of B	Base Mismatch	The interval bases define time periods that are so different that nesting is not possible for any multiple or shift. For example, YEAR always begins on January 1 of each year and is shifted by months. However, YEARV always begins on the Monday on or immediately preceding January 4, and YEARV is shifted by ISO 8601 weeks that begin on Monday. Since January 1 is only a Monday for some years, the intervals will not consistently start on the same day. The same problem exists if the YEAR interval is shifted by months, since the first of a month would not be a Monday for all years. `intnest('year', 'yearv')`

Example

The following program illustrates the INTNEST function:

```
data _null_;
   dcl double a;
   dcl char(10) x y;
   method run();
      x='SEMIMONTH3';
      y='MONTH';
      a=intnest(x, y);
      put a;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
M
```

INTNX Function

Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: DOUBLE

Syntax

INTNX({*interval*[*multiple*][.*shift-index*]}, *start-from*, *increment*[, '*alignment*'])

INTNX(*start-from*, *increment*[, '*alignment*'])

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Note: The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type: DOUBLE

See: "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example: YEAR2 specifies a two-year, or biennial, interval type.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

start-from
specifies an expression that represents a SAS date, time, or datetime value encoded as a DOUBLE and that identifies a starting point.

Data type DOUBLE

increment
specifies a negative, positive, or zero whole number that represents the number of date, time, or datetime intervals. *Increment* is the number of intervals to shift the value of *start-from*.

Data type DOUBLE

'alignment'
controls the position of SAS dates within the interval. You must enclose *alignment* in quotation marks. *Alignment* can be one of these values:

BEGINNING
specifies that the returned date or datetime value is aligned to the beginning of the interval.

Alias B

MIDDLE
specifies that the returned date or datetime value is aligned to the midpoint of the interval, which is the average of the beginning and ending alignment values.

Alias M

END
specifies that the returned date or datetime value is aligned to the end of the interval.

Alias E

SAME
specifies that the date that is returned has the same alignment as the input date.

Aliases S

SAMEDAY

See "SAME Alignment" on page 735

Default BEGINNING

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "Aligning SAS Date Output within Its Intervals" on page 735

Details

The Basics

The INTNX function increments a date, time, or datetime value by intervals such as DAY, WEEK, QTR, and MINUTE, or a custom interval that you define. The increment is based on a starting date, time, or datetime value, and on the number of time intervals that you specify.

The INTNX function returns the SAS date value for the beginning date, time, or datetime value of the interval that you specify in the *start-from* argument. (To convert the date value to a calendar date, use any valid DS2 date format, such as the DATE9. format.) The following example shows how to determine the date of the start of the week that is six weeks from the week of October 17, 2011.

```
sasdate=to_double(date'2011-10-17');
x=intnx('week', sasdate, 6);
put x date9.;
```

INTNX returns the value 27NOV2011.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][*.shift-index*]

For more information, see "Arguments" on page 732.

Aligning SAS Date Output within Its Intervals

SAS date values are typically aligned with the beginning of the time interval that is specified with the *interval* argument.

You can use the optional *alignment* argument to specify the alignment of the date that is returned. The values BEGINNING, MIDDLE, or END align the date to the beginning, middle, or end of the interval, respectively.

SAME Alignment

If you use the SAME value of the *alignment* argument, then INTNX returns the same calendar date after computing the interval increment that you specified. The same calendar date is aligned based on the interval's shift period, not the interval. To view the valid shift periods, see "Intervals by Category" on page 704.

Most of the values of the shift period are equal to their corresponding intervals. The exceptions are the intervals WEEK, WEEKDAY, QTR, SEMIYEAR, YEAR, and their DT counterparts. WEEK and WEEKDAY intervals have a shift period of DAYS; and QTR, SEMIYEAR, and YEAR intervals have a shift period of MONTH. For example, when you use SAME alignment with YEAR, the result is same-day alignment based on MONTH, the interval's shift period. The result is not aligned to the same day of the YEAR interval. If you specify a multiple interval, then the default shift interval is based on the interval, and not on the multiple interval.

When you use SAME alignment for QTR, SEMIYEAR, and YEAR intervals, the computed date is the same number of months from the beginning of the interval as the input date. The day of the month matches as closely as possible. Because not all months have the same number of days, it is not always possible to match the day of the month.

For more information about shift periods, see "Intervals by Category" on page 704.

Alignment Intervals

Use the SAME value of the *alignment* argument if you want to base the alignment of the computed date on the alignment of the input date.

```
/*** returns 22MAR2011 ***/
dcl double x having format date9.;
sasdate=to_double(date'2011-03-15');
x=intnx('week', sasdate, 1, 'same');
put x;
```

```
/*** returns 22MAR11:08:45:00  ***/
dcl double y having format datetime.;
method init();
sasdt=to_double(timestamp'2011-03-15 08:45:00');
y=intnx('dtweek', sasdt, 1, 'same');
put y ;

/*** returns 15MAR2016 ***/
dcl double z having format date9.;
sasdate=to_double(date'2011-03-15');
z=intnx('year', sasdate, 5, 'same');
put z;
```

Adjusting Dates

The INTNX function automatically adjusts for the date if the date in the interval that is incremented does not exist. Here is an example:

```
/*** returns 15AUG2011 ***/
dcl double a having format date9.;
sasdate=to_double(date'2011-03-15');
a=intnx('month', sasdate, 5, 'same');
put a;

/*** returns 28FEB2014 ***/
dcl double b having format date9.;
sasdate=to_double(date'2012-02-29');
b=intnx('year', sasdate, 2, 'same');
put b;


/*** returns 30SEP2011 ***/
dcl double c having format date9.;
sasdate=to_double(date'2011-08-31');
c=intnx('month', sasdate, 1, 'same');
put c;


/*** returns 01MAR2012 (the 1st day of the 3rd month of the year) ***/
dcl double d having format date9.;
sasdate=to_double(date'2011-03-01');
d=intnx('year', sasdate, 1, 'same');
put d;


/*** returns 29FEB2012 (the 60th day of the year) ***/
dcl double d having format date9.;
sasdate=to_double(date'2011-03-01');
d=intnx('year', sasdate, 1, 'same', 'day');
put d;
```

In the following example, the INTNX function returns the value 01JAN2014, which is the beginning of the year two years from the starting date (29FEB2012).

```
dcl double a having format date9.;
sasdate=to_double(date'2012-02-29');
```

```
a=intnx('year', sasdate, 2);
put a;
```

In this example , the INTNX function returns the value 28FEB2014. In this case, the starting date begins in the year 2012, the year is two years later (2014), the month is the same (February), and the date is the 28th, because that is the closest date to the 29th in February 2014.

```
dcl double b having format date9.;
sasdate=to_double(date'2012-02-29');
b=intnx('year', sasdate, 2, 'same');
put b;
```

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Examples

Example 1: Using the INTNX Function

The following programs illustrate the INTNX function:

```
data test (overwrite=yes);
   dcl double sasdate1 yr;
   method run();
      sasdate1 = to_double(date'2019-02-05');
      yr=intnx('year', sasdate1, 3);
      put yr;
      put yr date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
22646
01JAN22
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   method run();
      sasdate1 = to_double(date'2019-01-05');
      x=intnx('month', sasdate1, 0);
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21550
01JAN19
```

```
data test (overwrite=yes);
   dcl double sasdate1 next;
   method run();
      sasdate1 = to_double(date'2019-01-01');
      next=intnx('semiyear', sasdate1, 1);
      put next;
      put next date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21731
01JUL19
```

```
data test (overwrite=yes);
   dcl double sasdate1 past;
   method run();
      sasdate1 = to_double(date'2019-08-01');
      past=intnx('month2', sasdate1, -1);
      put past;
      put past date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21670
01MAY19
```

```
data test (overwrite=yes);
   dcl double sasdate1 sm;
   method run();
      sasdate1 = to_double(date'2019-04-01');
      sm=intnx('semimonth2.2', sasdate1, 4);
      put sm;
      put sm date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21746
16JUL19
```

```
data test (overwrite=yes);
   dcl double sasdate1 nextmon;
   dcl char x;
```

```
   method run();
      x='month';
      sasdate1 = to_double(date'2019-06-01');
      nextmon=intnx(x, sasdate1, 1);
      put nextmon;
      put nextmon date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21731
01JUL19
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   dcl char m1 m2;
   method run();
      m1='month     ';
      m2=trim(m1);
      sasdate1 = to_double(date'2019-06-15') - 100;
      x=intnx(m2, sasdate1, 1);
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21640
01APR19
```

Example 2: Using the ALIGNMENT Argument

The following programs show the results of advancing a date by using the optional *alignment* argument.

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   method run();
      sasdate1 = to_double(date'2019-01-01');
      x=intnx('month', sasdate1, 5, 'beginning');
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21701
01JUN19
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
```

```
   method run();
      sasdate1 = to_double(date'2019-01-01');
      x=intnx('month', sasdate1, 5, 'middle');
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21715
15JUN19
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   method run();
      sasdate1 = to_double(date'2019-01-01');
      x=intnx('month', sasdate1, 5, 'end');
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21730
30JUN19
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   method run();
      sasdate1 = to_double(date'2019-01-01');
      x=intnx('month', sasdate1, 5, 'sameday');
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21701
01JUN19
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   method run();
      sasdate1 = to_double(date'2019-03-15');
      x=intnx('month', sasdate1, 5, 'same');
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21776
15AUG19
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   dcl char interval align;
   method run();
      interval='month';
      align='m';
      sasdate1 = to_double(date'2019-09-01');
      x=intnx(interval, sasdate1,2, align);
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21868
15NOV19
```

```
data test (overwrite=yes);
   dcl double sasdate1 x;
   dcl char m1 m2;
   method run();
      m1='month     ';
      m2=trim(m1);
      sasdate1 = to_double(date'2019-09-01');
      x=intnx(m2, sasdate1, 2, 'm');
      put x;
      put x date7.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
21868
15NOV19
```

See Also

Functions:

- “INTCK Function” on page 701
- “INTDT Function” on page 715
- “INTSHIFT Function” on page 748
- “INTTS Function” on page 755

Other References:

- "Dates and Times in DS2" in *SAS DS2 Programmer's Guide*

INTRR Function

Returns the internal rate of return as a decimal value.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

INTRR(*freq*, *c0, c1*[*..., cn*])

Arguments

freq
is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *freq* > 0

Data type DOUBLE

Tip The case *freq* = 0 is a flag to allow continuous compounding.

***c0, c1*[*..., cn*]**
are numeric, the optional cash payments.

Data type DOUBLE

Details

The INTRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c0*, *c1*, ..., *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument *freq* > 0 describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

The internal rate of return is the interest rate such that the sequence of payments has a 0 net present value. (See the "NPV Function" on page 876.) It is given by the following equation.

$$r = \begin{cases} \frac{1}{x^{freq}} - 1 & freq > 0 \\ -log_e(x) & freq = 0 \end{cases}$$

In this equation, *x* is the real root of the polynomial.

$$\sum_{i=0}^{n} c_i x^i = 0$$

In the case of multiple roots, one real root is returned and a warning is issued concerning the non-uniqueness of the returned internal rate of return. Depending on the value of payments, a root for the equation does not always exist. In that case, a missing value is returned.

Missing values in the payments are treated as 0 values. When *freq* > 0, the computed rate of return is the effective rate over the specified base period. To compute a quarterly internal rate of return (the base period is three months) with monthly payments, set *freq* to 3.

If *freq* is 0, continuous compounding is assumed and the base period is the time interval between two consecutive payments. The computed internal rate of return is the nominal rate of return over the base period. To compute with continuous compounding and monthly payments, set *freq* to 0. The computed internal rate of return is a monthly rate.

Comparisons

The IRR function is identical to INTRR, except for in the IRR function, the internal rate of return is a percentage.

Example

For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return can be calculated as follows:

```
data _null_;
   dcl double rate;
   method init();
      rate=intrr(1,-400,100,200,300);
      put rate;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
0.19437709962747
```

```
0.1943770996
```

See Also

Functions:

- “IRR Function” on page 762

INTSEAS Function

Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: DOUBLE

Syntax

INTSEAS({*interval*[*multiple*][.*shift-index*]}[, *seasonality*])

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Note: The possible values of *interval* are listed in “Intervals Used with Date and Time Functions” in *SAS Language Reference: Concepts*.

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type: DOUBLE

See: “Incrementing Dates and Times by Using Multipliers and By Shifting Intervals” in *SAS Functions and CALL Routines: Reference* for more information.

Example: YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions: The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type: DOUBLE

See: "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example: YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

seasonality

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default: 52

Data type: DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example: The *seasonality* argument in the following example

```
INTSEAS('MONTH', 'qtr');
```

causes the function call to return the value 3. The function call

```
INTSEAS('MONTH');
```

does not have a *seasonality* argument and returns the value 12.

Details

The Basics

The INTSEAS function returns the number of intervals in a seasonal cycle. For example, when the interval for a time series is described as monthly, then many

procedures use the option INTERVAL=MONTH. Each observation in the data then corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12.

Quarterly data is also considered to be periodic for a one-year period. A year contains four quarters, so the number of intervals in a seasonal cycle is four.

The periodicity is not always one year. For example, INTERVAL=DAY is considered to have a period of one week. Because there are seven days in a week, the number of intervals in the seasonal cycle is seven.

For more information about working with date and time intervals, see "Date and Time Intervals" in *SAS Functions and CALL Routines: Reference*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][*.shift-index*]

For more information, see "Arguments" on page 744.

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTSEAS function uses the concept of seasonality and returns the length of the seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and forecasting, see the *SAS/ETS User's Guide*.

Example

The following program illustrates the INTSEAS function:

```
data test (overwrite=yes);
   dcl double cycle_years cycle_smiyears cycle_quarters cycle_number;
   dcl double cycle_months cycle_smimonths cycle_tendays;
   dcl double cycle_weeks cycle_wkdays cycle_hours  cycle_minutes;
   dcl double cycle_minutes cycle_month2 cycle_week2 cycle_var1;
   dcl couble cycle_day1;
   dcl char var1;
   method run();
      cycle_years = intseas('year');
      put cycle_years;

      cycle_smiyears = intseas('semiyear');
      put cycle_smiyears;

      cycle_quarters = intseas('quarter');
      put cycle_quarters;

      cycle_number = intseas('month', 'qtr');
      put cycle_number;

      cycle_months = intseas('month');
      put cycle_months;

      cycle_smimonths = intseas('semimonth');
      put cycle_smimonths;

      cycle_tendays = intseas('tenday');
      put cycle_tendays;

      cycle_weeks = intseas('week');
      put cycle_weeks;

      cycle_wkdays = intseas('weekday');
      put cycle_wkdays;

      cycle_hours = intseas('hour');
      put cycle_hours;

      cycle_minutes = intseas('minute');
      put cycle_minutes;

      cycle_month2 = intseas('month2.2');
      put cycle_month2;

      cycle_week2 = intseas('week2');
      put cycle_week2;

      var1 = 'month4.3';
      cycle_var1 = intseas(var1);
      put cycle_var1;

      cycle_day1 = intseas('day1');
      put cycle_day1;

   end;
enddata;
```

```
run;
```

SAS writes the following output to the log.

```
1
2
4
3
12
24
36
52
5
24
60
6
26
3
7
```

See Also

Functions:

- “INTCYCLE Function” on page 710
- “INTINDEX Function” on page 722

Other References:

- *SAS/ETS User's Guide*

INTSHIFT Function

Returns the shift interval that corresponds to the base interval.

Categories:	CAS Date and Time
Restriction:	DS2 does not support custom date or time intervals.
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTSHIFT(*interval*[*multiple*][.*shift-index*])

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies yearly intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 consists of two-year, or biennial, periods.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

Details

The Basics

The INTSHIFT function returns the shift interval that corresponds to the base interval. For custom intervals, the value that is returned is the base custom interval name. INTSHIFT ignores multiples of the interval and interval shifts.

The INTSHIFT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][*.shift-index*]

For more information, see "Arguments" on page 749.

Example

The following programs illustrate the INTSHIFT function:

```
data test (overwrite=yes);
   dcl char(10) c;
   method run();
      c=intshift('year');
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
MONTH
```

```
data test (overwrite=yes);
   dcl char(10) c;
   method run();
      c=intshift('dtyear');
      put c;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
DTMONTH
```

```
data test (overwrite=yes);
   dcl char(10) c;
   method run();
      c=intshift('minute');
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
DTMINUTE
```

```
data test (overwrite=yes);
   dcl char(10) c x;
   method run();
      x='weekdays';
      c=intshift(x);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
WEEKDAY
```

```
data test (overwrite=yes);
   dcl char(10) c;
   method run();
      c=intshift('weekdays5.4');
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
WEEKDAY
```

```
data test (overwrite=yes);
   dcl char(10) c;
   method run();
      c=intshift('qtr');
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
MONTH
```

```
data test (overwrite=yes);
   dcl char(10) c;
   method run();
      c=intshift('dttenday');
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
DTTENDAY
```

INTTEST Function

Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.

Categories: CAS
Date and Time

Restriction: DS2 does not support custom date or time intervals.

Returned data type: DOUBLE

Syntax

INTTEST(*interval*[*multiple*][.*shift-index*])

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Note: The possible values of *interval* are listed in "Intervals Used with Date and Time Functions" in *SAS Language Reference: Concepts*.

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

See "Incrementing Dates and Times by Using Multipliers and By Shifting Intervals" in *SAS Functions and CALL Routines: Reference* for more information.

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

Details

The Basics

The INTTEST function checks for a valid interval name. This function is useful when checking for valid values of *multiple* and *shift-index*. For more information about multipliers and shift indexes, see "Multiunit Intervals" in *SAS Language Reference: Concepts*.

The INTTEST function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval[*multiple*][*.shift-index*]

For more information, see "Arguments" on page 752.

Example

In the following program, SAS returns a value of 1 if the *interval* argument is valid, and 0 if the interval argument is invalid.

```
data test (overwrite=yes);
   dcl double test1 test2 test3 test4 test5;
   dcl char var1;
   method run();
      test1 = inttest('month');
      put test1;

      test2 = inttest('week6.13');
      put test2;

      test3 = inttest('tenday');
      put test3;

      test4 = inttest('twoweeks');
      put test4;

      var1 = 'hour2.2';
      test5 = inttest(var1);
      put test5;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
1
1
1
0
1
```

INTTS Function

Specifies the number of seconds to add to a TIMESTAMP value.

Categories: CAS
Date and Time

Returned data type: TIMESTAMP

Syntax

INTTS(*expression*, *increment*)

Arguments

expression
specifies any valid expression that represents a TIMESTAMP value.

Data type TIMESTAMP

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

increment
specifies a negative, positive, or zero whole number that represents the number of seconds to add to the time.

Data type DOUBLE

Details

The INTTS function increments a TIMESTAMP value by the number of seconds that you specify.

Comparisons

The INTNX function increments a SAS date, time, or datetime value encoded as a DOUBLE value.

Example

The following programs illustrate the INTTS function:

```
data _null_;
   dcl timestamp y z;
   method init();
      y= timestamp '2019-03-01 16:51:36.00';
      z=intts(y, 43);
      put y;
      put z;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2019-03-01 16:51:36
2019-03-01 16:52:19
```

```
data _null_;
   dcl timestamp y z;
   dcl date a;
   method init();
      a= date '2019-01-01';
      y= timestamp '2019-05-01 02:58:17.00';
      z=intts(y, -2500);
      put a;
      put y;
      put z;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2019-01-01
2019-05-01 02:58:17
2019-05-01 02:16:37
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "INTCK Function" on page 701
- "INTDT Function" on page 715
- "INTNX Function" on page 732

INTZ Function

Returns the whole number portion of the argument, using zero fuzzing.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

INTZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The following rules apply:

- If the value of the argument is an exact whole number, INTZ returns that whole number.
- If the argument is positive and not a whole number, INTZ returns the largest whole number that is less than the argument.
- If the argument is negative and not a whole number, INTZ returns the smallest whole number that is greater than the argument.

Comparisons

Unlike the INT function, the INTZ function uses zero fuzzing. If the argument is within 1E-12 of a whole number, the INT function fuzzes the result to be equal to that whole number. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

Example

The following program illustrates the INTZ function:

```
data test(overwrite=yes);
   dcl double var1 var2 var3 var4 a b c d ;
   method run();
      var1=2.1;
      a=intz(var1);
      put a;
      var2=-2.4;
      b=intz(var2);
      put b;
      var3=1+1.e-11;
      c=intz(var3);
      put c;
      var4=-1.6;
      d=intz(var4);
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2
-2
1
-1
```

See Also

Functions:

- “CEIL Function” on page 419
- “CEILZ Function” on page 421
- “FLOOR Function” on page 644
- “FLOORZ Function” on page 646
- “INT Function” on page 694
- “MOD Function” on page 829
- “MODZ Function” on page 831
- “ROUND Function” on page 1035
- “ROUNDZ Function” on page 1047

IPMT Function

Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

IPMT(*rate*, *period*, *number-of-periods*, *principal-amount*[, *future-amount*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

period
specifies the payment period for which the interest payment is computed.

Requirement *Period* must be a positive whole number that is less than or equal to the value of *number-of-periods*.

Data type INTEGER

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type INTEGER

principal-amount
specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing value is specified.

future-amount
specifies the future amount.

Data type DOUBLE

Notes *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings.

Zero is assumed if *future-amount* is omitted or if a missing value is specified.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments.

Data type INTEGER

Note 0 is assumed if *type* is omitted or if a missing value is specified.

Example

The interest payment on the first periodic payment of an $8,000 loan, where the nominal annual interest rate is 10% and the end-of-period monthly payments are 36, is computed as follows:

```
data test (overwrite=yes);
   dcl double interestpaid1;
   method run();
      InterestPaid1 = ipmt(0.1/12, 1, 36, 8000);
      put interestpaid1;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
66.6666666666666

66.66667
```

If the same loan has beginning-of-period payments, then the interest payment can be computed as follows:

- ```
  data test (overwrite=yes);
     dcl double interestpaid2;
     method run();
        InterestPaid2 = ipmt(0.1/12, 1, 36, 8000, 0, 1);
        put interestpaid2;
     end;
  enddata;
  run;
  ```

  This computation returns a value of 0.

- ```
  data test (overwrite=yes);
     dcl double interestpaid3;
     method run();
        InterestPaid3 = ipmt(0.1, 3, 3, 8000);
        put interestpaid3;
  ```

```
   end;
enddata;
run;
```

This computation returns a value of 292.447129909366.

- ```
  data test (overwrite=yes);
     dcl double interestpaid4;
     method run();
        InterestPaid4 = ipmt(0.09/12, 359, 360, 125000, 0, 1);
        put interestpaid4;
     end;
  enddata;
  run;
  ```

  This computation returns a value of 14.8075736630449.

IQR Function

Returns the interquartile range.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

IQR(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If all arguments have null or missing values, the result is a null or missing value depending on whether you are in ANSI mode or SAS mode. For more information, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.
  ```

Otherwise, the result is the interquartile range of the non-null or nonmissing values. The formula for the interquartile range is the same as the one that is used in the Base SAS UNIVARIATE procedure. For more information, see *Base SAS Procedures Guide: Statistical Procedures*.

Example

The following program illustrates the IQR function:

```
data test (overwrite=yes);
   dcl double a;
   method run();
      a=iqr(2,4,1,3,999999);
      put  'a= ' a;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
2
```

See Also

Functions:

- “MAD Function” on page 808
- “PCTL Function” on page 884

IRR Function

Returns the internal rate of return as a percentage.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

IRR(*freq*, *c1, c2* [*..., cn*])

Arguments

freq
is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *freq* > 0.

Data type DOUBLE

Tip The case *freq* = 0 is a flag to allow continuous compounding.

***c1, c2*[..., *cn*]**
are numeric, the optional cash payments.

Requirement At minimum, two cash payment values are required.

Data type DOUBLE

Details

The IRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c1*, *c2*, ..., *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument *freq* > 0 describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

Comparisons

The IRR function is identical to INTRR, except that in the IRR function, the internal rate of return is a percentage.

Example

For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return as a percentage can be expressed by the following program:

```
data test (overwrite=yes);
   dcl double rate2;
   method run();
      rate2=irr(1,-400,100,200,300);
      put rate2;
   end;
enddata;
run;
```

The value that is returned is 19.437709962747.

See Also

Functions:

- “INTRR Function” on page 742

JBESSEL Function

Returns the value of the Bessel function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

JBESSEL(*nu*, *x*)

Arguments

nu
specifies a numeric constant, variable, or expression.

Range $nu \geq 0$

x
specifies a numeric constant, variable, or expression.

Range $x \geq 0$

Details

The JBESSEL function returns the value of the Bessel function of order *nu* evaluated at *x* (For more information, see Abramowitz and Stegun 1964; Amos, Daniel, and Weston 1977).

Example

The following program illustrates the JBESSEL function:

```
data test(overwrite=yes);
```

```
   dcl double x;
   method run();
      x=jbessel(2,2);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.35283402861563
```

See Also

Functions:

- "IBESSEL Function" on page 683

JULDATE Function

Returns the Julian date from a SAS date value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

JULDATE(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE function converts a SAS date value to a Julian date. If *date* falls within the 100-year span defined by the system option YEARCUTOFF=, the result has three, four or five digits: In a five-digit result, the first two digits represent the year, and the next three digits represent the day of the year (1 to 365, or 1 to 366 for leap years). As leading zeros are dropped from the result, the year portion of a Julian date can be omitted (for years ending in 00) or it can have only one digit (for years ending 01–09). Otherwise, the result has seven digits: the first four digits represent the year, and the next three digits represent the day of the year.

For years that end between 00–09, you can format the five-digit Julian date by using the Z5. format.

For more information about how DS2 handles dates, see "DS2 Expressions" in *SAS DS2 Programmer's Guide*.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following program illustrates the JULDATE function:

```
data;
   dcl double julian1 julian3 julian4;
   dcl double julian2 having format z5.;
   method run();
      julian1=juldate(mdy(12,31,2007));
      put julian1;
      julian2=juldate(mdy(12,31,2007));
      put julian2;
      julian3=juldate(mdy(9,1,1999));
      put julian3;
      julian4=juldate(mdy(7,1,1886));
      put julian4;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
7365
07365
99244
1886182
```

See Also

Functions:

- “DATEJUL Function” on page 501
- “JULDATE7 Function” on page 767

JULDATE7 Function

Returns a seven-digit Julian date from a SAS date value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

JULDATE7(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE7 function returns a seven-digit Julian date from a SAS date value. The first four digits represent the year, and the next three digits represent the day of the year.

For more information about how DS2 handles dates, see “DS2 Expressions” in *SAS DS2 Programmer’s Guide*.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following program illustrates the JULDATE7 function:

```
data test(overwrite=yes);
   dcl double julian7;
   method run();
      julian7=juldate7(mdy(12,31,1996));
      put julian7=;
      julian7=juldate7(mdy(1,1,2099));
      put julian7=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
julian7=1996366
julian7=2099001
```

See Also

Functions:

- “JULDATE Function” on page 765

KURTOSIS Function

Returns the kurtosis.

Categories:	CAS Descriptive Statistics
Returned data type:	DOUBLE

Syntax

KURTOSIS(*expression-1, expression-2, expression-3, expression-4* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	At least four non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.
Data type	DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

Kurtosis is primarily a measure of the heaviness of the tails of a distribution. Large kurtosis values indicate that the distribution has heavy tails.

Null values and missing values are ignored and are not included in the computation.

If all non-null or nonmissing arguments have equal values, the kurtosis is mathematically undefined and the KURTOSIS function returns a null or missing value.

Example

The following program illustrates the KURTOSIS function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3 x4 x5;
   method run();
      x1=kurtosis(5, 9, 3, 6);
      x2=kurtosis(5, 8, 9, 6, .);
      x3=kurtosis(8, 9, 6, 1);
      x4=kurtosis(8, 1, 6, 1);
      x5=kurtosis(of x1-x4);
      put _all_;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=0.92799999999999 x2=-3.3 x3=1.5 x4=-4.48337950138504 x5=-5.06569275379433 _N_=1
```

LAG Function

Returns values from a queue.

Categories: CAS
Special

Returned data type: Same as the expression's data type

Syntax

LAG [*n*] (*expression*)

Arguments

n
specifies the number of lagged values.

Range 1 to 100

Data type Integer

expression
specifies any valid expression.

Data type ANY

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The Basics

If the LAG function returns a value to a character variable that has not yet been assigned a length, by default the variable is assigned the same length as the variable used in the argument.

The LAG functions, LAG1, LAG2, ..., LAG*n* return values from a queue. LAG1 can also be written as LAG. A LAG*n* function stores a value in a queue and returns the value supplied by the *n*th previous call. Each occurrence of a LAG*n* function in a program generates its own queue of values.

The queue for each occurrence of LAG*n* is initialized with *n* missing values, where *n* is the length of the queue (for example, a LAG2 queue is initialized with two missing values). When an occurrence of LAG*n* is executed, the value at the top of

its queue is removed and returned, the remaining values are shifted upward, and the new value of the argument is placed at the bottom of the queue. Hence, missing values are returned for the first *n* executions of each occurrence of LAG*n*, after which the lagged values of the argument begin to appear.

Note: Storing values at the bottom of the queue and returning values from the top of the queue occurs only when the function is executed. An occurrence of the LAG*n* function that is executed conditionally stores and return values only from the observations for which the condition is satisfied.

TIP If the argument of LAG*n* is an array element, such as X[i], a separate queue is maintained for each index into the array.

Memory Limit for the LAG Function

When the LAG function is compiled, SAS allocates memory in a queue to hold the values of the variable that is listed in the LAG function. For example, if the variable in function LAG100(x) is numeric with a length of 8 bytes, then the memory that is needed is 8 times 100, or 800 bytes. Therefore, the memory limit for the LAG function is based on the memory that SAS allocates, which varies with different operating environments.

Examples

Example 1: Generating Two Lagged Values

The following program generates two lagged values for each observation.

LAG1 returns one missing value and the values of X (lagged once). LAG2 returns two missing values and the values of X (lagged twice).

```
proc ds2;
data two (overwrite=yes);
   dcl char(8) x y z;
   method run();
      do x='abc', 'def','ghi','jkl','mno','pqr';
         y=lag1(x);
         z=lag2(x);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=two;
   title LAG Output;
run;
```

Output 7.10 *Output from Generating Two Lagged Values*

LAG Output

Obs	x	y	z
1	abc		
2	def	abc	
3	ghi	def	abc
4	jkl	ghi	def
5	mno	jkl	ghi
6	pqr	mno	jkl

Example 2: Generating a Fibonacci Sequence of Numbers

The following program generates a Fibonacci sequence of numbers. You start with 0 and 1, and then add the two previous Fibonacci numbers to generate the next Fibonacci number.

```
data _null_;
   dcl int n f;
   method run();
      put 'Fibonacci Sequence';
      n=1;
      f=1;
      put n= f=;
      do n=2 to 10;
         f=sum(f, lag(f));
         put n=f=;
      end;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
Fibonacci Sequence
n=1 f=1
n=2 f=1
n=3 f=2
n=4 f=3
n=5 f=5
n=6 f=8
n=7 f=13
n=8 f=21
n=9 f=34
n=10 f=55
```

Example 3: Using Expressions for the LAG Function Argument

The following program uses an expression that creates a data set that contains the values for X, Y, and Z. LAG dequeues the previous values of the expression and enqueues the current value.

```
proc ds2;
data one (overwrite=yes);
   declare int x y z;
```

```
   method run();
      do x=1 to 6;
         y=lag1(x+10);
         z=lag2(x);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=one;
   title LAG Output: Using an Expression;
run;
```

Output 7.11 *Output from the LAG Function: Using an Expression*

LAG Output: Using an Expression

Obs	x	y	z
1	1	.	.
2	2	11	.
3	3	12	1
4	4	13	2
5	5	14	3
6	6	15	4

Example 4: Generating a Lag Queue for Array Elements

The following program generates a separate queue for each index into the array.

```
data _null_;
   dcl double x[2];
   method init();
      dcl int i j k y;
      y=1;
      i=1;
      j=1;
      do while (y > 0 or missing(y));
         x[1]=i * 3;
         x[2]=i * 7;
         if i < 9 or i > 15 then
            k=1;
         else
         k=2;
         y=lag(x[k]);
         put i=y=;
         i=i + 1;
         j=1 - j;
         if (i > 20) then
            y=-1;
      end;
end;
enddata;
```

```
run;
```

SAS writes the following output to the log:

```
i=1 y=
i=2 y=3
i=3 y=6
i=4 y=9
i=5 y=12
i=6 y=15
i=7 y=18
i=8 y=21
i=9 y=
i=10 y=63
i=11 y=70
i=12 y=77
i=13 y=84
i=14 y=91
i=15 y=98
i=16 y=24
i=17 y=48
i=18 y=51
i=19 y=54
i=20 y=57
```

See Also

Functions:

- “DIF Function” on page 518

LARGEST Function

Returns the *k*th largest non-null or nonmissing value.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

LARGEST(*k*, *expression* [, ...*expression*])

Arguments

k

specifies any valid expression that evaluates to a numeric value that represents the largest value to return. For example, if *k* is 2, the LARGEST function returns the second largest value from the list of expressions.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression

specifies any valid expression that evaluates to a numeric value and that is to be searched.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If *k* is null or missing, less than zero, or greater than the number of values, the result is a null or missing value. Otherwise, if *k* is greater than the number of non-null or nonmissing values, the result is a null or missing value.

Example

The following program illustrates the LARGEST function:

```
data test(overwrite=yes);
   dcl double k largest1 largest2 largest3 largest4;
   method run();
      k=1;
      largest1=largest(k, 456, 789, .Q, 123);
      k=2;
      largest2=largest(k, 456, 789, .Q, 123);
      k=3;
      largest3=largest(k, 456, 789, .Q, 123);
      k=4;
      largest4=largest(k, 456, 789, .Q, 123);
      put largest1=;
      put largest2=;
      put largest3=;
      put largest4=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
largest1=789
largest2=456
largest3=123
largest4=.
```

See Also

Functions:

- “ORDINAL Function” on page 883
- “PCTL Function” on page 884
- “SMALLEST Function” on page 1085

LBOUND Function

Returns the lower bound of an array.

Categories: Array
CAS

Returned data type: INTEGER

Syntax

LBOUND(*array-name*[, *bound-n*])

Arguments

array-name
specifies the name of a temporary or a variable array.

bound-n
is a numeric constant, variable, or expression that specifies the dimension, in a multidimensional array, for which you want to know the lower bound.

If no *bound-n* value is specified, the LBOUND function returns the lower bound of the first dimension of the array.

Bound-n evaluates to an integral value.

Details

The LBOUND function returns the lower bound of a one-dimensional array, or the lower bound of a specified dimension of a multidimensional array. LBOUND and HBOUND can be used together to return the values of the lower and upper bounds of an array dimension.

If the LBOUND function is called with a dimension value that is outside the dimension of the array, then a run-time error occurs and the function returns a NULL integer value.

Comparisons

- DIM returns the number of elements in an array dimension.
- HBOUND returns the value of the upper bound of an array dimension.
- LBOUND returns the value of the lower bound of an array dimension.
- NDIMS returns the number of dimensions in an array.

Examples

Example 1: One-Dimensional Array

The following program shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions for a one-dimensional array:

```
data _null_;
   method init();
      declare char(15) a1[4];
      declare double i a2[2,3,4] sum numelems j k;
      a1 := ('red' 'yellow' 'green' 'blue');
      a2 := (24*2.0);
      do i = 1 to dim(a1);
         put a1[i];
      end;
      numelems = 0;
      do i = 1 to ndims(a2);
         numelems = numelems + dim(a2, i);
      end;
      sum = 0;
      do i = lbound(a2, 1) to hbound(a2, 1);
        do j = lbound(a2, 2) to hbound(a2, 2);
          do k = lbound(a2, 3) to hbound(a2, 3);
             sum = sum + a2[i,j,k];
          end;
        end;
      end;
      put sum=;
   end;
```

```
   enddata;
   run;
```

SAS writes the following output to the log:

```
red
yellow
green
blue
sum=48
```

Example 2: Multi-dimensional Array

The following program shows how to use the LBOUND array function for a multi-dimensional array:

```
data test(overwrite=yes);
   dcl double x1 x2 x3;
   vararray double mult[2:6, 4:13, 2] mult1-mult100;
   method run();
      x1=LBOUND(MULT,1);
      x2=LBOUND(MULT,2);
      x3=LBOUND(MULT,3);
      put x1= x2= x3=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=2 x2=4 x3=1
```

See Also

Functions:

- “DIM Function” on page 521
- “HBOUND Function” on page 674
- “NDIMS Function” on page 838

LCM Function

Returns the least common multiple for a set of whole numbers.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

LCM(*expression*-1, *expression*-2 [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a whole number.

Requirement	At least two arguments are required.
Data type	DOUBLE
Note	If any expression evaluates to zero, an error occurs and a missing value is returned.
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The least common multiple is the smallest number that two or more numbers divide into evenly.

Example

The following program illustrates the LCM function:

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=lcm(10, 15);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
30
```

See Also

Functions:

- "GCD Function" on page 662

LCOMB Function

Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of *n* objects taken *r* at a time.

Categories: CAS
Combinatorial

Returned data type: DOUBLE

Syntax

LCOMB(*n*, *r*)

Arguments

n
is a nonnegative whole number that represents the total number of elements from which the sample is chosen.

r
is a nonnegative whole number that represents the number of chosen elements.

Restriction $r \leq n$

Comparisons

The LCOMB function computes the logarithm of the COMB function.

Example

The following program illustrates the LCOMB function:

```
data test(overwrite=yes);
   dcl double x, y;
   method init();
      x=lcomb(5000,500);
      y=lcomb(100,10);
      put x= y=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1621.44113611415
30.4823233622786
```

See Also

Functions:

- “COMB Function” on page 435

LEFT Function

Left aligns a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

LEFT(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

LEFT returns a character string with leading blanks moved to the end of the value.

Example

The following program illustrates the LEFT function:

```
data _null_;
   dcl varchar(12) a;
   method run();
      a=left(' END-OF-YEAR');
      put a;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
END-OF-YEAR
```

See Also

Functions:

- “COMPRESS Function” on page 460
- “RIGHT Function” on page 1032
- “STRIP Function” on page 1102
- “TRIM Function” on page 1151

LENGTH Function

Returns the length of a character string, excluding trailing blanks, in characters.

Categories:	CAS Character
Alias:	LENGTHN
Returned data type:	BIGINT, INTEGER

Syntax

LENGTH(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The LENGTH function returns an integer that represents the position of the rightmost non-blank character or number in *expression*. If the value of *expression* is a blank-filled or an empty string, LENGTH returns a value of 0. If *expression* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string.

Comparisons

- The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks. LENGTH always returns a value that is less than or equal to the value returned by LENGTHC.
- The LENGTH function returns the length of a character string in characters, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. LENGTH always returns a value that is less than or equal to the value returned by LENGTHM. However, with an expression argument, LENGHTM always returns a null value, whereas LENGTH returns the length, in characters, of the character value.

Example

The following program illustrates the LENGTH function. The data type for **g** is converted to NCHAR with a value of 20943 whose length is 5. The data type for **d** and **e**, the null or missing value (.), is converted from DOUBLE to NCHAR with a value of . whose length is 1.

```
data test (overwrite=yes);
   dcl double c d e f g;
   method run();
      c=length('ABDCEF    ');
      d=length('   ');
      e=length(.);
      g=date();
      f=length(g);
      put c;
```

```
        put d;
        put e;
        put f;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
6
0
1
5
```

See Also

Functions:

- “LENGTHC Function” on page 784
- “LENGTHM Function” on page 787

LENGTHC Function

Returns the length of a character string, including trailing blanks, in characters.

Categories: CAS
Character

Returned data type: BIGINT, INTEGER

Syntax

LENGTHC(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

For fixed-length variables, LENGTHC returns the character length of a character string, including trailing blanks. If *expression* is a fixed-length variable, the value returned by LENGTHC is equal to the declared length of the variable. If *expression* is a varying-length variable, the value returned by LENGTHC is less than or equal to the declared length of the variable. If the value of *expression* is missing and contains blanks, LENGTHC returns the number of blanks in *expression*. If *expression* is a numeric expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string.

Comparisons

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTH function returns the length of a character string, excluding trailing blanks. LENGTHC always returns a value that is greater than or equal to the value returned by LENGTH.
- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. For fixed-length character strings, LENGTHC and LENGTHM always return the same value. For varying-length character strings, LENGTHC always returns a value that is less than or equal to the value returned by LENGTHM. However, with an expression argument, LENGHTM always returns a null value, whereas LENGTHC returns the length, in characters, of the character value.

Examples

Example 1: LENGTHC Returns Actual Length of String

The following program illustrates the LENGTHC function:

```
data test (overwrite=yes);
   dcl double c;
   method run();
      c=lengthc('string with trailing blanks     ');
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
32
```

Example 2: LENGTHC with Varying-Length String

The following program illustrates the LENGTHC function:

```
data test2 (overwrite=yes);
   dcl varchar(25) string;
   dcl double a;
   method run();
      string='The power to know.  ';
      a=lengthc(string);
      put a;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
20
```

Example 3: LENGTHC with Fixed-Length String

The following program illustrates the LENGTHC function:

```
data test2 (overwrite=yes);
   dcl char(25) string;
   dcl double b;
   method run();
      string='The power to know.  ';
      b=lengthc(string);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
25
```

Example 4: LENGTHC with a Zero-Length String

The following program illustrates the LENGTHC function:

```
data test2 (overwrite=yes);
   dcl double d;
   method run();
      d=lengthc('');
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
0
```

Example 5: LENGTHC with a Blank String

The following program illustrates the LENGTHC function:

```
data test2 (overwrite=yes);
   dcl double d;
   method run();
      d=lengthc(' ');
```

```
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1
```

See Also

Functions:

- “LENGTH Function” on page 782
- “LENGTHM Function” on page 787

LENGTHM Function

Returns the amount of memory, in bytes, that could or might be allocated for a character string.

Categories: CAS
Character

Returned data type: BIGINT, INTEGER

Syntax

LENGTHM(*variable*)

Arguments

variable
specifies any valid string variable. An expression that evaluates or can be coerced to a string value is an invalid argument.

Restriction: Only variable arguments are allowed. Expressions and literals are not valid and will return a null byte length

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The LENGTHM function returns the maximum amount of memory, in bytes, that might be allocated for a character variable. This is true regardless of the character data type qualifiers such as NATIONAL, VARYING, CHARACTER SET, and so on.

The value returned by the LENGTHM function is independent of the current value stored in the variable and might be greater than the currently allocated byte length of the variable. This is true for both fixed-length and varying-length character variables. For example, if you have a CHAR(100) CHARACTER SET UTF-8 variable or a VARCHAR(100) CHARACTER SET UTF-8 variable, the LENGTHM function returns 400 bytes regardless of the value stored in the variable.

For a literal or an expression, the value returned by the LENGTHM function is a null byte length and a warning is issued that the argument is invalid for the function.

Comparisons

The LENGTHM function returns the amount of memory in bytes that is allocated for a character string, whereas the LENGTH and LENGTHC functions return the length, in characters, of a character value. With a variable argument, LENGTHM always returns a value that is greater than or equal to the values returned by LENGTH and LENGTHC. With an expression argument, LENGHTM always returns a null value, whereas LENGTH and LENGTHC return the length, in characters, of the character value.

Examples

Example 1: LENGHTM with Latin1 Character Set

The following program illustrates the LENGTHM function:

```
data test(overwrite=yes);
   dcl char(25) character set latin1 string;
   dcl double a;
   method run();
      string='The power to know.   ';
      a=lengthm(string);
      put a;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
25
```

Example 2: LENGTHM with UTF-8 Character Set

The following program illustrates the LENGTHM function:

```
data test(overwrite=yes);
   dcl varchar(15) character set utf8 a;
   dcl double b;
   method run();
      a='ABCDEF     ';
      b=lengthm(a);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
60
```

See Also

Functions:

- “LENGTH Function” on page 782
- “LENGTHC Function” on page 784

LFACT Function

Computes the logarithm of the FACT (factorial) function.

Categories: CAS
Combinatorial

Returned data type: DOUBLE

Syntax

LFACT(*n*)

Arguments

n
is a whole number that represents the total number of elements from which the sample is chosen.

Details

The LFACT function computes the logarithm of the FACT function.

Example

The following program illustrates the LFACT function:

```
data _null_;
   dcl double x y ;
   method init();
      x=lfact(5000);
      y=lfact(100);
      put x=   y=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=37591.1435088767 y=363.739375555563
```

See Also

Functions:

- “FACT Function” on page 535

LGAMMA Function

Returns the natural logarithm of the Gamma function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

LGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	Must be a positive number.
Data type	DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the LGAMMA function:

```
data test(overwrite=yes);
   dcl double a;
   method run();
      a=lgamma(2);
      put a=;
      a=lgamma(1.5);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=0
a=-0.12078223763524
```

LOG Function

Returns the natural logarithm (base e) of a numeric value expression.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

LOG(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	Must be a positive number.
Data type	DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the LOG function:

```
data test(overwrite=yes);
   dcl double a b;
   method run();
      a=log(1.0);
      b=log(10.0);
      put a=;
      put b=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=0
b=2.30258509299404
```

See Also

Functions:

- "LOG10 Function" on page 792
- "LOG2 Function" on page 795

LOG10 Function

Returns the base-10 logarithm of a numeric value expression.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

LOG10(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	Must be a positive number.
Data type	DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the LOG10 function:

```
data test(overwrite=yes);
   dcl double a b;
   method run();
      a=log10(1.0);
      b=log10(10.0);
      put a=;
      put b=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=0
b=1
```

See Also

Functions:

- "LOG Function" on page 791
- "LOG2 Function" on page 795

LOG1PX Function

Returns the log of 1 plus the argument.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

LOG1PX(*x*)

Arguments

x

specifies a numeric variable, constant, or expression.

Data type DOUBLE

Details

The LOG1PX function computes the log of 1 plus the argument. The LOG1PX function is mathematically defined by the following equation, where –1 < x:

$$LOG1PX(x) = log(1 + x)$$

When x is close to 0, `LOG1PX(x)` can be more accurate than `LOG(1+x)`.

Examples

Example 1: Computing the Log with the LOG1PX Function

The following program computes the log of 1 plus the value 0.5.

```
data _null_;
   dcl double x;
   method run();
      x=log1px(0.5);
      put x=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=0.40546510810816
```

Example 2: Comparing the LOG1PX Function with the LOG Function

In the following program, the value of X is computed by using the LOG1PX function. The value of Y is computed by using the LOG function.

```
data _null_;
   dcl double x y;
   method run();
      x=log1px(1.e-5);
      put x= hex16.;
      y=log(1+1.e-5);
      put y= hex16.;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=3EE4F8AEA9AE7317
y=3EE4F8AEA9AF0A25
```

See Also

Functions:

- "LOG Function" on page 791

LOG2 Function

Returns the base 2 logarithm of a numeric value expression.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

LOG2(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	Must be a positive number.
Data type	DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the LOG2 function:

```
data test(overwrite=yes);
   dcl double a;
   method run();
      a=log2(8.0);
      put a=;
      a=log2(4);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=3
a=2
```

See Also

Functions:

- "LOG Function" on page 791
- "LOG10 Function" on page 792

LOGBETA Function

Returns the logarithm of the beta function.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

LOGBETA(*a*, *b*)

Arguments

a

is the first shape parameter, where *a*>0.

Data type DOUBLE

b

is the second shape parameter, where *b*>0.

Data type DOUBLE

Details

The LOGBETA function is mathematically given by the equation

$$log(\beta(a,b)) = log(\Gamma(a)) + log(\Gamma(b)) - log(\Gamma(a+b))$$

In the equation, $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, LOGBETA returns a missing value.

Example

The following DS2 program computes the logarithm of the beta function. The first shape parameter is 5, and the second shape parameter is 3.

```
data test(overwrite=yes);
   dcl double y;
   method run();
      y=logbeta(5,3);
      put y=;
   end;
enddata;
run;
```

The following line is written to the SAS log.

```
y=-4.65396035015752
```

See Also

Functions:

- “BETA Function” on page 334

LOGCDF Function

Returns the logarithm of a left cumulative distribution function.

Categories: CAS
Probability

See: “CDF Function” on page 370

Syntax

LOGCDF(*'distribution'*, *quantile* [, *parameter-1*, ..., *parameter-k*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the LOGCDF distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'

Distribution	Argument
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric variable, constant, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The LOGCDF function computes the logarithm of a left cumulative distribution function (logarithm of the left side) from various continuous and discrete distributions. For more information, see the individual distributions in the table above.

For more information about the distributions that are listed in the table, see "CDF Function" on page 370.

See Also

Functions:

- "CDF Function" on page 370
- "LOGPDF Function" on page 801
- "LOGSDF Function" on page 804
- "PDF Function" on page 886
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

LOGISTIC Function

Returns the logistic transformation of the argument.

Categories: CAS
Mathematical

Syntax

LOGISTIC(*argument*)

Arguments

argument
is a numeric variable, constant, or expression that specifies the value of a numeric random variable. When *argument* is a missing or null value, the LOGISTIC function returns a missing or null value.

Details

The LOGISTIC function returns the logistic transformation of an argument. It is typically used to convert a log odds value to a value on the probability scale. The function is mathematically expressed by the following equation:

$$logistic=\frac{e^x}{1+e^x}$$

If the argument contains a missing value, then the LOGISTIC function returns a missing or null value.

Example

The following program illustrates the LOGISTIC function:

```
data test (overwrite=yes);
   dcl double a b;
   method run();
      a=logistic(.5);
      b=logistic(7.3);
      put  'a= ' a;
      put 'b=' b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=  0.62245933120185
b= 0.99932491726936
```

LOGPDF Function

Computes the logarithm of the probability density (mass) function from various continuous and discrete distributions.

Categories: CAS
Probability

Alias: LOGPMF

See: "PDF Function" on page 886

Syntax

LOGPDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the LOGPDF distribution functions are identical to those of the corresponding PDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL' | 'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'

Distribution	Argument
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD' | 'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The LOGPDF function computes the logarithm of the probability density (mass) function from various continuous and discrete distributions. For more information, see the individual distributions in the table above.

For more information about the distributions that are listed in the table, see "PDF Function" on page 886.

See Also

Functions:

- "CDF Function" on page 370
- "LOGCDF Function" on page 798
- "LOGSDF Function" on page 804

- "PDF Function" on page 886
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

LOGSDF Function

Returns the logarithm of a survival function.

Categories: CAS
Probability

See: "SDF Function" on page 1065

Syntax

LOGSDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the LOGSDF distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'

Distribution	Argument
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The LOGSDF function computes the logarithm of the survival function from various continuous and discrete distributions. For more information, see "SDF Function" on page 1065.

For more information about the distributions that are listed in the table, see "CDF Function" on page 370.

See Also

Functions:

- "CDF Function" on page 370
- "LOGCDF Function" on page 798
- "LOGPDF Function" on page 801
- "PDF Function" on page 886
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

LOWCASE Function

Converts all letters in a character expression to lowercase.

Categories:	CAS Character
Alias:	LOWER
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

LOWCASE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Requirement	Literal character expressions must be enclosed in single quotation marks.
Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The LOWCASE function copies a character expression, converts all uppercase letters to lowercase letters, and returns the altered value as a result.

Comparisons

The UPCASE function converts all letters in an argument to uppercase letters. The LOWCASE function converts all letters in an argument to lowercase letters.

Example

The following program illustrates the LOWCASE function:

```
data test(overwrite=yes);
   dcl char(12) x;
   method run();
      x=lowcase('INTRODUCTION');
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
introduction
```

See Also

Functions:

- “UPCASE Function” on page 1155

MAD Function

Returns the median absolute deviation from the median.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

MAD(*expression-1*[, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value of which the median absolute deviation from the median is to be computed.

Data type DOUBLE

Details

If all arguments have missing or null values, the result is a missing or null value. Otherwise, the result is the median absolute deviation from the median of the nonmissing or non-null values. The formula for the median is the same as the one that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide: Statistical Procedures.

Example

The following program illustrates the MAD function:

```
data test (overwrite=yes);
   dcl double c;
   method run();
```

```
      c=mad(2, 4, 1, 3, 5, 999999);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1.5
```

See Also

Functions:

- “IQR Function” on page 761
- “MEDIAN Function” on page 821
- “PCTL Function” on page 884

MARGRCLPRC Function

Calculates call prices for European options on stocks, based on the Margrabe model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

MARGRCLPRC(*X_1*, *t*, *X_2*, *sigma1*, *sigma2*, *rho12*)

Arguments

X_1
is a nonmissing, positive value that specifies the price of the first asset.

Requirement: Specify X_1 and X_2 in the same units.

Data type: DOUBLE

t
is a nonmissing value that specifies the time to expiration, in years.

Data type: DOUBLE

X_2
is a nonmissing, positive value that specifies the price of the second asset.

Requirement Specify X_2 and X_1 in the same units.

Data type DOUBLE

sigma1
is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

sigma2
is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

rho12
specifies the correlation between the first and second assets, $\rho_{x_1 x_2}$.

Range between –1 and 1

Data type DOUBLE

Details

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$\text{CALL} = X_1 N(d_1) - X_2 N(d_2)$$

Arguments

X_1
specifies the price of the first asset.

X_2
specifies the price of the second asset.

N
specifies the cumulative normal density function.

$$d_1 = \frac{\left(\ln\left(\frac{N_1}{N_2}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$
$$d_2 = d_1 - \sigma\sqrt{t}$$
$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1,x_2}\sigma_{x_1}\sigma_{x_2}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration.

$\sigma_{x_1}^2$
specifies the variance of the first asset.

$\sigma_{x_2}^2$
specifies the variance of the second asset.

σ_{x_1}
specifies the volatility of the first asset.

σ_{x_2}
specifies the volatility of the second asset.

ρ_{x_1, x_2}
specifies the correlation between the first and second assets.

For the special case of *t*=0, the following equation is true:

$$\text{CALL} = \max((X_1 - X_2), 0)$$

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following program illustrates the MARGRCLPRC function:

```
data test(overwrite=yes);
   dcl double a b;
   method run();
      a=margrclprc(15, .5, 13, .06, .05, 1);
      put a;
      b=margrclprc(2, .25, 1, .3, .2, 1);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
2
1
```

See Also

Functions:

- “MARGRPTPRC Function” on page 812

MARGRPTPRC Function

Calculates put prices for European options on stocks, based on the Margrabe model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

MARGRPTPRC(*X_1*, *t*, *X_2*, *sigma1*, *sigma2*, *rho12*)

Arguments

X_1
is a nonmissing, positive value that specifies the price of the first asset.

Requirement: Specify X_1 and X_2 in the same units.

Data type: DOUBLE

t
is a nonmissing value that specifies the time to expiration, in years.

Data type: DOUBLE

X_2
is a nonmissing, positive value that specifies the price of the second asset.

Requirement: Specify X_2 and X_1 in the same units.

Data type: DOUBLE

sigma1
is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

sigma2
is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

rho12
specifies the correlation between the first and second assets, $\rho_{x_1 y_1}$.

Range between –1 and 1

Data type DOUBLE

Details

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$\text{PUT} = X_2 N(pd_1) - X_1 N(pd_2)$$

Arguments

X_1
specifies the price of the first asset.

X_2
specifies the price of the second asset.

N
specifies the cumulative normal density function.

$$pd_1 = \frac{\left(\ln\left(\frac{N_1}{N_2}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$pd_2 = pd_1 - \sigma\sqrt{t}$$

$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1, x_2}\sigma_{x_1}\sigma_{x_2}$$

The following arguments apply to the preceding equation:

`t`
is a nonmissing value that specifies the time to expiration, in years.

$\sigma_{x_1}^2$
specifies the variance of the first asset.

$\sigma_{x_2}^2$
specifies the variance of the second asset.

σ_{x_1}
specifies the volatility of the first asset.

σ_{x2}
specifies the volatility of the second asset.

$\rho_{x1,\,x2}$
specifies the correlation between the first and second assets.

To view the corresponding CALL relationship, see the "MARGRCLPRC Function" on page 809.

For the special case of t=0, the following equation is true:

$$\text{PUT} = \max((X_2 - X_1), 0)$$

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see "Using Pricing Functions" in *SAS Functions and CALL Routines: Reference*.

Comparisons

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following program illustrates the MARGRPTPRC function:

```
data test(overwrite=yes);
   dcl double a b;
   method run();
      a=margrptprc(2, .25, 3, .06, .2, 1);
      put a;
      b=margrptprc(3, .25, 4, .05, .3, 1);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1.00000000009729
1.00157624907711
```

See Also

Functions:

■ “MARGRCLPRC Function” on page 809

MAX Function

Returns the largest value from a list of arguments.

Categories: CAS
Descriptive Statistics

Returned data type: BIGINT, DECIMAL, DOUBLE

Syntax

MAX(*expression-1, expression-2* [, ...*expression-n*])

Arguments

expression
is any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: BIGINT, DECIMAL, DOUBLE

See: “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Comparisons

The MAX function returns the largest value from a list of arguments. The MAX operator (<>) returns the largest of two operands.

The MAX function returns a null or missing value only if all arguments are null or missing. The MAX operator (<>) returns a null or missing value only if both

operands are null or missing. In this case, it returns the value of the operand that is higher in the sort order for null or missing values.

Example

The following program illustrates the MAX function:

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=max(8, 3);
      put x;
      x=max(2, 6, .);
      put x;
      x=max(2, -3, 1, -1);
      put x;
      x=max(3, ., -3);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
8
6
2
3
```

See Also

Functions:

- “MIN Function” on page 823

MD5 Function

Returns the result of the message digest of a specified string in binary format.

Categories:	CAS Character
Restriction:	When you are using character arguments, MD5 accepts CHAR with any character set, including the NATIONAL character set (NCHAR).
Returned data type:	BINARY

Syntax

MD5(*string*)

Arguments

string
specifies a character constant, variable, or expression.

Data type: BINARY, CHAR

Tips: Enclose a literal string of characters in single quotation marks.

For scalar character variable arguments, the initial character set encoding that is specified in the DECLARE statement is used to transcode the variable before it is passed to the MD5 function. For binary arguments, the binary value is treated as a character string and the session encoding is used to transcode the value it is passed to the MD5 function.

Details

The Basics

The MD5 function converts a string, based on the MD5 algorithm, into a 128-bit hash value. This hash value is referred to as a *message digest* (digital signature), and it is nearly unique for each string that is passed to the function.

The MD5 function does not format its own output. Use the $BINARY*w*. or the $HEX*w*. formats to view readable results.

The Message Digest Algorithm

A message digest results from manipulating and compacting an arbitrarily long stream of binary data. An ideal message digest algorithm never generates the same result for two different sets of input. However, generating such a unique result would require a message digest as long as the input itself. Therefore, MD5 generates a message digest of modest size (16 bytes), created with an algorithm that is designed to make a nearly unique result.

Using the MD5 Function

You can use the MD5 function to track changes in your tables. The MD5 function can generate a digest of a set of column values in a record in a table. This digest could be treated as the signature of the record, and be used to keep track of changes that are made to the record. If the digest from the new record matches the existing digest of a record in a table, then the two records are the same. If the digest is different, then a column value in the record has changed. The new changed

record could then be added to the table along with a new surrogate key because it represents a change to an existing keyed value.

The MD5 function can be useful when developing shell scripts or Perl programs for software installation, for file comparison, and for detection of file corruption and tampering.

You can also use the MD5 function to create a unique identifier for observations to be used as the key of a hash package. For more information, see "Using the Hash Package" in *SAS DS2 Programmer's Guide*.

Example

Here is an example of how to generate results that are returned by the MD5 function.

```
data _null_;
   method init();
   dcl binary(32) y z having format $hex32.;
      y = md5('abc');
      z = md5('access method');
      put y= ;
      put z= ;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
y=900150983CD24FB0D6963F7D28E17F72
z=53128C19421A8E0C7F6436D06A026537
```

MDY Function

Returns a SAS date value from month, day, and year values.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

MDY(*month*, *day*, *year*)

Arguments

month

specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

day

specifies a numeric expression that represents a whole number from 1 through 31.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

year

specifies a numeric expression that represents a two-digit or four-digit year. The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following programs illustrate the MDY function:

```
data test (overwrite=yes);
   dcl double d m y;
   dcl double birthday having format mmddyy10.;
   method run();
       d=8; m=27; y=19;
       birthday= mdy(d,m,y);
       put birthday;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
08/27/2019
```

```
data test2 (overwrite=yes);
   dcl double d m y;
   dcl double anniversary having format date9.;
   method run();
       d=7; m=11; y=19;
       anniversary= mdy(d,m,y);
       put anniversary;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log:

```
11JUL2019
```

See Also

Concepts:

- "Dates and Times in DS2" in *SAS DS2 Programmer's Guide*

Functions:

- "DAY Function" on page 505
- "MONTH Function" on page 834
- "YEAR Function" on page 1185

MEAN Function

Returns the arithmetic mean (average) of the non-null or nonmissing arguments.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

MEAN(*expression-1*[, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Comparisons

The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, whereas the MEAN function returns the arithmetic mean (average).

Example

The following program illustrates the MEAN function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3;
   method run();
      x1=mean(2, ., ., 6);
      put x1;
      x2=mean(1, 2, 3, 2);
      put x2;
      x3=mean(of x1-x2);
      put x3;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
4
2
3
```

See Also

Functions:

- “GEOMEAN Function” on page 666
- “GEOMEANZ Function” on page 668
- “HARMEAN Function” on page 670
- “HARMEANZ Function” on page 672
- “MEDIAN Function” on page 821

MEDIAN Function

Returns the median value.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

MEDIAN(*expression-1*[, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The MEDIAN function returns the median of the nonmissing or nonnull values. If all arguments have missing or null values, the result is a missing or null value.

Note: The formula that is used in the MEDIAN function is the same as the formula that is used in PROC UNIVARIATE in Base SAS Procedures Guide: Statistical Procedures. For more information, see SAS Elementary Statistics Procedures.

Comparisons

The MEDIAN function returns the median of nonmissing or nonnull values, whereas the MEAN function returns the arithmetic mean (average).

Example

The following program illustrates the MEDIAN function:

```
data test (overwrite=yes);
   dcl double x y;
   method run();
      x=median(2, 4, 1, 3);
      put x;
      y=median(5, 8, 0, 3, 4);
      put y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
2.5
4
```

See Also

Functions:

- "MEAN Function" on page 820

MIN Function

Returns the smallest value.

Categories: CAS
Descriptive Statistics

Returned data type: BIGINT, DECIMAL, DOUBLE

Syntax

MIN(*expression-1, expression-2* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: BIGINT, DECIMAL, DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Comparisons

The MIN function returns the smallest value from a list of values. The MIN operator (><) returns the smallest value of two operands.

The MIN function returns a null or missing value only if all arguments are null or missing. The MIN operator returns a null or missing value only if either operand is null or missing. In this case, it returns the value of the operand that is lower in the sort order for null or missing values.

Example

The following program illustrates the MIN function:

```
data test (overwrite=yes);
   dcl double x x1 x2 x3 x4 x5;
   method run();
      x=min(7,4);
      put x;
      x1=min(2, ., 6);
      put x1;
      x2=min(2, -3, 1, -1);
      put x2;
      x3=min(0, 4);
      put x3;
      x4=min(of x1-x3);
      put x4;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
4
2
-3
0
-3
```

MINUTE Function

Returns the minute from a SAS time or datetime value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

MINUTE(*time* | *datetime*)

Arguments

time
specifies any valid expression that represents a SAS time value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The MINUTE function returns a whole number that represents a specific minute of the hour. MINUTE always returns a positive number in the range of 0 through 59. Null or missing values are ignored.

Example

The following programs illustrate the MINUTE function:

```
data _null_;
   dcl double d;
   method init();
      d=minute(time());
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
38
```

```
data test(overwrite=yes);
   dcl double m dt;
   dcl time t;
   method run();
      t=time '03:19:24';
      dt=to_double(t);
      m=minute(dt);
```

```
      put m;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
19
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "HOUR Function" on page 682
- "SECOND Function" on page 1071

MISSING Function

Returns a number that indicates whether the argument contains a missing value.

Categories:	CAS Special
Returned data type:	INTEGER

Syntax

MISSING(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a value.

Data type	All data types
Note	If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2_SASMISSING environment variable is set to TRUE.
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

- The MISSING function checks if a value is a null or missing value and returns a numeric result. If the argument does not contain a missing value, SAS returns a value of 0. If the argument contains a missing value, SAS returns a value of 1.
- In SAS mode, a blank-filled character value is defined to be the SAS missing value. In ANSI mode, a blank-filled character value is defined as nonmissing and non-null.
- In SAS mode, a DOUBLE value could be a SAS missing value (., .A through .Z). The other numeric types do not support SAS missing values.
- The MISSING function returns a 1 if a package instance does not exist. That is, the package variable is a missing package reference. The MISSING function returns a 0 if the package variable references a package instance.

Comparisons

The MISSING function can have only one argument. The NMISS function requires numeric arguments and returns the number of missing values in the list of arguments.

Note: Missing values and null values are treated differently in SAS mode versus ANSI mode. Missing and null values might be converted dependent on mode.

Examples

Example 1: Using the MISSING Function

The following program illustrates the MISSING function.

```
data test(overwrite=yes);
   dcl int a[3];
   dcl double i;
   method run();
      a[1]=2;
      a[2]=4;
      a[3]=.;
      do i = 1 to 3;
         if missing(a[i]) then put 'Missing';
            else put 'Not Missing';
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
Not Missing
Not Missing
Missing
```

Example 2: MISSING Function with SAS Mode and ANSI Mode

This program illustrates how a DS2 program with a MISSING function can return different results based on mode.

```
data test(overwrite=yes);
   dcl char(1) a[3];
   dcl double  b[3];
   dcl int c[3];
   dcl int i;
   method run();
      a := ('a', '', NULL);
      b := (1, ., NULL);
      c := (1, NULL, NULL);
      do i = 1 to 3;
         if (missing(a[i])) then put a[i]= 'missing';
            else put a[i]= 'not missing';
          if (missing(b[i])) then put b[i]= 'missing';
            else put b[i]= 'not missing';
          if (missing(c[i])) then put c[i]= 'missing';
            else put c[i]= 'not missing';
      end;
   end;
enddata;
run;
```

In SAS mode, the following lines are written to the SAS log.

```
a[1]=a not missing
b[1]=1 not missing
c[1]=1 not missing
a[2]=  missing
b[2]=. missing
c[2]= missing
a[3]=  missing
b[3]=. missing
c[3]= missing
```

In ANSI mode, the following lines are written to the SAS log.

```
a[1]=a not missing
b[1]=1 not missing
c[1]=1 not missing
a[2]=  not missing
b[2]= missing
c[2]= missing
a[3]= missing
b[3]= missing
c[3]= missing
```

See Also

- "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*

Functions:

- "CMISS Function" on page 425
- "NMISS Function" on page 842
- "N Function" on page 837
- "NULL Function" on page 878

MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

MOD(*dividend-expression*, *divisor-expression*)

Arguments

dividend-expression
specifies a dividend that is any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

divisor-expression
specifies a divisor that is any valid expression that evaluates to a numeric value.

Restriction *divisor-expression* cannot be 0

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The MOD function returns the remainder from the division of *dividend-expression* by *divisor-expression*. When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MOD function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representations.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. In this case, the following occurs.

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more. In this case, SAS also writes an error message to the log.

Comparisons

Here are some comparisons between the MOD and MODZ functions:

- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- The MODZ function performs no fuzzing.
- Both the MOD and MODZ functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

The following program illustrates the MOD and MODZ functions:

```
data test(overwrite=yes);
   dcl double x1 x2 x3  xa xb xc having format 9.4;
   dcl double x4 xd having format 24.20;
   method run();
      x1=mod(10, 3);
      put x1=;
      xa=modz(10, 3);
      put xa=;
      x2=mod(.3, -.1);
      put x2=;
      xb=modz(.3, -.1);
```

```
        put xb=;
        x3=mod(1.7, .1);
        put x3=;
        xc=modz(1.7, .1);
        put xc=;
        x4=mod(.9, .3);
        put x4=;
        xd=modz(.9, .3);
        put xd=;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=   1.0000
xa=   1.0000
x2=   0.0000
xb=   0.1000
x3=   0.0000
xc=   0.0000
x4=  0.0000000000000000000
xd=  0.0000000000000011102
```

See Also

Functions:

- “MODZ Function” on page 831
- “INT Function” on page 694
- “INTZ Function” on page 757

MODZ Function

Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

MODZ(*dividend-expression*, *divisor-expression*)

Arguments

dividend-expression
specifies a dividend that is any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

divisor-expression
specifies a divisor that is any valid expression that evaluates to a numeric value.

Restriction *divisor-expression* cannot be 0

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The MODZ function returns the remainder from the division of *dividend-expression* by *divisor-expression*. When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MODZ function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representation.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. For example, when you use exact arithmetic and the result is zero, MODZ might return a very small positive value or a value slightly less than the second argument.

Comparisons

Here are some comparisons between the MODZ and MOD functions:

- The MODZ function performs no fuzzing.
- The MOD function performs fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- Both the MODZ and MOD functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

The following statements illustrate the MOD and MODZ functions:

```
data test(overwrite=yes);
   dcl double x1 x2 x3  xa xb xc having format 9.4;
   dcl double x4 xd having format 24.20;
   method run();
      x1=mod(10, 3);
      put x1=;
      xa=modz(10, 3);
      put xa=;
      x2=mod(.3, -.1);
      put x2=;
      xb=modz(.3, -.1);
      put xb=;
      x3=mod(1.7, .1);
      put x3=;
      xc=modz(1.7, .1);
      put xc=;
      x4=mod(.9, .3);
      put x4=;
      xd=modz(.9, .3);
      put xd=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=   1.0000
xa=   1.0000
x2=   0.0000
xb=   0.1000
x3=   0.0000
xc=   0.0000
x4=  0.00000000000000000000
xd=  0.00000000000000011102
```

See Also

Functions:

- “INT Function” on page 694
- “INTZ Function” on page 757
- “MOD Function” on page 829

MONTH Function

Returns a number that represents the month from a SAS date value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

MONTH(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Range 1–12

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the MONTH function when the month is March:

```
data test(overwrite=yes);
   dcl double d;
   method init();
      d=month(date());
      put d;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
3
```

The following program illustrates the MONTH function when the date is February 5, 2019:

```
data test(overwrite=yes);
   dcl double m dat;
```

```
   dcl date d;
   method run();
      d=date '2019-02-05';
      dat=to_double(d);
      m=month(dat);
      put m;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
2
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "DAY Function" on page 505
- "YEAR Function" on page 1185

MORT Function

Returns amortization parameters.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

MORT(*a*, *p*, *r*, *n*)

Arguments

a

specifies any valid expression that evaluates to the initial amount.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

p

specifies any valid expression that evaluates to the periodic payment.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

r

specifies any valid expression that evaluates to the periodic interest rate that is expressed as a fraction.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

n

specifies any valid expression that evaluates to the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

Calculating Results

The MORT function returns the missing argument in the list of four arguments from an amortization calculation with a fixed interest rate that is compounded each period. The arguments are related by the following equation:

$$p = \frac{ar(1+r)^n}{(1+r)^n - 1}$$

One missing argument must be provided. The value is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Restrictions in Calculating Results

The MORT function returns an invalid argument note to the SAS log and sets _ERROR_ to 1 if one of the following argument combinations is true:

- rate < –1 or n < 0
- principal <= 0 or payment <= 0 or n <= 0
- principal <= 0 or payment <= 0 or rate <= –1
- principal * rate > payment
- principal > payment * n

Example

In the following program, an amount of $50,000 is borrowed for 30 years at an annual interest rate of 10% compounded monthly.

```
data test (overwrite=yes);
   dcl double payment;
   method run();
      payment=mort(50000, . , .10/12, 30*12);
      put payment;
   end;
enddata;
run;
```

The value that is returned is 438.79 (rounded). The second argument is set to missing, which indicates that the periodic payment is to be calculated. The 10% nominal annual rate has been converted to a monthly rate of 0.10/12. The rate is the fractional (not the percentage) interest rate per compounding period. The 30 years are converted to 360 months.

N Function

Returns the number of non-null or nonmissing numeric values.

Categories: CAS
Special

Returned data type: INTEGER

Syntax

N(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one argument is required.

Data type: DECIMAL, DOUBLE, NUMERIC

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

Null values are converted to missing values and are counted as missing values.

Comparisons

The N function counts non-null and nonmissing values, whereas the NMISS function counts missing values. The N function requires numeric arguments.

Example

The following program illustrates the N function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3;
   method run();
      x1=n(1, 0,., 2, 5, .);
      x2=n(1, 2);
      x3=n(of x1-x2);
      put x1= x2= x3=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=4 x2=2 x3=2
```

See Also

Functions:

- “NMISS Function” on page 842

NDIMS Function

Returns the number of dimensions in an array.

Categories: Array
CAS

Returned data type: INT

Syntax

NDIMS(*array-name*)

Arguments

array-name
specifies the name of a temporary or a variable array.

Details

The NDIMS function returns the number of dimensions of a multidimensional array, or returns 1 for a one-dimensional array.

Comparisons

- DIM returns the number of elements in an array dimension.
- HBOUND returns the value of the upper bound of an array dimension.
- LBOUND returns the value of the lower bound of an array dimension.
- NDIMS returns the number of dimensions in an array.

Example

The following program shows how to use the DIM, HBOUND, LBOUND, and NDIMS array functions:

```
data test(overwrite=yes);
   dcl char(15) a1[4];
   dcl double   a2[2,3,4] sum i j k numelems;
   method init();
      a1 := ('red' 'yellow' 'green' 'blue');
      a2 := (24*2.0);
      do i = 1 to dim(a1);
         put a1[i];
      end;
      numelems = 0;
      do i = 1 to ndims(a2);
         numelems = numelems + dim(a2, i);
      end;
      sum = 0;
      do i = lbound(a2, 1) to hbound(a2, 1);
        do j = lbound(a2, 2) to hbound(a2, 2);
          do k = lbound(a2, 3) to hbound(a2, 3);
            sum = sum + a2[i,j,k];
          end;
        end;
```

```
        end;
        put sum=;
    end;
run;
```

SAS writes the following output to the log:

```
red
yellow
green
blue
sum=48
```

See Also

Functions:

- “DIM Function” on page 521
- “HBOUND Function” on page 674
- “LBOUND Function” on page 776

NETPV Function

Returns the net present value as a percent.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

NETPV(*r*, *freq*, *c0, c1, ..., cn*)

Arguments

r

is numeric, the interest rate over a specified base period of time expressed as a fraction.

Range *r* >= 0

Data type DOUBLE

freq
is numeric, the number of payments during the base period of time that is specified with the rate *r*.

Range *freq* > 0

Data type DOUBLE

Note The case *freq* = 0 is a flag to allow continuous discounting.

c0, c1 ,..., cn
are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ...n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The *c0* argument and the *c1* argument are required.

Data type DOUBLE

Details

The NETPV function returns the net present value at time 0 for the set of cash payments *c0*,*c1*, ...,*cn*, with a rate *r* over a specified base period of time. The argument *freq*>0 describes the number of payments that occur over the specified base period of time.

The net present value is given by the equation:

$$\mathrm{NETPV}(r, freq, c_0, c_1, ..., c_n) = \sum_{i=0}^{n} c_i x^i$$

The following relationship applies to the preceding equation:

$$x = \begin{cases} \frac{1}{(1+r)^{(1/freq)}} & freq > 0 \\ \varepsilon^{-r} & freq = 0 \end{cases}$$

Missing values in the payments are treated as 0 values. When *freq*>0, the rate *r* is the effective rate over the specified base period. To compute with a quarterly rate (the base period is three months) of 4% with monthly cash payments, set *freq* to 3 and set *r* to .04.

If *freq* is 0, continuous discounting is assumed. The base period is the time interval between two consecutive payments, and the rate *r* is a nominal rate.

To compute with a nominal annual interest rate of 11% discounted continuously with monthly payments, set *freq* to 0 and set *r* to .11/12.

Example

For an initial investment of $500 that returns biannual payments of $200, $300, and $400 over the succeeding 6 years and an annual discount rate of 10%, the net present value of the investment can be expressed as follows:

```
data test(overwrite=yes);
   dcl double value;
   method run();
      value=netpv(.10,.5,-500,200,300,400);
      put value;
   end;
enddata;
run;
```

The value that is returned is 95.982864829379.

See Also

Functions:

- “NPV Function” on page 876

NMISS Function

Returns the number of null and SAS missing numeric values.

Categories: CAS
Descriptive Statistics

Returned data type: INTEGER

Syntax

NMISS(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one argument is required.

Data type: DECIMAL, DOUBLE, NUMERIC

See: “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

Null values are converted to SAS missing values and are counted as missing values.

Comparisons

The NMISS function returns the number of null or SAS missing values, whereas the N function returns the number of non-null and nonmissing values. NMISS requires numeric values and works with multiple numeric values, whereas MISSING works with only one value that can be either numeric or character.

Example

The following program illustrates the NMISS function:

```
data test (overwrite=yes);
   dcl double x1 x2 x3;
   method run();
      x1=nmiss(1,0,.,2,5,.);
      x2=nmiss(1,0);
      x3=nmiss(of x1-x2);
      put x1= x2= x3=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=2 x2=0 x3=0
```

See Also

- "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*

Functions:

- "CMISS Function" on page 425
- "MISSING Function" on page 826
- "N Function" on page 837
- "NULL Function" on page 878

NOMRATE Function

Returns the nominal annual interest rate.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

NOMRATE(*compounding-interval*, *rate*)

Arguments

compounding-interval
is a SAS interval. This value represents how often the returned value is compounded.

Data type CHAR

rate
is numeric. *Rate* is the effective annual interest rate (expressed as a percentage) that is compounded at each interval.

Data type DOUBLE

Details

The NOMRATE function returns the nominal annual interest rate. NOMRATE computes the nominal annual interest rate that corresponds to an effective annual interest rate.

The following details apply to the NOMRATE function:

- The values for rates must be at least –99.
- In considering an effective interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by NOMRATE equals $\log_e(1+[rate/100])$.

  If *compounding-interval* is not 'CONTINUOUS', and *m* intervals occur in a year, the value that is returned by NOMRATE equals the following:

$$m\left((1+\frac{rate}{100})^{\frac{1}{m}}-1\right)$$

- The following values are valid for *compounding-interval*:
  - 'CONTINUOUS'
  - 'DAY'
  - 'SEMIMONTH'
  - 'MONTH'
  - 'QUARTER'
  - 'SEMIYEAR'
  - 'YEAR'
- If the interval is 'DAY', then *m*=365.

Example

- If an effective rate is 10% when compounded monthly, the corresponding nominal rate can be expressed as follows:

```
data test(overwrite=yes);
   dcl double effective_rate1;
   method run();
      effective_rate1 = NOMRATE('MONTH', 10);
      put effective_rate1=;
   end;
enddata;
run;
```

  SAS writes the following output to the log.

```
effective_rate1=9.56896851468451
```

- If an effective rate is 10% when compounded quarterly, the corresponding nominal rate can be expressed as follows:

```
data test(overwrite=yes);
   dcl double effective_rate2;
   method run();
      effective_rate2 = NOMRATE('QUARTER', 10);
      put effective_rate2=;
   end;
enddata;
run;
```

  SAS writes the following output to the log.

```
effective_rate2=9.64547563377804
```

NOTALNUM Function

Searches a character string for a non-alphanumeric character, and returns the first character position at which the character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTALNUM('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTALNUM function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTALNUM function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase letter. If such a character is found, NOTALNUM returns the position in the string of that character. If no such character is found, NOTALNUM returns a value of 0.

If you use only one argument, NOTALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALNUM function searches a character string for a non-alphanumeric character. The ANYALNUM function searches a character string for an alphanumeric character.

Example

The following program uses the NOTALNUM function to search a string from left to right for non-alphanumeric characters.

```
data _null_;
   dcl nchar(16) string c;
   dcl double j i;
   method run();
      string='Next = Last + 1;';
      j=0;
      do until(j=0);
         j=notalnum(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=6 c==
j=7 c=
j=12 c=
j=13 c=+
j=14 c=
j=16 c=;
The end
```

See Also

Functions:

■ “ANYALNUM Function” on page 292

NOTALPHA Function

Searches a character string for a nonalphabetic character, and returns the first character position at which the character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTALPHA(‘*expression*’[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTALPHA function depend directly on the translation table that is in effect (see “TRANTAB= System Option” in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTALPHA function searches a string for the first occurrence of any character that is not an uppercase or lowercase letter. If such a character is found, NOTALPHA returns the position in the string of that character. If no such character is found, NOTALPHA returns a value of 0.

If you use only one argument, NOTALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALPHA function searches a character string for a nonalphabetic character. The ANYALPHA function searches a character string for an alphabetic character.

Examples

Example 1: Searching a String for Nonalphabetic Characters

The following program uses the NOTALPHA function to search a string from left to right for nonalphabetic characters.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notalpha(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
j=18 c=;
The end
```

Example 2: Identifying Control Characters By Using the NOTALPHA Function

You can execute the following program to show the control characters that are identified by the NOTALPHA function.

```
data test;
   dcl nchar(3) byte1 hex1;
   dcl double dec notalpha1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         notalpha1=notalpha(byte1);
         output;
      end;
   end;
enddata;
run;
```

See Also

Functions:

- “ANYALPHA Function” on page 295

NOTCNTRL Function

Searches a character string for a character that is not a control character, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

NOTCNTRL('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTCNTRL function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTCNTRL function searches a string for the first occurrence of a character that is not a control character. If such a character is found, NOTCNTRL returns the position in the string of that character. If no such character is found, NOTCNTRL returns a value of 0.

If you use only one argument, NOTCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTCNTRL returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTCNTRL function searches a character string for a character that is not a control character. The ANYCNTRL function searches a character string for a control character.

Example

You can execute the following program to show the control characters that are identified by the NOTCNTRL function.

```
data test (overwrite=yes);
   dcl double dec notcntrl1;
   dcl char byte1 hex1;
   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec, hex2.);
         notcntrl1=notcntrl(byte1);
         output;
      end;
   end;
enddata;
run;
```

See Also

Functions:

- “ANYCNTRL Function” on page 298

NOTDIGIT Function

Searches a character string for any character that is not a digit, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

NOTDIGIT('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTDIGIT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTDIGIT function searches a string for the first occurrence of any character that is not a digit. If such a character is found, NOTDIGIT returns the position in the string of that character. If no such character is found, NOTDIGIT returns a value of 0.

If you use only one argument, NOTDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTDIGIT function searches a character string for any character that is not a digit. The ANYDIGIT function searches a character string for a digit.

Example

The following program uses the NOTDIGIT function to search for a character that is not a digit.

```
data _null_;
   dcl nchar(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notdigit(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=16 c=E
j=18 c=;
The end
```

See Also

Functions:

- “ANYDIGIT Function” on page 300

NOTFIRST Function

Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTFIRST('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are any except the underscore (_) and uppercase or lowercase English letters. If such a character is found, NOTFIRST returns the position in the string of that character. If no such character is found, NOTFIRST returns a value of 0.

If you use only one argument, NOTFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*,

specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Example

The following program uses the NOTFIRST function to search a string for any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

```
data _null_;
   dcl nchar(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notfirst(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=6 c==
j=7 c=
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
j=18 c=;
The end
```

See Also

Functions:

- "ANYFIRST Function" on page 302

NOTGRAPH Function

Searches a character string for a non-graphical character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTGRAPH('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTGRAPH function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTGRAPH function searches a string for the first occurrence of a non-graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, NOTGRAPH returns the position in the string of that character. If no such character is found, NOTGRAPH returns a value of 0.

If you use only one argument, NOTGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTGRAPH function searches a character string for a non-graphical character. The ANYGRAPH function searches a character string for a graphical character.

Examples

Example 1: Searching a String for Non-Graphical Characters

The following program uses the NOTGRAPH function to search a string for a non-graphical character.

```
data _null_;
   dcl nchar(18) string c;
   dcl double j i;
```

```
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notgraph(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=7 c=
j=11 c=
j=13 c=
The end
```

Example 2: Identifying Control Characters By Using the NOTGRAPH Function

You can execute the following program to show the control characters that are identified by the NOTGRAPH function.

```
data test (overwrite=yes);
   dcl nchar byte1 hex1;
   dcl double dec notgraph1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec,hex2.);
         notgraph1=notgraph(byte1);
         output;
      end;
   end;
enddata;
run;
```

See Also

Functions:

- “ANYGRAPH Function” on page 304

NOTLOWER Function

Searches a character string for a character that is not a lowercase letter, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTLOWER('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTLOWER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTLOWER function searches a string for the first occurrence of any character that is not a lowercase letter. If such a character is found, NOTLOWER returns the position in the string of that character. If no such character is found, NOTLOWER returns a value of 0.

If you use only one argument, NOTLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument,

start, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTLOWER function searches a character string for a character that is not a lowercase letter. The ANYLOWER function searches a character string for a lowercase letter.

Example

The following program uses the NOTLOWER function to search a string for any character that is not a lowercase letter.

```
data _null_;
   dcl nchar(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notlower(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

See Also

Functions:

- “ANYLOWER Function” on page 307

NOTNAME Function

Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTNAME(‘*expression*’[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. These characters are any except underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, NOTNAME returns the position in the string of that character. If no such character is found, NOTNAME returns a value of 0.

If you use only one argument, NOTNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7.

Example

The following program uses the NOTNAME function to search a string for any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

```
data _null_;
   dcl nchar(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notname(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=6 c==
j=7 c=
j=11 c=
j=12 c=+
j=13 c=
j=18 c=;
The end
```

See Also

Functions:

- “ANYNAME Function” on page 309

NOTPRINT Function

Searches a character string for a nonprintable character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTPRINT(‘*expression*’[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTPRINT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTPRINT function searches a string for the first occurrence of a non-printable character. If such a character is found, NOTPRINT returns the position in the string of that character. If no such character is found, NOTPRINT returns a value of 0.

If you use only one argument, NOTPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTPRINT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTPRINT function searches a character string for a non-printable character. The ANYPRINT function searches a character string for a printable character.

Example

You can execute the following program to show the control characters that are identified by the NOTPRINT function.

```
data test (overwrite=yes);
   dcl double dec notprint1;
   dcl nchar byte1 hex1;
   method run();
      do dec=0 to 255;
         byte1=byte(dec);
         hex1=put(dec, hex2.);
         notprint1=notprint(byte1);
         output;
      end;
   end;
enddata;
run;
```

See Also

Functions:

- “ANYPRINT Function” on page 311

NOTPUNCT Function

Searches a character string for a character that is not a punctuation character, and returns the first character position at which that character is found.

Categories:	CAS Character
Returned data type:	DOUBLE

Syntax

NOTPUNCT(’*expression*’[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTPUNCT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTPUNCT function searches a string for the first occurrence of a character that is not a punctuation character. If such a character is found, NOTPUNCT returns the position in the string of that character. If no such character is found, NOTPUNCT returns a value of 0.

If you use only one argument, NOTPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTPUNCT function searches a character string for a character that is not a punctuation character. The ANYPUNCT function searches a character string for a punctuation character.

Examples

Example 1: Searching a String for Characters That Are Not Punctuation Characters

The following program uses the NOTPUNCT function to search a string for characters that are not punctuation characters.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notpunct(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=9 c=n
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
The end
```

Example 2: Identifying Control Characters By Using the NOTPUNCT Function

You can execute the following program to show the control characters that are identified by the NOTPUNCT function.

```
data test;
   dcl nchar(3) byte1 hex1;
   dcl double dec notpunct1;

   method run();
      do dec=0 to 255;
```

```
            byte1=byte(dec);
            hex1=put(dec,hex2.);
            notpunct1=notpunct(byte1);
            output;
         end;
      end;
enddata;
run;
quit;
 proc print data=test;
 run;
```

See Also

Functions:

- "ANYPUNCT Function" on page 314

NOTSPACE Function

Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTSPACE('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTSPACE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTSPACE function searches a string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, NOTSPACE returns the position in the string of that character. If no such character is found, NOTSPACE returns a value of 0.

If you use only one argument, NOTSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Examples

Example 1: Searching a String for a Character That Is Not a Whitespace Character

The following program uses the NOTSPACE function to search a string for a character that is not a whitespace character.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notspace(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c=+
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

Example 2: Identifying Control Characters By Using the NOTSPACE Function

You can execute the following program to show the control characters that are identified by the NOTSPACE function.

```
data test (overwrite=yes);
   dcl nchar(3) byte1 hex1;
   dcl double dec notspace1;

   method run();
      do dec=0 to 255;
         byte1=byte(dec);
```

```
            hex1=put(dec,hex2.);
            notspace1=notspace(byte1);
            output;
         end;
      end;
   enddata;
   run;
```

See Also

Functions:

- "ANYSPACE Function" on page 317

NOTUPPER Function

Searches a character string for a character that is not an uppercase letter, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTUPPER('*expression*'[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTUPPER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The NOTUPPER function searches a string for the first occurrence of a character that is not an uppercase letter. If such a character is found, NOTUPPER returns the position in the string of that character. If no such character is found, NOTUPPER returns a value of 0.

If you use only one argument, NOTUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTUPPER function searches a character string for a character that is not an uppercase letter. The ANYUPPER function searches a character string for an uppercase letter.

Example

The following program uses the NOTUPPER function to search a string for any character that is not an uppercase letter.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notupper(string, j+1);
         if j=0 then put 'The end';
         else do;
```

```
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
j=18 c=;
The end
```

See Also

Functions:

- “ANYUPPER Function” on page 319

NOTXDIGIT Function

Searches a character string for a character that is not a hexadecimal character, and returns the first character position at which that character is found.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

NOTXDIGIT(‘*expression*’[, *start*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTXDIGIT function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, NOTXDIGIT returns the position in the string of that character. If no such character is found, NOTXDIGIT returns a value of 0.

If you use only one argument, NOTXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character. The ANYXDIGIT function searches a character string for a character that is a hexadecimal character.

Example

The following program uses the NOTXDIGIT function to search a string for a character that is not a hexadecimal character.

```
data _null_;
   dcl char(18) string c;
   dcl double j i;
   method run();
      string='Next = _n_ + 12E3;';
      j=0;
      do until(j=0);
         j=notxdigit(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            put j= c=;
         end;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=18 c=;
The end
```

See Also

Functions:

- “ANYXDIGIT Function” on page 321

NPV Function

Returns the net present value with the rate expressed as a percentage.

Categories: CAS

Financial

Returned data type: DOUBLE

Syntax

NPV(*r*, *freq*, *c0*, *c1*, ..., *cn*)

Arguments

r
is numeric, the interest rate over a specified base period of time expressed as a percentage.

Data type DOUBLE

freq
is numeric, the number of payments during the base period of time specified with the rate *r*.

Range *freq* > 0

Data type DOUBLE

Note The case *freq* = 0 is a flag to allow continuous discounting.

c0, c1, ..., cn
are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ...n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The *c0* argument and the *c1* argument are required.

Data type DOUBLE

Comparisons

The NPV function is identical to NETPV, except that the *r* argument is provided as a percentage.

See Also

Functions:

- "NETPV Function" on page 840

NULL Function

Returns a 1 if the argument is null and a 0 if the argument is not null.

Categories: CAS
Special

Returned data type: INTEGER

Syntax

NULL(*expression*)

Arguments

expression
specifies any valid expression.

Data type: All data types

Note: If you are using SAS Federation Server, ANSI null values are translated to SAS missing values in FedSQL CALL invocations when the DS2_SASMISSING environment variable is set to TRUE.

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The NULL function returns a `1` only for a null value. It returns a `0` for any non-null value, including a SAS missing value.

The NULL function returns a `1` if a package instance does not exist, that is the package variable is a null package reference. The NULL function returns a `0` if the package variable references a package instance.

Note: Missing values and null values are treated differently in SAS mode versus ANSI mode. Missing and null values might be converted dependent on mode.

Example

The following program illustrates how NULL can differ in SAS mode and in ANSI mode.

In SAS mode, the following lines are written to the SAS log.

```
data test(overwrite=yes);
   dcl char(1) a[3];
   dcl double  b[3];
   dcl int c[3];
   dcl int i;
   method run();
      a := ('a', '', NULL);
      b := (1, ., NULL);
      c := (1, NULL, NULL);
      do i = 1 to 3;
         if (null(a[i])) then put a[i]= null;
            else put a[i]= 'not null;
         if (null(b[i])) then put b[i]= null;
            else put b[i]= 'not null;
         if (null(c[i])) then put c[i]= null;
            else put c[i]= 'not null;
      end;
   end;
enddata;
run;
```

```
a[1]=a not null
b[1]=1 not null
c[1]=1 not null
a[2]=  not null
b[2]=. not null
c[2]= null
a[3]=  not null
b[3]=. not null
c[3]= null
```

In ANSI mode, the following lines are written to the SAS log.

```
a[1]=a not null
b[1]=1 not null
c[1]=1 not null
a[2]=  not null
b[2]= null
c[2]= null
a[3]= null
b[3]= null
c[3]= null
```

Note that in ANSI mode, **b[2]** is null because the SAS missing value (.) is converted to null before being assigned to **b[2]** in `b := (1, ., NULL);`. In SAS mode, **a[3]** and b[3] are not null because the null value is converted to a SAS missing value (blank-filled string for **a[3]** and missing . for **b[3]**) before being assigned to **a[3]** and **b[3]** in `if (null(a[i])) then put a[i]= 'null';` and `else put a[i]= 'not null';`

See Also

- "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*

Functions:

- "MISSING Function" on page 826
- "N Function" on page 837
- "NMISS Function" on page 842

NWKDOM Function

Returns the date for the *n*th occurrence of a weekday for the specified month and year.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

NWKDOM(*n*, *weekday*, *month*, *year*)

Arguments

n
specifies the numeric week of the month that contains the specified day.

Range 1–5

Data type DOUBLE

Tip *N*=5 indicates that the specified day occurs in the last week of that month. Sometimes *n*=4 and *n*=5 produce the same results.

weekday
specifies the number that corresponds to the day of the week.

Range 1–7

Data type DOUBLE

Tip Sunday is considered the first day of the week and has a *weekday* value of 1.

month
specifies the number that corresponds to the month of the year.

Range 1–12

Data type DOUBLE

year
specifies a four-digit calendar year.

Data type DOUBLE

Details

The NWKDOM function returns a SAS date value for the *n*th weekday of the month and year that you specify. Use any valid SAS date format, such as the DATE9. format, to display a calendar date. You can specify *n*=5 for the last occurrence of a particular weekday in the month.

Sometimes *n*=5 and *n*=4 produce the same result. These results occur when there are only four occurrences of the requested weekday in the month. For example, if the month of January begins on a Sunday, there are five occurrences of Sunday, Monday, and Tuesday, but only four occurrences of Wednesday, Thursday, Friday, and Saturday. In this case, specifying *n*=5 or *n*=4 for Wednesday, Thursday, Friday, or Saturday produces the same result.

If the year is not a leap year, February has 28 days and there are four occurrences of each day of the week. In this case, *n*=5 and *n*=4 produce the same results for every day.

Comparisons

In the NWKDOM function, the value for *weekday* corresponds to the numeric day of the week beginning on Sunday. This value is the same value that is used in the WEEKDAY function, where Sunday =1, and so on. The value for *month* corresponds to the numeric month of the year beginning in January. This value is the same value that is used in the MONTH function, where January =1, and so on.

You can use the NWKDOM function to calculate events that are not defined by the HOLIDAY function. For example, if a university always schedules graduation on the first Saturday in June, then you can use the following statement to calculate the date: `UnivGrad = nwkdom(1, 7, 6, year);`

Examples

Example 1: Returning Date Values

The following program uses the NWKDOM function and returns the date for specific occurrences of a weekday for a specified month and year.

```
data _null_;
   dcl double a b c d e f;
   method run();
      /* Return the date of the third Monday in May 2012. */
      a=nwkdom(3, 2, 5, 2012);
      /* Return the date of the fourth Wednesday in November 2012. */
      b=nwkdom(4, 4, 11, 2012);
      /* Return the date of the fourth Saturday in November 2012. */
      c=nwkdom(4, 7, 11, 2012);
      /* Return the date of the first Sunday in January 2013. */
      d=nwkdom(1, 1, 1, 2013);
      /* Return the date of the second Tuesday in September 2012. */
      e=nwkdom(2, 3, 9, 2012);
      /* Return the date of the fifth Thursday in December 2012. */
      f=nwkdom(5, 5, 12, 2012);
      put a= weekdatx.;
      put b= weekdatx.;
      put c= weekdatx.;
      put d= weekdatx.;
      put e= weekdatx.;
      put f= weekdatx.;
   end;
enddata;
run;
```

SAS writes the following output to the log.

```
a=          Monday, 21 May 2012
b=  Wednesday, 28 November 2012
c=   Saturday, 24 November 2012
d=       Sunday, 6 January 2013
e=   Tuesday, 11 September 2012
f=   Thursday, 27 December 2012
```

Example 2: Returning the Date of the Last Monday in May

The following program returns the date that corresponds to the last Monday in the month of May in the year 2012.

```
data _null_;
   dcl double x;
   method init();
      /* The last Monday in May. */
      x=nwkdom(5,2,5,2012);
      put x date9.;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
28MAY2012
```

See Also

Functions:

- “HOLIDAY Function” on page 677
- “INTNX Function” on page 732
- “MONTH Function” on page 834
- “WEEKDAY Function” on page 1181

ORDINAL Function

Orders a list of values, and returns a value that is based on a position in the list.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

ORDINAL(*position*, *expression-1, expression-2* [, ...*expression-n*])

Arguments

position
specifies a whole number that is less than or equal to the number of elements in the list of arguments.

Requirement *position* must be a positive number.

Data type DOUBLE

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

The ORDINAL function sorts the list and returns the argument in the list that is specified by *position*. Missing values are sorted low and are placed before any numeric values.

Comparisons

The ORDINAL function counts both null, missing, non-null, and nonmissing values, whereas the SMALLEST function counts only non-null and nonmissing values.

Example

The following program illustrates the ORDINAL function:

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=ordinal(4,1,2,3,-4,5,6,7);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
3
```

PCTL Function

Returns the percentile that corresponds to the percentage.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

PCTL[*n*](*percentage*, *expression*[, ...*expression*])

Arguments

n

is a digit from 1 to 5 that specifies the definition of the percentile to be computed.

Default definition 5

Data type DOUBLE

See QNTLDEF= (alias PCTLDEF=) descriptions in the table "Methods for Computing Quantile Statistics" in the *Base SAS Procedures Guide*.

percentage

specifies the percentile to be computed.

Data type DOUBLE

Tip *percentage* is numeric where, $0 \leq percentage \leq 100$.

expression

specifies any valid expression that evaluates to a numeric value, whose value is computed in the percentile calculation.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The PCTL function returns the percentile of the non-null or nonmissing values corresponding to the percentage. If *percentage* is null or missing, less than zero, or greater than 100, the PCTL function generates an error message.

Example

The following program illustrates the PCTL function:

```
data test(overwrite=yes);
   dcl double lower_quartile percentile_def2 lower_tertile
percentile_def3;
   dcl double median upper_tertile upper_quartile;
   method run();
      lower_quartile=pctl(25, 2, 4, 1, 3);
      put lower_quartile=;
      percentile_def2=pctl2(25, 2, 4, 1, 3);
      put percentile_def2=;
      lower_tertile=pctl(100/3, 2, 4, 1, 3);
      put lower_tertile=;
      percentile_def3=pctl3(100/3, 2, 4, 1, 3);
      put percentile_def3=;
```

```
        median=pctl(50, 2, 4, 1, 3);
        put median=;
        upper_tertile=pctl(200/3, 2, 4, 1, 3);
        put upper_tertile=;
        upper_quartile=pctl(75, 2, 4, 1, 3);
        put upper_quartile=;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
lower_quartile=1.5
percentile_def2=1
lower_tertile=2
percentile_def3=2
median=2.5
upper_tertile=3
upper_quartile=3.5
```

PDF Function

Returns a value from a probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Data type: DOUBLE

Syntax

PDF(*'distribution'*, *quantile* [, *parameter-1, ..., parameter-k*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'

Distribution	Argument
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note Except for T, F, and NORMALMIX, you can identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of the random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional numeric constants, variables, or expressions that specify the values of *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

Data type DOUBLE

See Also

Functions:

- "CDF Function" on page 370
- "LOGCDF Function" on page 798
- "LOGPDF Function" on page 801
- "LOGSDF Function" on page 804
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

References

Shmueli, G., T. Minka,, S. Borle, and P. Boatwright. 2005. "A Useful Distribution for Fitting Discrete Data: Revival of the Conway-Maxwell-Poisson Distribution." *Applied Statistics* : 54:127–142.

PDF BERNOULLI Distribution Function

Returns a value from the Bernoulli probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('BERNOULLI', *x*, *p*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

Details

The PDF function for the Bernoulli distribution returns the probability density function with the probability of success equal to *p*. The PDF function is evaluated at the value *x*.

$$PDF('BERN', x, p) = \begin{cases} 0 & x < 0 \\ 1-p & x = 0 \\ 0 & 0 < x < 1 \\ p & x = 1 \\ 0 & x > 1 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following program illustrates the PDF Bernoulli distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('BERN',0,.25);
      put 'Bern dist:      ' y;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log:

```
Bern dist:      0.75
```

See Also

Functions:

- “CDF BERNOULLI Distribution Function” on page 372
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF BETA Distribution Function

Returns a value from the beta probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF (’BETA’, *x*, *a*, *b* [, *l*, *r*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range *a* > 0

Data type DOUBLE

b

is a numeric constant, variable, or expression that specifies a shape parameter.

Range *b* > 0

Data type DOUBLE

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range *r* > *l*

Data type DOUBLE

Details

The PDF function for the beta distribution returns the probability density function with the shape parameters *a* and *b*. The PDF function is evaluated at the value *x*.

$$PDF('BETA', x, a, b, l, r) = \begin{cases} 0 & x < l \\ \frac{1}{\beta(a,b)} \frac{(x-l)^{a-1}(r-x)^{b-1}}{(r-l)^{a+b-1}} & l \le x \le r \\ 0 & x > r \end{cases}$$

Note: The quantity $\frac{x-l}{r-l}$ is forced to be $\varepsilon \le \frac{x-l}{r-l} \le 1 - 2\varepsilon$.

Example

The following program illustrates the PDF Beta distribution function:

```
data _null_;
   dcl double y;
   method init();
```

```
        y=pdf('BETA', 0.2, 3, 4);
        put 'Beta dist:      ' y;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
Beta dist:       1.2288
```

See Also

Functions:

- “CDF BETA Distribution Function” on page 374
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF BINOMIAL Distribution Function

Returns a value from the binomial probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF (‘BINOMIAL’, *m*, *p*, *n*)

Arguments

m

is a random variable that counts the number of successes. This argument must be a whole number.

Range *m*=0, 1, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

n

is a parameter that counts the number of independent Bernoulli trials. This argument must be a whole number.

Range *n*=0, 1, ...

Data type DOUBLE

Details

The PDF function for the binomial distribution returns the probability density function with the parameters *p* and *n*. The PDF function is evaluated at the value *m*.

$$PDF('BINOM', m, p, n) = \begin{cases} 0 & m < 0 \\ \binom{n}{m} p^m (1-p)^{n-m} & 0 \le m \le n \\ 0 & m > n \end{cases}$$

Note: There are no *location* or *scale* parameters for the binomial distribution.

Example

The following program illustrates the PDF Binomial distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('BINOM',4,.5,10);
      put 'Binom dist:     ' y;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
Binom dist:     0.205078125
```

See Also

Functions:

- “CDF BINOMIAL Distribution Function” on page 376
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF CAUCHY Distribution Function

Returns a value from the Cauchy probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF (‘CAUCHY’, *x* [, *θ*, *λ*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ
is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default	1
Range	λ > 0
Data type	DOUBLE

Details

The PDF function for the Cauchy distribution returns the probability density function with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('CAUCHY', x, \theta, \lambda) = \frac{1}{\pi}\left(\frac{\lambda}{\lambda^2 + (x-\theta)^2}\right)$$

Example

The following program illustrates the PDF Cauchy distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('CAUCHY',2);
      put 'Cauchy dist:    ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Cauchy dist:    0.06366197723675
```

See Also

Functions:

- “CDF CAUCHY Distribution Function” on page 378
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF Chi-Square Distribution Function

Returns a value from the chi-square probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('CHISQUARE', *x*, *df* [, *nc*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df
is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range $df > 0$

Data type DOUBLE

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range $nc \geq 0$

Data type DOUBLE

Details

The PDF function for the chi-square distribution returns the probability density function of a chi-square distribution, with *df* degrees of freedom and the noncentrality parameter *nc*. The PDF function is evaluated at the value *x*. This function accepts noninteger degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central chi-square distribution.

$$PDF('CHISQ', x, \nu, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} p_c(x, \nu + 2j) & x \geq 0 \end{cases}$$

In this equation, $p_c(.,.)$ denotes the density from the central chi-square distribution:

$$p_c(x, a) = \frac{1}{2} p_g\left(\frac{x}{2}, \frac{a}{2}\right)$$

In this equation, $p_g(y,b)$ is the density from the gamma distribution:

$$p_g(y, b) = \frac{1}{\Gamma(b)} e^{-y} y^{b-1}$$

Example

The following program illustrates the PDF Chi-Square distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('CHISQ',11.264,11);
      put 'Chisq dist:     ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Chisq dist:     0.08168618682435
```

See Also

Functions:

- “CDF Chi-Square Distribution Function” on page 380
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF Conway-Maxwell-Poisson Distribution Function

Returns a value from the Conway-Maxwell-Poisson probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF('CONMAXPOI',*y*,*λ*,*ν*)

Arguments

y

is a numeric constant, variable, or expression that specifies a nonnegative, whole number representing a count.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a location parameter, similar to the Poisson mean parameter.

Data type DOUBLE

v

is a numeric constant, variable, or expression that specifies a dispersion parameter.

Data type DOUBLE

Details

The Conway-Maxwell-Poisson (CMP) distribution is a generalization of the Poisson distribution that enables you to model underdispersed and overdispersed data. The CMP distribution is defined according to this equation:

$$P\left(Y = y\,;\lambda,\nu\right) = \frac{1}{Z(\lambda,\nu)}\frac{\lambda^{y}}{\left(y!\right)^{\nu}} \qquad y = 0, 1, 2, \ldots$$

The normalization factor is expressed by this equation:

$$Z\left(\lambda,\nu\right)=\sum_{n=0}^{\infty}\frac{\lambda^{n}}{(n!)^{\nu}}$$

λ and v are nonnegative and not simultaneously zero.

The additional parameter, v, allows for flexibility in modeling the tail behavior of the distribution. If v=1, the ratio is equal to the rate of decay of the Poisson distribution. If v<1, the rate of decay decreases, which enables you to model processes that have longer tails than the Poisson distribution (overdispersed data). If v>1, the rate of decay increases in a nonlinear manner, thus shortening the tail of the distribution (underdispersed data).

There are several special cases of the Conway-Maxwell-Poisson distribution. If λ<1 and v→∞, the Conway-Maxwell-Poisson distribution results in the Bernoulli distribution. In this case, the data can take only the values 0 and 1, which represent an extreme underdispersion. If v=1, the Poisson distribution is recovered with its equidispersion property. When v=0 and λ<1, the normalization factor is convergent and forms this geometric series:

$$Z\left(\lambda,0\right)=\frac{1}{1-\lambda}$$

The probability density function is represented by this equation:

$$P\left(Y=y\,;\lambda,\nu=0\right)=\left(1-\lambda\right)\lambda^{y}$$

The geometric distribution represents a case of severe overdispersion.

Mean, Variance, and Dispersion for the Conway-Maxwell-Poisson Model

The mean and variance of the Conway-Maxwell-Poisson distribution are defined by these equations:

$$E\left[Y\right]=\frac{\partial\,1n\;Z}{\partial\,1n\,\lambda}$$

$$V\left[Y\right]=\frac{\partial^{2}\,1n\,Z}{\partial^{2}\,1n\,\lambda}$$

The Conway-Maxwell-Poisson distribution does not have closed-form expressions for its moments in terms of parameters λ and v. However, the moments can be approximated. (For more information about the Conway-Maxwell-Poisson distribution and discrete data, see the References section that is located at the end of this function.) Use asymptotic expressions for Z to derive E(Y) and V(Y) as these equations show:

$$E\left[Y\right]\approx\lambda^{1/\nu}+\frac{1}{2\nu}-\frac{1}{2}$$

$$V\left[Y\right]\approx\frac{1}{\nu}\lambda^{1/\nu}$$

In the Conway-Maxwell-Poisson model, the summation of infinite series is evaluated using a logarithmic expansion. (For more information about the Conway-Maxwell-Poisson distribution and discrete data, see the References section that is

located at the end of this function.) The mean and variance are calculated as follows for the Conway-Maxwell-Poisson model:

$$E\left(Y\right) = \frac{1}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{j^{\lambda^j}}{(j!)^{\nu}}$$

$$V\left(Y\right) = \frac{1}{Z(\lambda,\nu)} \sum_{j=0}^{\infty} \frac{j^2\lambda^j}{(j!)^{\nu}} - E(Y)^2$$

The dispersion is defined as follows:

$$D\left(Y\right) = \frac{V(Y)}{E(Y)}$$

Example

The following program illustrates the PDF Conway-Maxwell-Poisson distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('CONMAXPOI', .2, 2.3, .4);
      put 'ConMaxPoi dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
ConMaxPoi dist:      0.00977326351239
```

See Also

Functions:

- “CDF Conway-Maxwell-Poisson Distribution Function” on page 382
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF EXPONENTIAL Distribution Function

Returns a value from the exponential probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('EXPONENTIAL', *x* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the exponential distribution returns the probability density function of an exponential distribution, with the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('EXPO', x, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

Example

The following program illustrates the PDF Exponential distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('EXPO',1);
      put 'Expo dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Expo dist:      0.36787944117144
```

See Also

Functions:

- “CDF Exponential Distribution Function” on page 383
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF F Distribution Function

Returns a value from the F probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF (‘F’, *x*, *ndf*, *ddf* [, *nc*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

ndf

is a numeric constant, variable, or expression that specifies the numerator degrees of freedom.

Range *ndf*> 0

Data type DOUBLE

ddf

is a numeric constant, variable, or expression that specifies the denominator degrees of freedom.

Range *ddf* > 0

Data type DOUBLE

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The PDF function for the *F* distribution returns the probability density function of an *F* distribution, with *ndf* numerator degrees of freedom, *ddf* denominator degrees of freedom, and the noncentrality parameter *nc*. The PDF function is evaluated at the value *x*. This PDF function accepts noninteger degrees of freedom for *ndf* and *ddf*. If *nc* is omitted or equal to zero, the value returned is from a central *F* distribution. In the following equation, let $\nu_1 = ndf$, let $\nu_2 = ddf$, and let $\lambda = nc$. This equation describes the PDF function for the *F* distribution:

$$PDF('F', x, \nu_1, \nu_2, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} p_f(f, \nu_1 + 2j, \nu_2) & x \geq 0 \end{cases}$$

In the equation, $p_f(f,u_1,u_2)$ is the density from the central *F* distribution:

$$p_f(f, u_1, u_2) = p_B\left(\frac{u_1 f}{uf + u_2}, \frac{u_1}{2}, \frac{u_2}{2}\right) \frac{u_1 u_2}{(u_2 + u_1 f)^2}$$

In the equation, $p_B(x,a,b)$ is the density from the standard beta distribution.

Note: There are no *location* or *scale* parameters for the *F* distribution.

Example

The following program illustrates the PDF F distribution function:

```
data _null_;
   dcl double y;
   method init();
     y=pdf('F',3.32,2,3);
     put 'F dist:          ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
F dist:         0.05402696258579
```

See Also

Functions:

- “CDF F Distribution Function” on page 385
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF GAMMA Distribution Function

Returns a value from the gamma probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('GAMMA', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $\lambda > 0$

Data type DOUBLE

Details

The PDF function for the gamma distribution returns the probability density function of a gamma distribution, with the shape parameter *a* and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('GAMMA', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda^a \Gamma(a)} x^{a-1} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

Example

The following program illustrates the PDF Gamma distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('GAMMA',1,3);
      put 'Gamma dist:     ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Gamma dist:     0.18393972058572
```

See Also

Functions:

- “CDF GAMMA Distribution Function” on page 387
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “SDF Function” on page 1065

PDF Generalized Poisson Distribution Function

Returns a value from the generalized Poisson probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF (‘GENPOISSON’, *x*, *θ*, *η*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $<10^5$ and >0

Data type DOUBLE

η

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≥0 and <0.95

Data type DOUBLE

Tip When η =0, the distribution is the Poisson distribution with a mean and variance of θ. When η>0, the mean is $\theta \div (1-\eta)$ and the variance is $\theta \div (1-\eta)^3$.

Details

Here is the probability mass function for the generalized Poisson distribution:

$$f(x;\theta,\eta) = \theta(\theta+\eta x)^{x-1} e^{-\theta-\eta x}/x!, \quad x = 0,1,2,..., \qquad \theta > 0,\ 0 \le \eta < 1$$

Example

The following program illustrates the PDF Generalized Poisson distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('GENPOISSON', 9, 1, .7);
      put 'GENPOISSON dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
GENPOISSON dist:      0.01501309145504
```

See Also

Functions:

- “CDF Generalized Poisson Distribution Function” on page 389
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998

- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

PDF GEOMETRIC Distribution Function

Returns a value from the geometric probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('GEOMETRIC', *m*, *p*)

Arguments

m

is a numeric constant, variable, or expression that specifies the number of failures before the first success.

Range	$m \geq 0$
Data type	DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range	$0 \leq p \leq 1$
Data type	DOUBLE

Details

The PDF function for the geometric distribution returns the probability density function of a geometric distribution, with the parameter *p*. The PDF function is evaluated at the value *m*.

$$PDF('GEOM', m, p) = \begin{cases} 0 & m < 0 \\ p(1-p)^m & m \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Example

The following program illustrates the PDF Geometric distribution function:

```
data _null_;
   dcl double y;
   method init();
     y=pdf('GEOMETRIC', 5, .3);
     put 'geometric dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
geometric dist:      0.050421
```

See Also

Functions:

- “CDF GEOMETRIC Distribution Function” on page 391
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF Hypergeometric Distribution Function

Returns a value from a hypergeometric probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('HYPER', *x*, *N*, *R*, *n* [, *o*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Data type DOUBLE

N

is a numeric constant, variable, or expression that specifies a population size parameter. This argument must be a whole number.

Range *N*=1, 2, ...

Data type DOUBLE

R

is a numeric constant, variable, or expression that specifies the number of items in the category of interest. This argument must be a whole number.

Range *R*=0, 1, ..., *N*

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a sample size parameter. This argument must be a whole number.

Range *n*=1, 2, ..., *N*

Data type DOUBLE

o

is a numeric constant, variable, or expression that specifies an optional odds ratio parameter.

Range *o* > 0

Data type DOUBLE

Details

The PDF function for the hypergeometric distribution returns the probability density function of an extended hypergeometric distribution, with population size *N*, number of items *R*, sample size *n*, and odds ratio *o*. The PDF function is evaluated at the value *x*. If *o* is omitted or equal to 1, the value returned is from the usual hypergeometric distribution.

$$PDF\ ('HYPER', x, N, R, n, o) =$$
$$\begin{cases} 0 & x < max(0, R+n-N) \\ \dfrac{\binom{R}{x}\binom{N-R}{n-x}o^x}{\sum_{j=max(0,R+n-N)}^{min(R,n)}\binom{R}{j}\binom{N-R}{n-j}o^j} & max(0, R+n-N) \le x \le min(R, n) \\ 0 & x > min(R, n) \end{cases}$$

Example

The following program illustrates the PDF Hypergeometric distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('HYPER', 2, 200, 50, 10);
      put 'Hyper dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Hyper dist:      0.28685059643368
```

See Also

Functions:

- “CDF HYPERGEOMETRIC Distribution Function” on page 392
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF LAPLACE Distribution Function

Returns a value from the Laplace probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('LAPLACE', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the Laplace distribution returns the probability density function of the Laplace distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('LAPLACE', x, \theta, \lambda) = \frac{1}{2\lambda} exp\left(- \frac{|x - \theta|}{\lambda} \right)$$

Example

The following program illustrates the PDF Laplace distribution function:

```
data _null_;
   dcl double y;
   method init();
```

```
      y=pdf('LAPLACE', 1);
      put 'Laplace dist:   ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Laplace dist:   0.18393972058572
```

See Also

Functions:

- “CDF LAPLACE Distribution Function” on page 395
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF LOGISTIC Distribution Function

Returns a value from the logistic probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF (‘LOGISTIC’, *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ
is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the logistic distribution returns the probability density function of a logistic distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('LOGISTIC', x, \theta, \lambda) = \frac{exp\left(-\frac{x-\theta}{\lambda}\right)}{\lambda\left(1+exp\left(-\frac{x-\theta}{\lambda}\right)\right)^2}$$

Example

The following program illustrates the PDF Logistic distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('LOGISTIC', 1);
      put 'Logistic dist:   ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Logistic dist:   0.19661193324148
```

See Also

Functions:

- “CDF LOGISTIC Distribution Function” on page 396

- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF LOGNORMAL Distribution Function

Returns a value from the lognormal probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF (‘LOGNORMAL’, *x* [, *θ*, *λ*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ
is a numeric constant, variable, or expression that specifies a log scale parameter. exp(θ) is a scale parameter.

Default 0

Data type DOUBLE

λ
is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the lognormal distribution returns the probability density function of a lognormal distribution, with the log scale parameter θ and the shape parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('LOGN', x, \theta, \lambda) = \begin{cases} 0 & x \le 0 \\ \frac{1}{x\lambda\sqrt{2\,\pi}}\ exp\left(-\frac{(log(x)-\theta)^2}{2\lambda^2}\right) & x > 0 \end{cases}$$

Example

The following program illustrates the PDF Lognormal distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('LOGNORMAL', 1);
      put 'LogNormal dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
LogNormal dist:      0.39894228040143
```

See Also

Functions:

- “CDF LOGNORMAL Distribution Function” on page 398
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF NEGBINOMIAL Distribution Function

Returns the value from the negative binomial probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('NEGBINOMIAL', *m*, *p*, *n*)

Arguments

m

is a numeric constant, variable, or expression that specifies a random variable that counts the number of failures. This argument must be a positive, whole number.

Range *m*=0, 1, ...

Data type DOUBLE

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies a value that counts the number of successes.

Range *n*>0

Data type DOUBLE

Details

The PDF function for the negative binomial distribution returns the probability density function of a negative binomial distribution, with probability of success *p* and number of successes *n*. The PDF function is evaluated at the value *m*.

$$PDF('NEGB', m, p, n) = \begin{cases} 0 & m < 0 \\ \binom{n+m-1}{n-1} p^n (1-p)^m & m \ge 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the negative binomial distribution.

Example

The following program illustrates the PDF Negative Binomial distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('NEGB',1,.5,2);
      put 'NegB dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
NegB dist:      0.25
```

See Also

Functions:

- “CDF NEGBINOMIAL Distribution Function” on page 400
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “SDF Function” on page 1065

PDF NORMAL Distribution Function

Returns a value from the normal probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF ('NORMAL', *x* [, *θ*, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Data type DOUBLE

Details

The PDF function for the normal distribution returns the probability density function of a normal distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('NORMAL', x, \theta, \lambda) = \frac{1}{\lambda\sqrt{2\,\pi}}\ exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

Example

The following program illustrates the PDF Normal distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('NORMAL',1.96);
      put 'Normal dist:    ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Normal dist:    0.05844094433345
```

See Also

Functions:

- “CDF NORMAL Distribution Function” on page 402
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF NORMALMIX Distribution Function

Returns a value from the normal mixture probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF (‘NORMALMIX’, *x*, *n*, p_1, p_2, ... p_n, m_1, m_2, ... m_n, s_1, s_2, ... s_n)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

n

is a numeric constant, variable, or expression that specifies the number of mixtures. This argument must be a whole number.

Range *n*=1, 2, ...

Data type DOUBLE

***p*$_i$**

is a list of numeric constants, variables, or expressions that specifies the *n* proportions, $p_1, p_2, \ldots, p_n$, where $\sum_{i=1}^{i=n} p_i = 1$.

Range *p*=0, 1, ...

Data type DOUBLE

***m*$_i$**

is a list of numeric constants, variables, or expressions that specifies the *n* means $m_1, m_2, \ldots, m_n$.

***s*$_i$**

is a list of numeric constants, variables, or expressions that specifies the *n* standard deviations $s_1, s_2, \ldots, s_n$.

Range *s* > 0

Data type DOUBLE

Details

The PDF function for the Normal Mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to *x*.

$$PDF('NORMALMIX', x, n, p, m, s) = \sum_{i=1}^{i=n} p_i \, PDF('NORMAL', x, m_i, s_i)$$

Weights for the Normal Mixture distribution must be nonnegative. If the sum of the weights does not equal 1, then the weights are treated as relative weights and adjusted so that the sum equals 1.

Note: There are no *location* or *scale* parameters for the Normal Mixture distribution.

Example

The following program illustrates the PDF Normal Mixture distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('NORMALMIX',2.3, 3, .33, .33, .34,
```

```
          .5, 1.5, 2.5, .79, 1.6, 4.3);
        put 'Normal mix dist:' y;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
Normal mix dist: 0.11655396435559
```

See Also

Functions:

- “CDF NORMALMIX Distribution Function” on page 404
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF PARETO Distribution Function

Returns a value from the Pareto probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF (‘PARETO’, *x*, *a* [, *k*])

Arguments

x

is a numeric constant, variable, or expression that specifies a numeric random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

k

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $k > 0$

Data type DOUBLE

Details

The PDF function for the Pareto distribution returns the probability density function of a Pareto distribution, with the shape parameter *a* and the scale parameter *k*. The PDF function is evaluated at the value *x*.

$$PDF('PARETO', x, a, k) = \begin{cases} 0 & x < k \\ \frac{a}{k}\left(\frac{k}{x}\right)^{a+1} & x \geq k \end{cases}$$

Example

The following program illustrates the PDF Pareto distribution function:

```
data _null_;
   dcl double y;
   method init();
     y=pdf('PARETO', 1, 1);
     put 'Pareto dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Pareto dist:      1
```

See Also

Functions:

- "CDF PARETO Distribution Function" on page 406
- "LOGCDF Function" on page 798
- "LOGPDF Function" on page 801
- "LOGSDF Function" on page 804
- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

PDF POISSON Distribution Function

Returns a value from the Poisson probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('POISSON', *n*, *m*)

Arguments

n

is a numeric constant, variable, or expression that specifies a random variable. This argument must be a whole number.

Range *n*=0, 1, ...

Data type DOUBLE

m

is a numeric constant, variable, or expression that specifies a mean parameter.

Range $m > 0$

Data type DOUBLE

Details

The PDF function for the Poisson distribution returns the probability density function of a Poisson distribution, with mean *m*. The PDF function is evaluated at the value *n*.

$$PDF('POISSON', n, m) = \begin{cases} 0 & n < 0 \\ e^{-m}\frac{m^n}{n!} & n \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the Poisson distribution.

Example

The following program illustrates the PDF Poisson distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('POISSON', 2, 1);
      put 'Poisson dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Poisson dist:      0.18393972058572
```

See Also

Functions:

- “CDF POISSON Distribution Function” on page 407
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF T Distribution Function

Returns a value from the T probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('T', *t*, *df* [, *nc*])

Arguments

t
is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

df
is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

Data type DOUBLE

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Data type DOUBLE

Details

The PDF function for the *T* distribution returns the probability density function of a *T* distribution, with degrees of freedom *df* and the noncentrality parameter *nc*. The PDF function is evaluated at the value *x*. This PDF function accepts noninteger degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central *T* distribution. In this equation, let $\nu = df$ and let $\delta = nc$.

$$PDF('T',t,\nu,\delta) = \frac{1}{2^{\frac{\nu}{2}-1}\Gamma\left(\frac{\nu}{2}\right)}\int_0^{\infty} x^{\nu-1}e^{-\frac{1}{2}x^2}\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{tx}{\sqrt{\nu}}-\delta\right)^2}\frac{x}{\sqrt{\nu}}dx$$

Note: There are no *location* or *scale* parameters for the *T* distribution.

Example

The following program illustrates the PDF T distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('T',.9,5);
      put 'T dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
T dist:      0.24194434361358
```

See Also

Functions:

- “CDF T Distribution Function” on page 409
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF TWEEDIE Distribution Function

Returns a value from the Tweedie probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('TWEEDIE', *y*, *p* [, *μ*, *ϕ*])

Arguments

y

is a numeric constant, variable, or expression that specifies a random variable.

Range $y \geq 0$

Data type DOUBLE

Notes This argument is required.

When $y>1$, y is numeric. When $p=1$, y is a whole number.

p

is a numeric constant, variable, or expression that specifies the power parameter.

Range $p \geq 1$

Data type DOUBLE

Note This argument is required.

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu > 0$

Data type DOUBLE

ϕ

is a numeric constant, variable, or expression that specifies the dispersion parameter.

Default 1

Range $\phi > 0$

Data type DOUBLE

Details

The PDF function for the Tweedie distribution returns an exponential dispersion model with variance and mean related by the equation *variance*= $\phi * \mu^p$.

$$\frac{1}{y}\sum_{j=1}^{\infty}\left(\frac{y^{-j\alpha}(p-1)^{j\alpha}}{\phi^{j(1-\alpha)}(2-p)^{j}j!\Gamma(-j\alpha)}\right)\exp\left(\frac{1}{\phi}\left(y\frac{\mu^{1-p}-1}{1-p}-\frac{\mu^{2-p}-1}{2-p}\right)\right)$$

The following relationship applies to the preceding equation:

$$\alpha=\frac{2-p}{1-p}$$

Note: The accuracy of computed Tweedie probabilities is highly dependent on the location in parameter space. Ten digits of accuracy are usually available except when p is near 2 or ϕ is near 0. In either of these cases, the accuracy might be as low as six digits.

Note: To avoid issues with numerical data, μ and Φ cannot be less than the constant SQRTMACEPS.

Example

The following program illustrates the PDF Tweedie distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('TWEEDIE', .8, 5);
      put 'Tweedie dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Tweedie dist:      0.74229082358846
```

See Also

Functions:

- “CDF TWEEDIE Distribution Function” on page 411
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804

- "QUANTILE Function" on page 998
- "SDF Function" on page 1065
- "SQUANTILE Function" on page 1090

PDF UNIFORM Distribution Function

Returns a value from the uniform probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: DOUBLE

Syntax

PDF ('UNIFORM', *x* [, *l*, *r*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

l
is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

Data type DOUBLE

r
is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range $r > l$

Data type DOUBLE

Details

The PDF function for the uniform distribution returns the probability density function of a uniform distribution, with the left location parameter *l* and the right location parameter *r*. The PDF function is evaluated at the value *x*.

$$PDF('UNIFORM', x, l, r) = \begin{cases} 0 & x < l \\ \frac{1}{r-l} & l \le x \le r \\ 0 & x > r \end{cases}$$

Example

The following program illustrates the PDF Uniform distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('UNIFORM', 0.25);
      put 'Uniform dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Uniform dist:      1
```

See Also

Functions:

- “CDF UNIFORM Distribution Function” on page 413
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF Wald (Inverse Gaussian) Distribution Function

Returns a value from the Wald (also known as the inverse Gaussian) probability density (mass) distribution.

Categories: CAS
Probability

Alias: PMF

Returned data type: Double

Syntax

PDF ('WALD', *x*, *λ* [, *μ*])
PDF ('IGAUSS', *x*, *λ* [, *μ*])

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

λ
is a numeric constant, variable, or expression that specifies a shape parameter.

Range $\lambda > 0$

μ
is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu > 0$

Details

The PDF function for the Wald distribution returns the probability density function of a Wald distribution, with the shape parameter *λ*, which is evaluated at the value *x*.

$$fx(x) = \left[\frac{\lambda}{2\pi x^3}\right]^{1/2} \exp\left\{-\frac{\lambda}{2\mu^2 x}(x-\mu)^2\right\}, \qquad x > 0$$

Example

The following program illustrates the PDF Wald distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('WALD', 1, 2);
      put 'Wald dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Wald dist:      0.56418958354775
```

See Also

Functions:

- “CDF WALD (Inverse Gaussian) Distribution Function” on page 415
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PDF WEIBULL Distribution Function

Returns a value from the Weibull probability density (mass) distribution.

Categories:	CAS Probability
Alias:	PMF
Returned data type:	DOUBLE

Syntax

PDF('WEIBULL', *x*, *a* [, *λ*])

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

Data type DOUBLE

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

Data type DOUBLE

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $\lambda > 0$

Data type DOUBLE

Details

The PDF function for the Weibull distribution returns the probability density function of a Weibull distribution, with the shape parameter *a* and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('WEIBULL', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ exp\left(-\left(\frac{x}{\lambda}\right)^a\right)\frac{a}{\lambda}\left(\frac{x}{\lambda}\right)^{a-1} & x \geq 0 \end{cases}$$

Example

The following program illustrates the PDF Weibull distribution function:

```
data _null_;
   dcl double y;
   method init();
      y=pdf('WEIBULL', 1, 2);
      put 'WEIBULL dist:      ' y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
WEIBULL dist:      0.73575888234288
```

See Also

Functions:

- “CDF WEIBULL Distribution Function” on page 417
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “QUANTILE Function” on page 998
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

PERM Function

Computes the number of permutations of *n* items that are taken *r* at a time.

Categories: CAS
Combinatorial

Returned data type: DOUBLE

Syntax

PERM(*n*[, *r*])

Arguments

n

specifies any valid expression that represents the total number of elements from which the sample is chosen.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

r

Specifies any valid expression that represents the number of chosen elements.

Restriction $r \leq n$

Data type DOUBLE

Note If *r* is omitted, the function returns the factorial of *n*.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The mathematical representation of the PERM function is given by the following equation:

$$PERM(n, r) = \frac{n!}{(n-r)!}$$

with $n \geq 0$, $r \geq 0$, and $n \geq r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the PERM function.

Example

The following program illustrates the PERM function:

```
data _null_;
  dcl double x y z;
  method run();
     x=perm(5,1);
     y=perm(5);
     z=perm(5, 2);
     put x y z;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
5 120 20
```

See Also

Functions:

- "COMB Function" on page 435
- "FACT Function" on page 535

PMT Function

Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

PMT(*rate*, *number-of-periods*, *principal-amount*[, *future-amount*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a null or missing value is specified.

Data type DOUBLE

future-amount
specifies the future amount. *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a missing value is specified.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a null or missing value is specified.

Data type DOUBLE

Example

- The monthly payment for a $10,000 loan with a nominal annual interest rate of 8% and 10 end-of-month payments can be computed in the following ways:

```
data _null_;
   dcl double Payment1;
   method init();
      Payment1 = PMT(0.08/12., 10, 10000);
      /* same results with PMT(0.08/12., 10, 10000, 0, 0) ***/
      put Payment1=;
   end;
enddata;
run;
```

  SAS writes the following output to the log:

```
1037.03208935915
```

- If the same loan has beginning-of-period payments, then payment can be computed as follows:

```
data _null_;
   dcl double Payment2;
   method init();
      Payment2 = PMT(0.08/12., 10, 10000, 0, 1);
      put Payment2= ;
   end;
enddata;
run;
```

  SAS writes the following output to the log:

```
1030.16432717796
```

- The payment for a $5,000 loan earning a 12% nominal annual interest rate that is to be paid back in five monthly payments is computed as follows:

```
data _null_;
   dcl double Payment3;
   method init();
      Payment3 = PMT(.01/12., 5, 5000);
      put Payment3= ;
   end;
enddata;
run;
```

  SAS writes the following output to the log:

```
1002.50138831008
```

- The payment for monthly periodic savings that accrue more than 18 years at a 6% nominal annual interest rate and that accumulate $50,000 at the end of the 18 years is computed as follows:

```
data _null_;
```

```
   dcl double Payment4;
   method init();
      /* this generates a negative number */
      Payment4 = PMT(.06/12., 216, 0, 50000, 0);
      put Payment4= ;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
-129.081160867993
```

POISSON Function

Returns the probability from a Poisson distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

POISSON(*m*, *n*)

Arguments

m

specifies any valid expression that evaluates to a numeric mean parameter.

Range $m \geq 0$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

n

specifies any valid expression that evaluates to a random variable.

Range $n \geq 0$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The POISSON function returns the probability that an observation from a Poisson distribution, with mean *m*, is less than or equal to *n*. To compute the probability that an observation is equal to a given value, *n*, compute the difference of two probabilities from the Poisson distribution for *n* and *n*−1.

Example

The following program illustrates the POISSON function:

```
data _null_;
   dcl double x;
   method run();
      x=poisson(1,2);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.9196986029286
```

POWER Function

Returns the value of a numeric value expression raised to a specified power.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

POWER(*numeric-expression*, *whole-number-expression*)

Arguments

numeric-expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

whole-number-expression
specifies any valid expression that evaluates to a numeric value. This argument must be a whole number.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

If *numeric-expression* is null, then the POWER function returns null. If the result is a number that does not fit into the range of the argument’s data type, the POWER function fails.

Example

The following program illustrates the POWER function:

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=power(5*3, 2);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
225
```

PPMT Function

Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

PPMT(*rate*, *period*, *number-of-periods*, *principal-amount*[, *future-amount*][, *type*])

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

period
specifies the payment period for which the principal payment is computed.

Requirement *Period* must be a whole number that is less than or equal to the value of *number-of-periods*

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a null or missing value is specified.

Data type DOUBLE

future-amount
specifies the future amount. *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a null or missing value is specified.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a null or missing value is specified.

Data type DOUBLE

Example

- The principal payment amount of the first monthly periodic payment for a 2-year, $2,000 loan with a nominal annual interest rate of 10% is computed as follows:

```
data test(overwrite=yes);
   dcl double PrincipalPayment;
   method run();
      PrincipalPayment = PPMT(0.1/12., 1, 24, 2000);
      put PrincipalPayment;
   end;
enddata;
run;
```

  SAS writes the following output to the log:

```
75.6231860083663
```

- The principal payment for a 3-year, $20,000 loan with beginning-of-month payments is computed as follows:

```
data test(overwrite=yes);
   dcl double PrincipalPayment2;
   method run();
      PrincipalPayment2 = PPMT(0.1/12, 1, 36, 20000, 0, 1);
      put PrincipalPayment2;
   end;
enddata;
run;
```

  SAS writes the following output to the log as the principal that was paid with the first payment:

```
640.010324505867
```

- The principal payment of an end-of-month payment loan with an outstanding balance of $5,000 at the end of three years is computed as follows:

```
data test(overwrite=yes);
   dcl double PrincipalPayment3;
   method run();
      PrincipalPayment3 = PPMT(0.1/12, 1, 36, 20000, 5000, 0);
      put PrincipalPayment3;
   end;
enddata;
run;
```

  SAS writes the following output to the log as the principal that was paid with the first payment:

```
359.007807907562
```

PROBBETA Function

Returns the probability from a beta distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBBETA(*x*, *a*, *b*)

Arguments

x

is a numeric random variable.

Range $0 \leq x \leq 1$

Data type DOUBLE

a

is a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

b

is a numeric shape parameter.

Range $b > 0$

Data type DOUBLE

Details

The PROBBETA function returns the probability that an observation from a beta distribution, with shape parameters *a* and *b*, is less than or equal to *x*.

Example

The following program illustrates the PROBBETA function:

```
data test (overwrite=yes);
   dcl double x;
   method run();
      x=probbeta(.2,3,4);
      put x= ;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log:

```
x=0.09888
```

PROBBNML Function

Returns the probability from a binomial distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBBNML(*p*, *n*, *m*)

Arguments

p
is a numeric probability of success parameter.

Range $0 \le p \le 1$

Data type DOUBLE

n
is the number of independent Bernoulli trials parameter. This argument must be a whole number.

Range $n > 0$

Data type DOUBLE

m
is the number of successes random variable. This argument must be a whole number.

Range $0 \le m \le n$

Data type DOUBLE

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success *p*, number of trials *n*, and number of successes *m*, is less than or equal to *m*. To compute the probability that an observation is equal to a given value *m*, compute the difference of two probabilities from the binomial distribution for *m* and *m*−1 successes.

Example

The following program illustrates the PROBBNML function:

```
data _null_;
   method run();
      x=probbnml(0.5,10,4);
      put x= ;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=0.376953125
```

PROBBNRM Function

Returns a probability from a bivariate normal distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBBNRM(*x*, *y*, *r*)

Arguments

x
specifies a numeric constant, variable, or expression.

Data type DOUBLE

y

specifies a numeric constant, variable, or expression.

Data type DOUBLE

r

is a numeric correlation coefficient.

Range $-1 \le r \le 1$

Data type DOUBLE

Details

The PROBBNRM function returns the probability that an observation (X, Y) from a standardized bivariate normal distribution with mean 0, variance 1, and a correlation coefficient *r*, is less than or equal to (*x*, *y*). That is, it returns the probability that X≤*x* and Y≤*y*. The following equation describes the PROBBNRM function, where *u* and *v* represent the random variables *x* and *y*, respectively:

$$\text{PROBBNRM}(x, y, r) = \frac{1}{2\pi\sqrt{1-r^2}} \int_{-\infty}^{x} \int_{-\infty}^{y} \exp\left[-\frac{u^2 - 2ruv + v^2}{2(1-r^2)} \right] dv\, du$$

Example

The following program illustrates the PROBBNRM function:

```
data _null_;
   dcl double p;
   method run();
      p=probbnrm(.4, -.3, .2);
      put p= ;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.27831833451901
```

PROBCHI Function

Returns the probability from a chi-square distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBCHI(*x*, *df*[, *nc*])

Arguments

x

is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

df

is a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

nc

is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Data type DOUBLE

Details

The PROBCHI function returns the probability that an observation from a chi-square distribution, with degrees of freedom *df* and noncentrality parameter *nc*, is less than or equal to *x*. This function accepts a noninteger degrees of freedom parameter *df*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central chi-square distribution.

Example

The following program illustrates the PROBCHI function:

```
data _null_;
   dcl double x;
   method run();
      x=probchi(11.264,11);
      put x= ;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log:

```
0.5785813293173
```

PROBDF Function

Calculates significance probabilities for Dickey-Fuller tests for unit roots in time series.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBDF(*x*, *n*[, *d*[, *type*]])

Arguments

x
is the test statistic.

Data type DOUBLE

n
is the sample size. The minimum value of *n* that is allowed depends on the value that is specified for the third argument, *d*. For *d* in the set (1,2,4,6,12), *n* must be a whole number greater than or equal to `max(2d,5)`. For other values of *d* the minimum value of *n* is 24.

Data type DOUBLE

d
is a whole number that gives the degree of the unit root that is tested for. For tests of a simple unit root, `(1-B)`, specify *d*=1. For tests for a seasonal unit root, specify that *d* is equal to the seasonal cycle length for tests. The default value of *d* is 1. That is, a test for a simple unit root is assumed if *d* is not specified. The maximum value of *d* is 12.

Data type DOUBLE

type
is a character argument that specifies the type of test statistic that is used. The values of *type* are the following:

RSM
specifies the regression test statistic for the single mean (intercept) case.

RTR
specifies the regression test statistic for the deterministic time trend case.

RZM
specifies the regression test statistic for the zero mean (no intercept) case.

SSM
specifies the studentized test statistic for the single mean (intercept) case.

STR
specifies the studentized test statistic for the deterministic time trend case.

SZM
specifies the studentized test statistic for the zero mean (no intercept) case.

Default SZM

Restriction The values STR and RTR are allowed only when *d*=1.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Theoretical Background

When a time series has a unit root, the series is nonstationary and the ordinary least squares (OLS) estimator is not normally distributed. Dickey (1976) and Dickey and Fuller (1979) studied the limiting distribution of the OLS estimator of autoregressive models for time series with a simple unit root. Dickey, Hasza, and Fuller (1984) obtained the limiting distribution for time series with seasonal unit roots. This section introduces the nonseasonal tests, and lists references for the nonseasonal tests.

In the Dicky-Fuller regression, the null hypothesis states that there is an autoregressive unit root $H_0 : \alpha = 1$, and an alternative, $H_\alpha : |\alpha| < 1$, where α is the autoregressive coefficient of the following time series:

$$y_t = \alpha y_{t-1} + \varepsilon_t$$

This model is referred to as the zero mean (ZM) model. The standard Dickey-Fuller (DF) test assumes that errors are white noise. There are two other types of regression models that include a constant or a time trend:

$$y_t = \mu + \alpha y_{t-1} + \varepsilon_t$$

$$y_t = \mu + \beta t + \alpha y_{t-1} + \varepsilon_t$$

These two models are referred to as the constant mean model (SM) and the trend model (TR), respectively. The constant mean model includes a constant mean μ of the time series. However, the interpretation of μ depends on the stationarity in the following sense: the mean in the stationary case when α < 1 is the trend in the

integrated case when α = 1. Therefore, the null hypothesis should be the joint hypothesis that α = 1 and μ = 0. However, for the unit root tests, the test statistics are concerned with the null hypothesis of α = 1. The joint null hypothesis is not commonly used. This issue is address in Bhargava, A. (1986) with a different nesting model.

Under the null of I(1) of the Dickey-Fuller test, the differenced process is not serially correlated. There is a great need for the generalization of this specification. The augmented Dickey-Fuller (ADF) test, originally proposed in Dickey and Fuller (1979), adjusts for the serial correlation in the time series by adding lagged first differences to the autoregressive model as follows. Consider the (p +1)th order autoregressive time series:

$$y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \ldots + \alpha_{p+1} y_{t-p-1} + e_t$$

The characteristic equation follows:

$$m^{p+1} - \alpha_1 m^p - \alpha_2 m^{p-1} - \ldots - \alpha_{p+1} = 0$$

If all the characteristic roots are less than 1 in absolute value, y_t, is stationary. y_t is nonstationary if there is a unit root. If there is a unit root, the sum of the autoregressive parameters is 1, and therefore you can test for a unit root by testing whether the sum of the autoregressive parameters is 1. The no-intercept model is parameterized as follows.

$$\nabla y_t = \delta y_{t-1} + \theta_1 \nabla y_{t-1} + \ldots + \theta_p \nabla y_{t-p} + e_t$$

In the equation above, the following relationships apply:

$$\nabla y_t = y_t - y_{t-1}$$

and

$$\delta = a_1 + \ldots + \alpha_{p+1} - 1$$

$$\theta_k = -\alpha_{k+1} - \ldots - \alpha_{p+1}$$

The estimators are obtained by regressing ∇y_t on y_{t-1}, $\nabla y_{t-1}, \ldots, \nabla y_{t-p}$. The t statistic of the ordinary least squares estimator of δ is the test statistic for the unit root test.

If the type argument value specifies a test for a nonzero mean (intercept case), the autoregressive model includes a mean term α_0. If the *type* argument value specifies a test for a time trend, the model also includes a time trend term and the model is as follows:

$$\nabla y_t = \alpha_0 + \gamma t + \delta y_{t-1} + \theta_1 \nabla y_{t-1} + \ldots + \theta_p \nabla y_{t-p} + e_1$$

For testing for a seasonal unit root, consider the multiplicative model.

$$(1 - \alpha_d \mathrm{B}^d)(1 - \theta_1 \mathrm{B} - \ldots - \theta_p \mathrm{B}^p) y_t = e_t$$

Let $\nabla^d y_t \equiv y_t - y_{t-d}$. The test statistic is calculated in the following steps:

1 Regress $\nabla^d y_t$ on $\nabla^d y_{t-1} \ \ldots \ \nabla^d y_{t-p}$ to obtain the initial estimators $\hat{\theta}_i$ and compute residuals $\hat{e}_t$. Under the null hypothesis that $\alpha_d = 1$, $\hat{\theta}_i$ are consistent estimators of θ_i.

2 Regress $\hat{e}_t$ on $(1 - \hat{\theta}_1 \mathrm{B} - \ldots - \hat{\theta}_p \mathrm{B}^p) y_{t-d}, \ \nabla^d y_{t-1}, \ldots, \nabla^d y_{t-p}$ to obtain estimates of $\delta = \alpha_d - 1$ and $\theta_i - \hat{\theta}_i$.

The *t* ratio for the estimates of δ that are produced by the second step is used as a test statistic for testing for a seasonal unit root. The estimates of θ_i are obtained by adding the estimates of $\theta_i - \hat{\theta}_i$ from the second step to $\hat{\theta}_i$ from the first step.

The series $(1 - \mathrm{B}^d) y_t$ is assumed to be stationary, where *d* is the value of the third argument to the PROBDF function.

If the series is an ARMA process, then a large value of *p* might be desirable in order to obtain a reliable test statistic. To determine an appropriate value for *p*, see Said and Dickey (1984).

Test Statistics

The Dickey-Fuller test is used to test the null hypothesis that the time series exhibits a lag *d* unit root against the alternative of stationarity. The PROBDF function computes the probability of observing a test statistic more extreme than *x* under the assumption that the null hypothesis is true. You should reject the unit root hypothesis when PROBDF returns a small (significant) probability value.

Consider the Dickey-Fuller regression first. There are several versions of the Dickey-Fuller test. The PROBDF function supports six versions, as selected by the *type* argument. Specify the *type* value that corresponds to how you calculated the test statistic *x*.

The last two characters of the *type* value specify the type of regression model that is used to compute the Dickey-Fuller test statistic. The meaning of the last two characters of the *type* value are as follows:

SM
specifies a single mean or intercept case. The test statistic *x* is assumed to be computed from the following regression model:

$$y_t = \mu + \alpha y_{t-1} + e_t$$

TR
specifies the intercept and deterministic time trend case. The test statistic *x* is assumed to be computed from the following regression model:

$$y_t = \mu + \gamma t + \alpha y_{t-1} + e_t$$

ZM
specifies the zero mean or no-intercept case. The test statistic *x* is assumed to be computed from the following regression model:

$$y_t = \alpha y_{t-1} + e_t$$

The first character of the *type* value specifies whether the regression test statistic or the studentized test statistic is used. Let $\widehat{\alpha}$ be the estimated regression coefficient for the lag of the series, and let se_{α} be the standard error of $\widehat{\alpha}$. The meaning of the first character of the *type* value is as follows:

R
: specifies the regression-coefficient-based test statistic. The test statistic follows:

$$\rho = n(\widehat{\alpha} - 1)$$

S
: specifies the studentized test statistic. The test statistic follows:

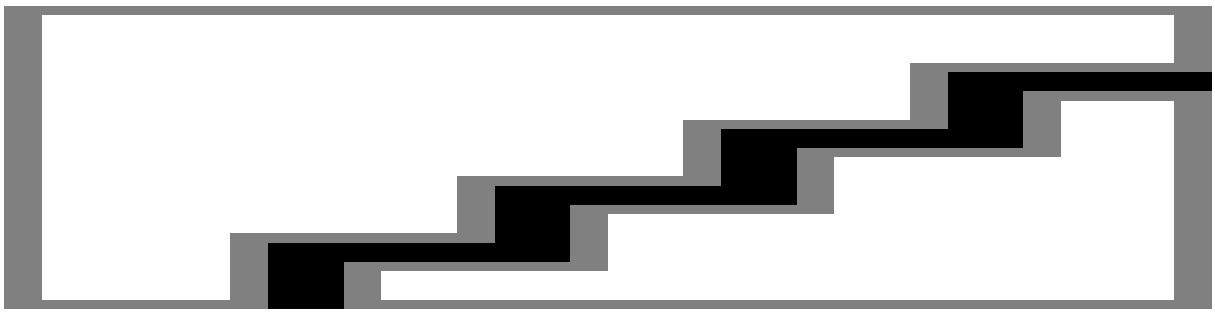
$$DF_t = \frac{(\widehat{\alpha} - 1)}{se_{\alpha}}$$

The equation for the *type* value of R is also called ρ-test. The equation for the *type* value of S is also called τ-test. For the zero mean model, the asymptotic distributions of the Dickey-Fuller test statistics follow:

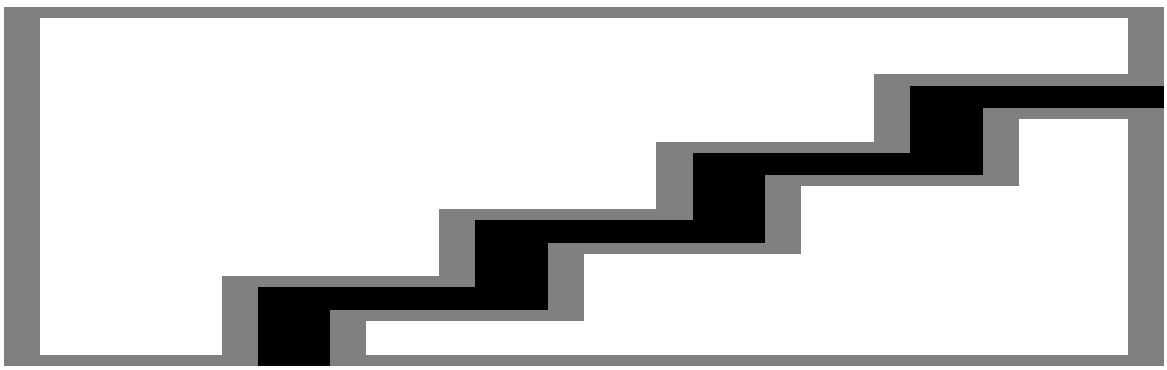
$$n(\widehat{\alpha} - 1) \Rightarrow \left(\int_0^1 W(r)\,dW(r)\right) \left(\int_0^1 W(r)^2 dr\right)^{-1}$$

$$DF_{\tau} \Rightarrow \left(\int_0^1 W(r)\,dW(r)\right) \left(\int_0^1 W(r)^2\,dr\right)^{-\frac{1}{2}}$$

For the constant mean model, the asymptotic distributions follow:



For the trend model, the asymptotic distributions follow:



The following equation applies to the equations that are shown above:

$$D = \int_0^1 W(r)^2\,dr - 12\left(\int_0^1 rW(r)dr\right)^2 + 12 \int_0^1 W(r)\,dr \int_0^1 rW(r)\,dr - 4\left(\int_0^1 W(r)dr\right)^2$$

For more information about the Dickey-Fuller test null distribution, see Dickey and Fuller (1979), Dickey, Hasza, and Fuller (1984), and Hamilton (1994). The preceding formulas are for the basic Dickey-Fuller test. The PROBDF function can also be used for the augmented Dickey-Fuller test, in which the error term is modeled as an autoregressive process. However, the test statistic is computed somewhat differently for the augmented Dickey-Fuller test. For the nonseasonal augmented Dickey-Fuller test, the test statistics can have one of the two forms similar to Dickey-Fuller test. One of the forms is the OLS *t* value, $\frac{\hat{\alpha}-1}{sd(\alpha)}$, and the other form is $\frac{n(\hat{\alpha}-1)}{1-\hat{\alpha}_1-\ldots-\hat{\alpha}_p}$

Example

In the following program, the table Test contains 104 observations of the time series variable *Y*. The program tests the null hypothesis that there exists a lag 4 seasonal unit root in the Y series. The following program illustrates how to perform the single-mean Dickey-Fuller regression coefficient test using PROC REG and the PROBDF function.

```
data test1;
   set test;
   y4 = lag4(y);
run;

proc reg data=test1 outest=alpha;
   model y = y4 / noprint;
run;

proc ds2;
data _null_;
   dcl double x p;
   method run();
      set alpha;
      x = 100 * ( y4 - 1 );
      p = probdf( x, 100, 4, 'RSM' );
      put p= pvalue5.3;
   end;
enddata;
run;
quit;
```

To perform the augmented Dickey-Fuller test, regress the differences of the series on lagged differences and on the lagged value of the series, and compute the test statistic from the regression coefficient for the lagged series. The following program illustrates how to perform the single-mean augmented Dickey-Fuller studentized test for a simple unit root using PROC REG and the PROBDF function:

```
data test1;
   set test;
   yl  = lag(y);
   yd  = dif(y);
   yd1 = lag1(yd); yd2 = lag2(yd);
   yd3 = lag3(yd); yd4 = lag4(yd);
```

```
run;

proc reg data=test1 outest=alpha covout;
   model yd = yl yd1-yd4 / noprint;
run;

proc ds2;
data _null_;
   dcl double a x p;
   method run();
      set alpha;
      retain a;
      if _type_ = 'PARMS' then
         a = yl;
      if _type_ = 'COV' & _NAME_ = 'Y1' then do;
         x = a / sqrt(yl);
         p = probdf( x, 99, 1, "SSM" );
         put p= ;
   end;
enddata;
run;
quit;
```

The %DFTEST macro provides an easier way to perform the Dickey-Fuller tests. The following statements perform the same tests as the preceding example:

```
%dftest(test, y, ar=4);
%put p=&dftest;
```

Note: When DS2 runs outside of SAS, such as in the SAS Federation Server and in grid computing environments, the SAS macro facility is not available and DS2 programs with macros fail to compile.

PROBF Function

Returns the probability from an *F* distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBF(*x*, *ndf*, *ddf*[, *nc*])

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

ndf
is a numeric numerator degrees of freedom parameter.

Range $ndf > 0$

Data type DOUBLE

ddf
is a numeric denominator degrees of freedom parameter.

Range $ddf > 0$

Data type DOUBLE

nc
is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Data type DOUBLE

Details

The PROBF function returns the probability that an observation from an *F* distribution, with numerator degrees of freedom *ndf*, denominator degrees of freedom *ddf*, and noncentrality parameter *nc*, is less than or equal to *x*. The PROBF function accepts noninteger degrees of freedom parameters *ndf* and *ddf*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central *F* distribution.

The significance level for an *F* test statistic is given by the following equation.

```
p=1-probf(x,ndf,ddf);
```

Example

The following program illustrates the PROBF function:

```
data test (overwrite=yes);
   dcl double x;
   method run();
      x=probf(3.32,2,3);
      put x= ;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.82639336022431
```

PROBGAM Function

Returns the probability from a gamma distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBGAM(*x*, *a*)

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

a
is a numeric shape parameter.

Data type DOUBLE

Details

The PROBGAM function returns the probability that an observation from a gamma distribution, with shape parameter *a*, is less than or equal to *x*.

Example

The following program illustrates the PROBGAM function:

```
data test (overwrite=yes);
   method run();
      x=probgam(1,3);
      put x= ;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.08030139707139
```

PROBHYPR Function

Returns the probability from a hypergeometric distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBHYPR(*N*, *K*, *n*, *x*[, *r*])

Arguments

N
is a population size parameter. This argument must be a whole number.

Range $N \geq 1$

Data type DOUBLE

K
is the number of items in the category of interest parameter. This argument must be a whole number.

Range $0 \leq K \leq N$

Data type DOUBLE

n
is the sample size parameter. This argument must be a whole number.

Range $0 \le n \le N$

Data type DOUBLE

x
is the random variable. This argument must be a whole number.

Range $max(0, K + n-N) \le x \le min(K,n)$

r
is a numeric odds ratio parameter.

Range $r \ge 0$

Data type DOUBLE

Details

The PROBHYPR function returns the probability that an observation from an extended hypergeometric distribution, with population size *N*, number of items *K*, sample size *n*, and odds ratio *r*, is less than or equal to *x*. If the optional parameter *r* is not specified or is set to 1, the value returned is from the usual hypergeometric distribution.

Example

The following program illustrates the PROBHYPR function:

```
data _null_;
   method run();
      x=probhypr(200,50,10,2);
      put x= ;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.52367340812173
```

PROBIT Function

Returns a quantile from the standard normal distribution.

Categories: CAS
Quantile

Returned data type: DOUBLE

Syntax

PROBIT(*p*)

Arguments

p
is a numeric probability.

Range $0 < p < 1$

Data type DOUBLE

Details

The PROBIT function returns the pth quantile from the standard normal distribution. The probability that an observation from the standard normal distribution is less than or equal to the returned quantile is *p*.

CAUTION
The result could be truncated to lie between -8.222 and 7.941.

Note: PROBIT is the inverse of the PROBNORM function.

Example

The following program illustrates the PROBIT function:

```
data _null_;
   dcl double x y;
   method run();
      x=probit(.025);
      put x= ;
      y=probit(1.e-7);
      put y= ;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=-1.95996398454005
y=-5.19933758219281
```

PROBMC Function

Returns a probability or a quantile from various distributions for multiple comparisons of means.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBMC(*distribution*, *q*, *prob*, *df*, *nparms*[, *parameters*])

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. The following distributions are valid:

Distribution	Argument
Analysis of Means	ANOM
One-sided Dunnett	DUNNETT1
Two-sided Dunnett	DUNNETT2
Maximum Modulus	MAXMOD
Partitioned Range	PARTRANGE
Studentized Range	RANGE
Williams	WILLIAMS

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

q
is the quantile from the distribution.

Restriction Either *q* or *prob* can be specified, but not both.

Data type DOUBLE

prob
is the left probability from the distribution.

Restriction Either *prob* or *q* can be specified, but not both.

Data type DOUBLE

df
is the degrees of freedom.

Note: A missing value is interpreted as an infinite value.

nparms
is the number of treatments.

Data type DOUBLE

Note For DUNNETT1 and DUNNETT2, the control group is not counted.

parameters
is a set of *nparms* parameters that must be specified to handle the case of unequal sample sizes. The meaning of *parameters* depends on the value of *distribution*. If *parameters* is not specified, equal sample sizes are assumed, which is usually the case for a null hypothesis.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Overview

The PROBMC function returns the probability or the quantile from various distributions with finite and infinite degrees of freedom for the variance estimate.

The *prob* argument is the probability that the random variable is less than *q*. Therefore, *p*-values can be computed as 1– *prob*. For example, to compute the critical value for a 5% significance level, set *prob*= 0.95. The precision of the computed probability is $O(10^{-8})$ (absolute error); the precision of computed quantile is $O(10^{-5})$.

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes.

Note: Williams' test is computed only for equal sample sizes.

Formulas and Parameters

The equations listed here define expressions that are used in equations that relate the probability, *prob*, and the quantile, *q*, for different distributions and different situations within each distribution. For these equations, let *v* be the degrees of freedom, *df*.

$$d\mu_\nu(x) = \frac{\nu^{\frac{\nu}{2}}}{\Gamma\left(\frac{\nu}{2}\right)2^{\frac{\nu}{2}-1}} x^{\nu-1}\varepsilon^{-\frac{\nu x^2}{2}}dx$$

$$\phi(x) = \frac{1}{\sqrt{2\pi}}\varepsilon^{-\frac{x^2}{2}}$$

$$\Phi(x) = \int_{-\infty}^{x} \phi(u)du$$

Computing the Analysis of Means

Analysis of Means (ANOM) applies to data that is organized as *k* (Gaussian) samples, the ith sample being of size n_i. Let $I = \sqrt{-1}$. The distribution function [1, 2, 3, 4, 5] is the CDF for the maximum absolute of a *k*-dimensional multivariate T vector, with ν degrees of freedom, and an associated correlation matrix $\rho_{ij} = -\alpha_i\alpha_j$. This equation can be written as follows.

$$\begin{aligned} prob &= r(|t_1| < h, |t_2| < h, ..., |t_k| < h) \\ &= \int_0^\infty \left\{ \int_0^\infty \prod_{j=0}^{j=k} g(sh, y, \alpha_j)\phi(y)dy \right\} d\mu_\nu(s) \end{aligned}$$

The following relationship applies to the preceding equation:

$$g(sh, y, \alpha_j) = \Phi\left(\frac{sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

In this equation, $\Gamma(.)$, $\phi(.)$, and $\Phi(.)$, are the gamma function, the density, and the CDF from the standard normal distribution, respectively.

For $\nu = \infty$, the distribution reduces to this equation.

$$r(|t_1| < h, |t_2| < h, ..., |t_k| < h) = \int_0^\infty \prod_{j=0}^{j=k} g(h, y, \alpha_j)\phi(y)dy$$

The following relationship applies to the preceding equation:

$$g(h, y, \alpha_j) = \Phi\left(\frac{h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

For the balanced case, the distribution reduces to the following equation:

$$r(|t_1| < h, |t_2| < h, ..., |t_n| < h) = \int_0^\infty f(h, y, \rho)^n \phi(y)dy$$

The following relationships apply to the preceding equation:

$$f(h, y, \rho) = \Phi\left(\frac{h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right) - \Phi\left(\frac{-h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right)$$

$$\rho = \frac{1}{n-1}$$

Here is the syntax for this distribution:

x=*probmc*('*anom*', *q*, *p*, *nu*, *n*[, *alpha*$_1$, ..., *alpha*$_n$]);

Arguments

x
is a numeric value with the returned result.

q
is a numeric value that denotes the quantile.

p
is a numeric value that denotes the probability. One of *p* and *q* must be missing.

nu
is a numeric value that denotes the degrees of freedom.

n
is a numeric value that denotes the number of samples.

alpha*$_i$, *i* = 1, ...,*k
are optional numeric values denoting the alpha values from the first equation of this distribution. See "Computing the Analysis of Means" on page 963.

Many-One *t*-Statistics: Dunnett's One-Sided Test

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\lambda_1, \ldots, \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + qx}{\sqrt{1-\lambda_i^2}}\right) dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed $\left(\lambda = \sqrt{\frac{1}{2}}\right)$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y)[\Phi(y + \sqrt{2qx})]^k dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\lambda_1, \ldots, \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + q}{\sqrt{1-\lambda_i^2}}\right) dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed $\left(\lambda = \sqrt{\frac{1}{2}}\right)$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2q})]^k dy$$

Many-One *t*-Statistics: Dunnett's Two-sided Test

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\lambda_1, \ldots, \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + qx}{\sqrt{1 - \lambda_i^2}} \right) - \Phi\left(\frac{\lambda_i y - qx}{\sqrt{1 - \lambda_i^2}} \right) \right] dy \, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2qx}) - \Phi(y - \sqrt{2qx})]^k dy \, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\lambda_1, \ldots, \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + q}{\sqrt{1 - \lambda_i^2}} \right) - \Phi\left(\frac{\lambda_i y - q}{\sqrt{1 - \lambda_i^2}} \right) \right] dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2q}) - \Phi(y - \sqrt{2q})]^k dy$$

Computing the Partitioned Range

RANGE applies to the distribution of the studentized range for *n* group means. PARTRANGE applies to the distribution of the partitioned studentized range. Let the *n* groups be partitioned into *k* subsets of size $n_1 + \ldots + n_k = n$. Then the partitioned range is the maximum of the studentized ranges in the respective subsets. The studentization factor is the same in all cases.

$$prob = \int_{0}^{\infty} \prod_{i=1}^{i=k} \left(\int_{-\infty}^{\infty} k\phi(y)(\Phi(y) - \Phi(y - qx))^{k-1} dy \right)^{n_i} d\mu_\nu(x)$$

Here is the syntax for this distribution:

x=probmc('partrange', q, p,nu, k, n_1,...,n_k);

Arguments

x
is a numeric value with the returned result (either the probability or the quantile).

q
is a numeric value that denotes the quantile.

p
is a numeric value that denotes the probability. One of *p* and *q* must be missing.

nu
is a numeric value that denotes the degrees of freedom.

k
is a numeric value that denotes the number of groups.

n_i i=1,...,k
are optional numeric values that denote the *n* values from the equation in this distribution. See "Computing the Partitioned Range" on page 965.

The Studentized Range

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty} \int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y - qx)]^{k-1} dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are σ_1, ..., σ_k, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \sum_{j=1}^{k} \left\{ \prod_{i=1}^{k} \left[\Phi\left(\frac{y}{\sigma_i}\right) - \Phi\left(\frac{y-q}{\sigma_i}\right) \right] \right\} \phi\left(\frac{y}{\sigma_j}\right) \frac{1}{\sigma_j} dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y - q)]^{k-1} dy$$

The Studentized Maximum Modulus

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are σ_1, ..., σ_k, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty} \prod_{i=1}^{k} \left[2\Phi\left(\frac{qx}{\sigma_i}\right) - 1 \right] d\mu_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty} [2\Phi(qx) - 1]^k d\mu_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are σ_1, ..., σ_k, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \prod_{i=1}^{k} \left[2\Phi\left(\frac{q}{\sigma_i}\right) - 1\right]$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = [2\Phi(q) - 1]^k$$

Williams' Test

PROBMC computes the probabilities or quantiles from the distribution defined in Williams (1971, 1972). See ""References"" in *SAS Functions and CALL Routines: Reference*.The need for the Williams' Test arises when you compare the dose treatment means with a control mean to determine the lowest effective dose of treatment.

Note: Williams' Test is computed only for equal sample sizes.

Let X_1, X_2, ..., X_k be identical independent N(0,1) random variables. Let Y_k denote their average given by the following equation.

$$Y_k = \frac{X_1 + X_2 + ... + X_k}{k}$$

It is required to compute the distribution of the following value.

$$(Y_k - Z)/S$$

Arguments

***Y*$_k$**
is as defined previously.

Z
is an N(0,1) independent random variable.

S
is such that ½νS^2 is a χ^2 variable with ν degrees of freedom.

As described in Williams (1971), the full computation is extremely lengthy, and is carried out in three stages. See ""References"" in *SAS Functions and CALL Routines: Reference*.

1 Compute the distribution of Y_k. It is the fundamental (expensive) part of this operation and it can be used to find both the density and the probability of Y_k. Let U_i be defined in this equation.

$$U_i = \frac{X_1 + X_2 + \ldots + X_i}{i},\ i = 1, 2, \ldots, k$$

You can write a recursive expression for the probability of $Y_k > d$. The value of d can be any real number.

$$\begin{aligned}\Pr(Y_k > d) &= \Pr(U_1 > d) \\ &\quad + \Pr(U_2 > d, U_1 < d) \\ &\quad + \Pr(U_3 > d, U_2 < d, U_1 < d) \\ &\quad + \ldots \\ &\quad + \Pr(U_k > d, U_{k-1} < d, \ldots, U_1 < d) \\ &= \Pr(Y_{k-1} > d) + \Pr(X_k + (k-1)U_{k-1} > kd)\end{aligned}$$

To compute this probability, start from an N(0,1) density function.

$$D(U_1 = x) = \phi(x)$$

And recursively compute the convolution.

$$D(U_k = x, U_{k-1} < d, \ldots, U_1 < d) =$$
$$\int_{-\infty}^{d} D(U_{k-1} = y, U_{k-2} < d, \ldots, U_1 < d)(k-1)\phi(kx - (k-1)y)dy$$

From this sequential convolution, it is possible to compute all the elements of the recursive equation for $\Pr(Y_k < d)$, shown previously.

2 Compute the distribution of $Y_k - Z$. This computation involves another convolution to compute the probability.

$$\Pr((Y_k - Z) > d) = \int_{-\infty}^{\infty} \Pr(Y_k > \sqrt{2d} + y)\phi(y)dy$$

3 Compute the distribution of $(Y_k - Z)/S$. This computation involves another convolution to compute the probability.

$$\Pr((Y_k - Z) > tS) = \int_{0}^{\infty} \Pr((Y_k - Z) > ty)d\mu_\nu(y)$$

The third stage is not needed when v = ∞. Due to the complexity of the operations, this lengthy algorithm is replaced by a much faster one when $k \le 15$ for both finite and infinite degrees of freedom v. For $k \ge 16$, the lengthy computation is carried out. It is extremely expensive and very slow due to the complexity of the algorithm.

Comparisons

The MEANS statement in the GLM Procedure of SAS/STAT Software computes the following tests:

- Dunnett's one-sided test
- Dunnett's two-sided test
- Studentized Range

Examples

Example 1: Computing Probabilities By Using PROBMC

The following program shows how to compute probabilities.

```
data _null_;
   dcl double par[5];
   dcl double df q prob;
   dcl char(20) test[3];
   dcl int i;
   method run();
      par := (.5 .51 .55 .45 .2);
      df = 40;
      q = 1;
      test := ('dunnett1' 'dunnett2' 'maxmod');
      do i = 1 to dim(test);
         prob=probmc(test[i], q, ., df, 5, par[1], par[2], par[3],
par[4],
            par[5]);
         put test[i] $10. df q e18.13 prob e18.13;
      end;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
dunnett1  40  1.00000000000E+00 4.82992196083E-01
dunnett2  40  1.00000000000E+00 1.64023105316E-01
maxmod    40  1.00000000000E+00 8.02784203408E-01
```

Example 2: Computing the Analysis of Means

The following program shows how to compute the analysis of means.

```
data _null_;
   dcl double q1 q2 q3 q4;
   method run();
      q1=probmc('anom',.,0.9,.,20);
      put q1=;
      q2=probmc('anom',.,0.9,20,5,0.1,0.1,0.1,0.1,0.1);
      put q2=;
      q3=probmc('anom',.,0.9,20,5,0.5,0.5,0.5,0.5,0.5);
      put q3=;
      q4=probmc('anom',.,0.9,20,5,0.1,0.2,0.3,0.4,0.5);
      put q4=;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
q1=2.78950610163346
q2=2.45499619666786
q3=2.45499619666786
q4=2.45323199941123
```

Example 3: Computing the Partitioned Range

The following program shows how to compute the partitioned range.

```
data _null_;
   dcl double q1 q2;
   method run();
      q1=probmc('partrange',.,0.9,.,4,3,4,5,6);
      put q1=;
      q2=probmc('partrange',.,0.9,12,4,3,4,5,6);
      put q2=;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
q1=4.1022397989482
q2=4.7888626337511
```

Example 4: Computing Williams' Test

In the following program, a substance has been tested at seven levels in a randomized block design of eight blocks. The observed treatment means are as follows:

Treatment	Mean
X_0	10.4
X_1	9.9
X_2	10.0
X_3	10.6
X_4	11.4
X_5	11.9
X_6	11.7

The mean square, with (7 – 1)(8 – 1) = 42 degrees of freedom, is $s^2 = 1.16$.

Determine the maximum likelihood estimates M_i through the averaging process.

- Because $X_0 > X_1$, form $X_{0,1} = (X_0 + X_1)/2 = 10.15$.

- Because $X_{0,1} > X_2$, form $X_{0,1,2} = (X_0 + X_1 + X_2)/3 = (2X_{0,1} + X_2)/3 = 10.1$.
- $X_{0,1,2} < X_3 < X_4 < X_5$
- Because $X_5 > X_6$, form $X_{5,6} = (X_5 + X_6)/2 = 11.8$.

Now the order restriction is satisfied.

The maximum likelihood estimates under the alternative hypothesis are:

- $M_0 = M_1 = M_2 = X_{0,1,2} = 10.1$
- $M_3 = X_3 = 10.6$
- $M_4 = X_4 = 11.4$
- $M_5 = M_6 = X_{5,6} = 11.8$

Now compute $t = (11.8 - 10.4)/\sqrt{2s^2/8} = 2.60$, and the probability that corresponds to $k = 6$, v = 42, and $t = 2.60$ is .9924467341, which shows strong evidence that there is a response to the substance.

You can also compute the quantiles for the upper 5% and 1% tails, as shown in the following program.

```
data _null_;
   dcl double prob quant5 quant1;
   method run();
      prob=probmc('WILLIAMS',2.6,.,42,6);
      put prob=;
      quant5=probmc('WILLIAMS',.,.95,42,6);
      put quant5=;
      quant1=probmc('WILLIAMS',.,.99,42,6);
      put quant1= ;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
prob=0.99244668715827
quant5=1.80656253603894
quant1=2.49090827298707
```

References

Guirguis, G. H., and R. D. Tobias. 2004. "On the Computation of the Distribution for the Analysis of Means." *Communications in Statistics: Simulation and Computation* 33: 861–887.

Nelson, P. R. 1981. "Numerical evaluation of an equicorrelated multivariate non-central t distribution." *Communications in Statistics: Part B - Simulation and Computation* 10: 41–50.

Nelson, P.R. 1982a. "An Approximation for the Complex Normal Probability Integral." *BIT* 22(1): 94–100.

Nelson, P. R. 1982b. "Exact critical points for the analysis of means." *Communications in Statistics: Part A - Theory and Methods* 11: 699–709.

Nelson, P. R. 1988. "Application of the Analysis of Means." *Proceedings of the SAS Users Group International Conference* 13: 225–230.

Nelson, P. R. 1991. "Numerical evaluation of multivariate normal integrals with correlations." *The Frontiers of Statistical Scientific Theory and Industrial Applications* 2: 97–114.

Nelson, P. R. 1993. "Additional Uses for the Analysis of Means and Extended Tables of Critical Values." *Technometrics* 35: 61–71.

PROBMED Function

Computes cumulative probabilities for the sample median.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBMED(*n*, *x*)

Arguments

n

specifies the sample size.

Data type DOUBLE

x

is the point of interest. That is, the PROBMED function calculates the probability that the median is less than or equal to *x*.

Data type DOUBLE

Details

The PROBMED function computes the probability that the sample median is less than or equal to *x* for a sample of *n* independent, standard normal random variables (mean 0, variance 1).

Let *n* represent the sample size, and $x_{(i)}$ represents the *i*th order statistic. Then, when *n* is odd, the function makes the following calculation:

$$\Pr\left[X_{((n+1)/2)} \le x\right] = \mathrm{I}_{\Phi(x)}\left(\frac{n+1}{2}, \frac{n+1}{2}\right)$$

The following equations refer to the preceding equation:

$$\mathrm{I}_p(a,b) = \frac{\mathrm{I}}{\mathrm{B}(a,b)} \int_0^p t^{a-1}(\mathrm{I}-t)^{b-1} dt$$

In the equation $\mathrm{B}(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b)$, Γ(.) is the gamma function. If *n* is even, the PROBMED function performs the following calculation:

$$\Pr[\frac{X_{(n/2)} + X_{((n/2)+1)}}{2} \le x] =$$

$$\frac{2}{\mathrm{B}(\frac{n}{2}, \frac{n}{2})} \int_{-\infty}^{x} \{[\mathrm{I} - \Phi(u)]^{n/2} - [\mathrm{I} - \Phi(2x-u)]^{n/2}\}[\Phi(u)]^{(n/2)-1} \phi(u) du$$

In this equation, $\mathrm{B}(n/2,\ n/2) = [\Gamma(n/2)]^2/\Gamma(n)$, and Φ(.) and ϕ(.) are the standard normal cumulative distribution function and density function, respectively.

Example

```
data _null_;
   dcl double b;
   method run();
      b=probmed(5,-0.1);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.42563808966747
```

References

David, H.A. 1981. *Order Statistics*. 2nd ed. New York, New York: John Wiley & Sons.

PROBNEGB Function

Returns the probability from a negative binomial distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBNEGB(*p*, *n*, *m*)

Arguments

p

is a numeric probability of success parameter.

Range $0 \le p \le 1$

Data type DOUBLE

n

is the number of successes parameter. This argument must be a whole number.

Range $n \ge 1$

Data type DOUBLE

m

is the random variable, the number of failures. This argument must be a positive, whole number.

Range $m \ge 0$

Data type DOUBLE

Details

The PROBNEGB function returns the probability that an observation from a negative binomial distribution, with probability of success *p* and number of successes *n*, is less than or equal to *m*.

To compute the probability that an observation is equal to a given value *m*, compute the difference of two probabilities from the negative binomial distribution for *m* and *m*−1.

Example

The following program illustrates the PROBNEGB function:

```
data _null_;
   dcl double x;
   method run();
```

```
      x=probnegb(0.5,2,1);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.5
```

PROBNORM Function

Returns the probability from the standard normal distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBNORM(*x*)

Arguments

x
is a numeric random variable.

Data type DOUBLE

Details

The PROBNORM function returns the probability that an observation from the standard normal distribution is less than or equal to *x*.

Note: PROBNORM is the inverse of the PROBIT function.

Example

The following program illustrates the PROBNORM function:

```
data _null_;
   dcl double x;
   method run();
      x=probnorm(1.96);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.97500210485177
```

PROBT Function

Returns the probability from a *t* distribution.

Categories: CAS
Probability

Returned data type: DOUBLE

Syntax

PROBT(*x*, *df*[, *nc*])

Arguments

x
is a numeric random variable.

Data type DOUBLE

df
is a numeric degrees of freedom parameter.

Range *df* > 0

Data type DOUBLE

nc
is a numeric noncentrality parameter.

Data type DOUBLE

Details

The PROBT function returns the probability that an observation from a Student's *t* distribution, with degrees of freedom *df* and noncentrality parameter *nc*, is less than or equal to *x*. This function accepts a noninteger degree of freedom parameter *df*. If the optional parameter, *nc*, is not specified or has the value 0, the value that is returned is from the central Student's *t* distribution.

The significance level of a two-tailed *t* test is given by the following equation.

```
p=(1-probt(abs(x),df))*2;
```

Example

The following program illustrates the PROBT function:

```
data _null_;
   dcl double x;
   method run();
      x=probt(0.9,5);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.79531439982768
```

PRXCHANGE Function

Performs a pattern-matching replacement.

Categories: CAS
Character String Matching

Restriction: This function is superseded by the PRCX packages except on the z/OS and 32-bit Windows operating systems.

Returned data type: CHAR

Note: SAS has adopted the International Components for Unicode (ICU) to implement regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXCHANGE(*perl-regular-expression* | *regular-expression-id*, *times*, *source*)

Arguments

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

Data type CHAR

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned from the PRXPARSE function.

Restriction If you use this argument, you must also use the PRXPARSE function.

Data type INTEGER

times
is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern.

Data type INTEGER

Tip If the value of *times* is –1, then matching patterns continue to be replaced until the end of *source* is reached.

source
specifies a character constant, variable, or expression that you want to search.

Data type CHAR

Details

The Basics

If you use *regular-expression-id*, the PRXCHANGE function searches the *source* with the *regular-expression-id* that is returned by PRXPARSE. It returns the value in *source* with the changes that were specified by the regular expression. If there is no match, PRXCHANGE returns the unchanged value in *source*.

If you use *perl-regular-expression*, PRXCHANGE searches the *source* with the *perl-regular-expression*, and you do not need to call PRXPARSE. You can use PRXCHANGE with a *perl-regular-expression* in a WHERE clause and in PROC SQL.

Note: The following restrictions apply to PRX functions in DS2:

- Only **`m`**, **`i`**, **`s`**, and **`x`** can be used in the PRX form /.../.../ that can be preceded or followed by a single character.

- The matching mode modifiers **p**, **o**, **c**, **a**, and **l** are not supported.
- The matching mode modifier **g** is supported.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " in *SAS Functions and CALL Routines: Reference*.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant or if it uses the /o option, then the Perl regular expression is compiled once, and each use of PRXCHANGE reuses the compiled expression. If *perl-regular-expression* is not a constant and if it does not use the /o option, then the Perl regular expression is recompiled for each call to PRXCHANGE.

Note: The compile-once behavior occurs when you use PRXCHANGE in a DS2 environment, in a WHERE clause, or in PROC SQL. For all other uses, the *perl-regular-expression* is recompiled for each call to PRXCHANGE.

Performing a Match

Perl regular expressions consist of characters and special characters that are called metacharacters. When performing a match, SAS searches a source string for a substring that matches the Perl regular expression that you specify.

To view a short list of Perl regular expression metacharacters that you can use when you build your code, see the table "Tables of Perl Regular Expression (PRX) Metacharacters" in *SAS Functions and CALL Routines: Reference*. You can find a complete list of metacharacters on the Perl website.

Examples

Example 1: Changing the Order of First and Last Names

The following program changes the order of first and last names.

```
/* Create a table that contains a list of names. */
proc ds2;
data names;
   dcl char(32) name;
    method run();
      name='Jones, Fred'; output;
      name='Kavich, Kate'; output;
      name='Turley, Ron'; output;
      name='Dulix, Yolanda'; output;
   end;
enddata;
run;
quit;
```

```
   /* Reverse last and first names */
proc ds2;
data ReversedNames;
   method run();
      set names;
      name=prxchange('s/(\w+), (\w+)/$2 $1/', -1, name);
   end;
enddata;
run;
quit;

proc print data=ReversedNames;
run;
```

Output 7.12 *Results from the PRXCHANGE Function*

The SAS System

Obs	name
1	Fred Jones
2	Kate Kavich
3	Ron Turley
4	Yolanda Dulix

Example 2: Changing a Matched Pattern to a Fixed Value

This program locates a pattern in a variable and replaces the variable with a predefined value. The program finds the phone numbers and replaces them with an informational message.

```
/* Create table that contains confidential information. */
proc ds2;
data a;
   dcl char(95) text;
   method run();
      text='The phone number for Ed is (801)443-9876 but not until
tonight.';
      output;
      text='He can be reached at (910)998-8762 tomorrow for testing
purposes.';
      output;
   end;
enddata;
run;
quit;
proc print data=a;
run;
quit;
```

```
   /* Locate confidential phone numbers and replace them with message
*/
   /* indicating that they have been removed.
*/
proc ds2;
data b;
   method run();
      set a;
      text=prxchange('s/\([2-9]\d\d\) ?[2-9]\d\d-\d\d\d\d/*PHONE
NUMBER REMOVED*/'
         , -1, text);
      put text=;
   end;
enddata;
run;
quit;
```

Output 7.13 *Results Before Changing a Matched Pattern to a Fixed Value*

The SAS System

Obs	text
1	The phone number for Ed is (801)443-9876 but not until tonight.
2	He can be reached at (910)998-8762 tomorrow for testing purposes.

Output 7.14 *Results from Changing a Matched Pattern to a Fixed Value*

The SAS System

Obs	text
1	The phone number for Ed is *PHONE NUMBER REMOVED* but not until tonight.
2	He can be reached at *PHONE NUMBER REMOVED* tomorrow for testing purposes.

See Also

Functions:

- “PRXMATCH Function” on page 981
- “PRXPARSE Function” on page 985
- “PRXPOSN Function” on page 988

PRXMATCH Function

Searches for a pattern match and returns the position at which the pattern is found.

Categories: CAS
Character String Matching

Restriction: This function is superseded by the PRCX packages except on the z/OS and 32-bit Windows operating systems.

Returned data type: INTEGER

Note: SAS has adopted the International Components for Unicode (ICU) to implement regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXMATCH(*perl-regular-expression*, *source*)

Arguments

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

Data type CHAR

source
specifies a character constant, variable, or expression that you want to search.

Data type CHAR

Details

The Basics

When you use *perl-regular-expression*, the PRXMATCH function searches *source* with the *perl-regular-expression* and returns the position at which the string begins. If there is no match, PRXMATCH returns a zero.

You can use PRXMATCH with a Perl regular expression in a WHERE clause and in PROC SQL.

Note: The following restrictions apply to PRX functions in DS2:

- Only `m`, `i`, `s`, and `x` can be used in the PRX form /.../.../ that can be preceded or followed by a single character.
- The matching mode modifiers `p`, `o`, `c`, `a`, and `l` are not supported.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX)" in *SAS Functions and CALL Routines: Reference*.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant or if it uses the /o option, then the Perl regular expression is compiled once and each use of PRXMATCH reuses the compiled expression. If *perl-regular-expression* is not a constant and if it does not use the /o option, then the Perl regular expression is recompiled for each call to PRXMATCH.

Examples

Example 1: Finding the Position of a Substring by Using PRXPARSE

The following program searches a string for a substring, and returns its position in the string.

```
data _null_;
   dcl double patternID position;
   method init();
      /* Use PRXPARSE to compile the Perl regular expression.    */
      patternID=prxparse('/world/');
      /* Use PRXMATCH to find the position of the pattern match. */
      position=prxmatch(patternID, 'Hello world!');
      put position=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
position=7
```

Example 2: Finding the Position of a Substring by Using a Perl Regular Expression

The following program uses a Perl regular expression to search a string (Hello world) for a substring (world) and to return the position of the substring in the string.

```
data _null_;
   dcl double position;
   method run();
      position=prxmatch('/world/', 'Hello world!');
      put position;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
7
```

Example 3: Extracting a ZIP Code

The following program searches each row in a table for a nine-digit ZIP code, and writes those rows to the table ZipPlus4.

Note: The backslash (\) must be preceded by another backslash (\) that acts as an escape character.

```
proc ds2;
data ZipCodes (overwrite=yes);
   dcl char(16) name;
   dcl char(10) zip;
   method run();
      name='Jonathan'; zip='32532-2343'; output;
      name='Seth'; zip='85030'; output;
      name='Kim'; zip='39204'; output;
      name='Samuel'; zip='93849-3843'; output;
   end;
 enddata;
 run;
quit;


proc print data=ZipCodes; run;
quit;

proc ds2;
data ZipPlus4 (overwrite=yes);
   method run();
      set zipcodes;
      select;
         when (prxmatch('/\d{5}-\d{4}/', zip))
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=ZipPlus4; run;
quit;
```

Output 7.15 *Original ZipCodes Table Output*

The SAS System

Obs	name	zip
1	Jonathan	32532-2343
2	Seth	85030
3	Kim	39204
4	Samuel	93849-3843

Output 7.16 *Nine-Digit ZIP Code Output*

The SAS System

Obs	name	zip
1	Jonathan	32532-2343
2	Samuel	93849-3843

See Also

Functions:

- “PRXCHANGE Function” on page 977
- “PRXPARSE Function” on page 985
- “PRXPOSN Function” on page 988

PRXPARSE Function

Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.

Categories: CAS
Character String Matching

Restrictions: This function is superseded by the PRCX packages except on the z/OS and 32-bit Windows operating systems.
Use with other Perl regular expressions.

Returned data type: INTEGER

Note: SAS has adopted the International Components for Unicode (ICU) to implement regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

regular-expression-id=**PRXPARSE**(*perl-regular-expression*)

Arguments

regular-expression-id
is a numeric pattern identifier that is returned by the PRXPARSE function.

Data type INTEGER

perl-regular-expression
specifies a character, constant, variable, or expression with a value that is a Perl regular expression.

Data type CHAR

Details

The Basics

The PRXPARSE function returns a pattern identifier number that is used by other Perl functions to match patterns. If an error occurs in parsing the regular expression, SAS returns a missing value.

PRXPARSE uses metacharacters in constructing a Perl regular expression. To view a table of common metacharacters, see "Tables of Perl Regular Expression (PRX) Metacharacters" in *SAS Functions and CALL Routines: Reference*.

Note: The following restrictions apply to PRX functions in DS2:

- Only `m`, `i`, `s`, and `x` can be used in the PRX form /.../.../ that can be preceded or followed by a single character.
- The matching mode modifiers `p`, `o`, `c`, `a`, and `l` are not supported.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " in *SAS Functions and CALL Routines: Reference*.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant, the Perl regular expression is compiled only once. Successive calls to PRXPARSE will not cause a recompile, but returns the *regular-expression-id* for the regular expression that was already compiled. This behavior simplifies the code because you do not need to use an initialization block (IF _N_ =1) to initialize Perl regular expressions.

Examples

Example 1: Compiling a Perl Regular Expression

The following program uses PRXPARSE to compile the Perl regular expression.

```
data _null_;
   dcl double patternID position;
   method init();
      /* Use PRXPARSE to compile the Perl regular expression.    */
      patternID=prxparse('/world/');
      /* Use PRXMATCH to find the position of the pattern match. */
      position=prxmatch(patternID, 'Hello world!');
      put position=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
position=7
```

Example 2: Using PRXPARSE to Reverse First and Last Names

The following program uses PRXPARSE to reverse the first and last names.

```
data _null_;
   dcl double patternID position;
   dcl char(32) names[4];
   dcl int i;
   method init();
       names := ('Jones, Fred   '
                 'Kavich, Kate  '
                 'Turley, Ron   '
                 'Dulix, Yolanda');

       /* Reverse last and first names */
       do i = 1 to dim(names);
          names[i]=prxchange('s/(\w+), (\w+)/$2 $1/', -1, names[i]);
          put names[i];
       end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Fred Jones
Kate Kavich
Ron Turley
Yolanda Dulix
```

See Also

Functions:

- “PRXCHANGE Function” on page 977
- “PRXMATCH Function” on page 981
- “PRXPOSN Function” on page 988

PRXPOSN Function

Returns a character string that contains the value for a capture buffer.

Categories: CAS
Character String Matching

Restriction: This function is superseded by the PRCX packages except on the z/OS and 32-bit Windows operating systems.

Returned data type: CHAR

Note: SAS has adopted the International Components for Unicode (ICU) to implement regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXPOSN(*regular-expression-id*, *capture-buffer*, *source*)

Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned by the PRXPARSE function.

Data type INTEGER

capture-buffer
is a numeric constant, variable, or expression that identifies the capture buffer for which to retrieve a value:

- If the value of *capture-buffer* is zero, PRXPOSN returns the entire match.

- If the value of *capture-buffer* is between 1 and the number of open parentheses in the regular expression, then PRXPOSN returns the value for that capture buffer.
- If the value of *capture-buffer* is greater than the number of open parentheses, then PRXPOSN returns a missing value.

Data type INTEGER

source
specifies the text from which to extract capture buffers.

Data type CHAR

Details

The PRXPOSN function uses the results of PRXMATCH or PRXCHANGE to return a capture buffer. A match must be found by one of these functions for PRXPOSN to return meaningful information.

Note: The following restrictions apply to PRX functions in DS2:

- Only **m**, **i**, **s**, and **x** can be used in the PRX form /.../.../ that can be preceded or followed by a single character.
- The matching mode modifiers **p**, **o**, **c**, **a**, and **l** are not supported.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " in *SAS Functions and CALL Routines: Reference*.

Examples

Example 1: Extracting First and Last Names

The following program uses PRXPOSN to extract first and last names from a table.

```
proc ds2;
data ReversedNames;
   dcl char(32) name;
   method init();
      name='Jones, Fred'; output;
      name='Kavich, Kate'; output;
      name='Turley, Ron'; output;
      name='Dulix, Yolanda'; output;
   end;
enddata;
run;
quit;

proc ds2;
```

```
data FirstLastNames (overwrite=yes);
   dcl char(16) first last;
   dcl double re;
   keep first last;
   retain re;

   method init();
      dcl varchar(32) expression;
      expression = '/(\w+), (\w+)/';
      re=prxparse(expression);
      if missing( re ) then do;
         put 'ERROR: Invalid expression ' expression;
         stop;
      end;
   end;

   method run();
      set ReversedNames;
      if prxmatch(re, name) then
         do;
            last=prxposn(re, 1, name);
            first=prxposn(re, 2, name);
         end;
   end;
enddata;
run;
quit;

proc print data=FirstLastNames;
run;
quit;
```

Output 7.17 *Output from PRXPOSN: First and Last Names*

The SAS System

Obs	first	last
1	Fred	Jones
2	Kate	Kavich
3	Ron	Turley
4	Yolanda	Dulix

Example 2: Extracting Names When Some Names Are Invalid

The following program creates a table that contains a list of names. Rows that have only a first name or only a last name are invalid. PRXPOSN extracts the valid names from the table, and writes the names to the table NEW.

```
proc ds2;
data old (overwrite=yes);
   dcl char(60) name;
   method init();
```

```
      name='Judith S Reaveley'; output;
      name='Ralph F. Morgan'; output;
      name='Jess Ennis'; output;
      name='Carol Echols'; output;
      name='Kelly Hansen Huff'; output;
      name='Judith'; output;
      name='Nick'; output;
      name='Jones'; output;
   end;
enddata;
run;
quit;

proc ds2;
data new (overwrite=yes);
   dcl char(40) first middle last;
   keep first middle last;

   method run();
      re=prxparse('/(\S+)\s+([^\s]+\s+)?(\S+)/i');
      set old;
      if prxmatch(re, name) then
         do;
            first=prxposn(re, 1, name);
            middle=prxposn(re, 2, name);
            last=prxposn(re, 3, name);
            output;
         end;
   end;
enddata;
run;
quit;

proc print data=new;
run;
quit;
```

Output 7.18 *Output of Valid Names*

The SAS System

Obs	first	middle	last
1	Judith	S	Reaveley
2	Ralph	F.	Morgan
3	Jess		Ennis
4	Carol		Echols
5	Kelly	Hansen	Huff

See Also

Functions:

- “PRXCHANGE Function” on page 977
- “PRXMATCH Function” on page 981
- “PRXPARSE Function” on page 985

PUT Function

Returns a value using a specified format.

Categories: CAS
Special

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

PUT(*expression*, *format*)

Arguments

expression
specifies any valid expression.

Data type: DOUBLE, DATE, TIME, TIMESTAMP, CHAR, NCHAR, NVARCHAR, VARCHAR

See: “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

format.
specifies either a DS2 format or a user-defined format that you want applied to *expression*.

To override the default alignment, you can add an alignment specification to a format:

- -L left aligns the value
- -C centers the value
- -R right aligns the value

Details

If a value is not specified for the format width or decimal specification, DS2 uses the default values for that format. Note that when using input-width aware formats, the width, decimal specification, or both is calculated from the input field width. For more information, see "Write Formatted Data in DS2" on page 61.

If *expression* is not a valid data type for the format type (either numeric or character), DS2 converts *expression* to a valid data type for *format.*, with these exceptions:

- date and time expressions are converted to a SAS date, time, or datetime DOUBLE value for numeric formats, and converted to NCHAR for character string formats
- when the format is a binary character format such as $BINARY, $HEX or $OCTAL, expressions with a data type of DOUBLE are converted to NCHAR
- an error is issued when an expression with a data type of VARBINARY is used with a numeric format that does not produce a data type of VARBINARY

When DS2 converts an expression's data type in an assignment statement, the result is left-aligned.

You can use the PUT function to convert a numeric value to a character value and to convert a date, time, or timestamp value to a SAS date/time value.

Comparisons

The PUT function and the PUT statement have similar behavior. However, the PUT statement directs its results to the SAS log whereas the PUT function returns an NCHAR value containing the result of formatting its argument.

Examples

Example 1: Formatting Values with PUT

The following programs illustrate the PUT function:

```
data test(overwrite=yes);
   dcl char x;
   method run();
      x=put(17180, date7.);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
14JAN07
```

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=put('AB', $binary.);
      put x;
      end;
enddata;
run;
```

SAS writes the following output to the log:

```
100000101000010
```

```
data test(overwrite=yes);
   dcl double a x;
   method run();
      a=35436745.3354;
      x=put(a, comma20.4);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
   35,436,745.3354
```

```
data test(overwrite=yes);
   dcl double a x;
   method run();
      a=35436745.3354;
      x=put(a, comma20.4 -L);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
35,436,745.3354
```

Example 2: Using a PROC FORMAT Format

The following program creates a user-defined format and uses that format with the PUT function.

```
proc format;
value abc 1="Yes" 2="No";
run;
quit;

proc ds2;
data test (overwrite=yes);
   dcl double d e;
   dcl char(3) x y;
   method run();
```

```
        d=1;
        x=put(d, abc.);
        put x=;
        e=2;
        y=put(e, abc.);
        put y=;
    end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
x=Yes
y=No
```

See Also

Functions:

- “INPUTC Function” on page 691
- “INPUTN Function” on page 693

PVP Function

Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

PVP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal per-year coupon rate, expressed as a fraction.

Range $0 \leq c < 1$

Data type DOUBLE

n

specifies the number of coupons per year.

Range $n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons.

Range $K > 0$

Data type DOUBLE

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of years.

Range $0 < k_0 \leq \frac{1}{n}$

Data type DOUBLE

y

specifies the nominal per-year yield-to-maturity, expressed as a fraction.

Range $y > 0$

Data type DOUBLE

Details

The PVP function is based on the following relationship:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

Example

The following program illustrates the PVP function:

```
data _null_;
   dcl double p;
   method run();
      p=pvp(1000,.01,4,14,.33/2,.10);
      put p;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
743.167613519067
```

QTR Function

Returns the quarter of the year from a SAS date value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

QTR(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The QTR function returns a value of 1, 2, 3, or 4 from a SAS date value to indicate the quarter of the year in which a date value falls.

For more information about how DS2 handles date and time values, see "DS2 Expressions" in *SAS DS2 Programmer's Guide*.

Example

The following program illustrates the QTR function:

```
data _null_;
   dcl double a b c;
   method run();
      a=17180;
      b=put(a, date7.);
      c=qtr(a);
      put c;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1
```

See Also

Functions:

- "YYQ Function" on page 1191

QUANTILE Function

Returns the quantile from a distribution when you specify the left probability (CDF).

Categories: CAS
Quantile

Returned data type: Double

See: "CDF Function" on page 370

Syntax

QUANTILE(*'distribution'*, *probability*, *parameter-1, ..., parameter-k*)

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the QUANTILE distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL' | 'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'

Distribution	Argument
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

probability
is a numeric constant, variable, or expression that specifies the value of a random variable.

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The QUANTILE function computes the quantile from the specified continuous or discrete distribution, based on the probability value that is provided. For more information, see the individual distributions noted in the table above.

The Conway-Maxwell-Poisson distribution for the QUANTILE function returns the counts value *y* that is the largest whole number whose CDF value is less than or equal to *p*. The syntax for the Conway-Maxwell-Poisson distribution in the QUANTILE function has the following form:

QUANTILE('CONMAXPOI',*p*,*λ*,*v*)

p
is a real number between 0 and 1, inclusively.

λ
is similar to the mean, as in the Poisson distribution.

v
is a dispersion parameter.

For more information, see "Conway-Maxwell-Poisson" distribution in the PDF function on page 898.

Example

The following program illustrates the QUANTILE function:

```
data _null_;
   dcl double a b c d e f g h i j k l m n o p q r x t;
   method init();
      a=quantile('BERN', .75, .25);
      b=quantile('BETA', 0.1,3,4);
      c=quantile('BINOM',.4, .5, 10);
      d=quantile('CAUCHY', .85);
      e=quantile('CHISQ', .6,11);
      f=quantile('CONMAXPOI',0.2,2.3,.4);
      g=quantile('EXPO', .6);
      h=quantile('F',.8,2,3);
      i=quantile('GAMMA', .4,3);
      j=quantile('GENPOI', .9, 1, .7);
      k=quantile('HYPER', .5, 200, 50, 10);
      l=quantile('LAPLACE', .8);
      m=quantile('LOGISTIC', .7);
      n=quantile('LOGNORMAL', .5);
      o=quantile('NEGB', .5, .5, 2);
      p=quantile('NORMAL', .975);
      q=quantile('NORMALMIX',0.5, 1, 0.2, 1.1,0.1);
      r=quantile('PARETO', .01,1);
      s=quantile('POISSON', .9, 1);
      t=quantile('T', .8, 5);
      u=quantile('TWEEDIE', .8, 5);
      v=quantile('UNIFORM', 0.25);
      w=quantile('WALD', .6, 2);
      x=quantile('WEIBULL', .6, 2);
      put a=;
      put b=;
      put c=;
      put c=;
      put e=;
      put f=;
      put g=;
      put h=;
      put i=;
      put j=;
      put k=;
      put l=;
      put m=;
      put n=;
      put o=;
      put p=;
      put q=;
      put r=;
      put s=;
      put t=;
      put u=;
      put v=;
      put w=;
      put x=;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log:

```
a=0
b=0.2009088788569
c=5
d=1.96261050550515
e=11.5298338409688
f=5
g=0.91629073187415
h=2.88602660731929
i=2.28507690400338
j=9
k=2
l=0.91629073187415
m=0.8472978603872
n=1
o=1
p=1.95996398454005
q=1.1
r=1.01010101010101
s=2
t=0.91954378024082
u=1.26111981969517
v=0.25
w=0.95262099270959
x=0.95723076208099
```

See Also

Functions:

- “CDF Function” on page 370
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Function” on page 886
- “SDF Function” on page 1065
- “SQUANTILE Function” on page 1090

QUOTE Function

Adds double quotation marks to a character value.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

QUOTE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The QUOTE function adds double quotation marks, the default character, to a character value. If double quotation marks are found within the argument, they are doubled in the output.

The length of the receiving variable must be long enough to contain the argument (including trailing blanks), leading and trailing quotation marks, and any embedded quotation marks that are doubled. For example, if the argument is ABC followed by three trailing blanks, then the receiving variable must have a length of at least eight to hold "ABC###". (The character # represents a blank space.) If the receiving field is not long enough, the QUOTE function returns a blank string, and writes an invalid argument note to the SAS log.

A string of characters enclosed in double quotation marks is a DS2 identifier and not a character constant. The double quotation marks become part of the identifier. Quoted identifiers cannot be used to create column names in an output table.

Example

The following program illustrates the QUOTE function:

```
data test(overwrite=yes);
   dcl varchar(6) a b c d;
   dcl varchar(30) e f;
   method run();
      a='A"B';
      b=quote(a);
      put b;
      c='A''B';
      d=quote(c);
      put d;
      e='Paul''s Catering Service     ';
      f=quote(trim(e));
      put f;
   end;
```

```
enddata;
run;
```

SAS writes the following output to the log:

```
"A""B"
"A'B"
"Paul's Catering Service"
```

RANBIN Function

Returns a random variate from a binomial distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('BINOMIAL') function on page 1004 instead.

RANCAU Function

Returns a random variate from a Cauchy distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('CAUCHY') function on page 1004 instead.

RAND Function

Generates random numbers from a distribution that you specify.

Categories: CAS
Random Number

Returned data type: DOUBLE

Syntax

RAND('*distribution*', *parameter-1, ...parameter-k*)

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Function Call	Parameter Values
Bernoulli	`RAND('BERNoulli', prob)`	where (0 <= prob <= 1)
Beta	`RAND('BETA', alpha, beta)`	where (0.01 <= alpha <= 1.5e6) and (0.20 <= beta <= 1.5e6)
Binomial	`RAND('BINOmial', prob, trials)`	where (0 <= prob <= 1) and (0 <= trials)
Cauchy	`RAND('CAUChy')`	
Chi-Square(d)	`RAND('CHISquare', df)`	where 0.03 <= df <= 4e9
Erlang	`RAND('ERLAng', alpha)`	where 1 <= alpha <= 2e9
Exponential	`RAND('EXPOnential', scale )`	where 0 <= scale <= 1e300
Extreme Value	`RAND('EXTReme')`	default m = 0, scale = 1, shape = 0
	`RAND('EXTReme', mu)`	where –1e300 <= mu <= 1e300
	`RAND('EXTReme', mu, scale)`	where scale > 0
	`RAND('EXTReme', mu, scale, shape)`	where (scale > 0) and (–0.5 <= shape <= 5)
F	`RAND('F', numdf, dendf)`	where (0.025 <= numdf <= 1e7) and (0.1 <= dendf <= 2e9)
Gamma	`RAND('GAMMa', alpha)`	where 0.015 <= alpha <= 2e9, default scale = 1
	`RAND('GAMMa', alpha, scale)`	where (1e–80 <= scale <= 1e295) or (scale = 0)
Geometric	`RAND('GEOMetric', prob)`	where 0 < prob <= 1
Gompertz	`RAND('GOMPertz')`	default shape = 1, scale = 1
	`RAND('GOMPertz' shape )`	where shape >= 1e–300
	`RAND('GOMPertz' shape , scale )`	where (shape >= 1e–300) and (scale >= 0)
Gumbel	`RAND('GUMBel')`	default mu = 0, scale = 1
	`RAND('GUMBel', mu)`	where –1e300 <= mu <= 1e300
	`RAND('GUMBel', mu, scale)`	where (-1e300 <= mu <= 1e300) and (scale >= 0)

Distribution	Function Call	Parameter Values
Hypergeometric	`RAND('HYPErgeometric', pop_size, cat_size, samp_size)`	where 1 <= pop_size <= 9e15 and 0 <= cat_size <= pop_size and 1 <= samp_size <= pop_size
Uniform Integer	`RAND('INTEger', int)`	where –2147483648 <= int <= 2147483647
Laplace	`RAND('LAPLace')`	default mu = 0, scale = 1
	`RAND('LAPLace', mu)`	where –1e300 <= mu <= 1e300
	`RAND('LAPLace', mu, scale)`	where (–1e300 <= mu <= 1e300) and (scale >= 0)
Logistic	`RAND('LOGIstic')`	default mu = 0, scale = 1
	`RAND('LOGIstic', mu)`	where –1e300 <= mu <= 1e300
	`RAND('LOGIstic', mu, scale)`	where (–1e300 <= mu <= 1e300) and (scale >= 0)
Log-normal	`RAND('LOGNormal')`	default mu = 0, sigma = 1
	`RAND('LOGNormal', mu)`	where ABS(mu) <= 702
	`RAND('LOGNormal', mu, sigma)`	where (ABS(mu) + 7*sigma <= 709) and ((sigma >= max(1, ABS(mu)) * 1e–13) or (sigma = 0))
Negative Binomial	`RAND('NEGBinomial', prob, n)`	where (0 < prob <= 1) and (n >= 1)
Normal or Gaussian	`RAND('NORMal')`	default mu = 0, sigma = 1
	`RAND('NORMal', mu)`	where ABS(mu) <= 1e14
	`RAND('NORMal', mu, sigma)`	where (ABS(mu) <= 1e14*sigma) or (sigma = 0)
Pareto	`RAND('PAREto')`	default shape = 1, scale = 1
	`RAND('PAREto', shape)`	where shape .04 <= xi <= 1e10
	`RAND('PAREto', shape, scale)`	where (shape .04 <= xi <= 1e10) and (0 <= scale <=1e100)
Poisson	`RAND('POISson', mean)`	where 0 <= mean <= 1e300

Distribution	Function Call	Parameter Values
Shifted Gompertz	`RAND('SHGOmpertz')`	default shape = 1, inverse_scale = 1
	`RAND('SHGOmpertz', shape)`	where shape >= 1e–300
	`RAND('SHGOmpertz', shape, inverse_scale)`	where (shape >= 1e–300) and (inverse_scale >= 1e-300)
T	`RAND('T', df)`	where df >= 0.05
Tabled	`RAND('TABLe', p_1, ..., p_n)`	where (0 <= pi <= 1) and (p_1 + ... + p_n <= 1)
Triangular	`RAND('TRIAngular', height)`	where 0 <= height <= 1
Uniform	`RAND('UNIForm')`	default a = 1, b = 0
	`RAND('UNIForm', a)`	uniform on (min(a,0) max(a,0))
	`RAND('UNIForm', a, b)`	uniform on (min(a,b) max(a,b))
Wald (Inverse Gaussian)	`RAND('WALD', shape)`	where 1e–18 <= shape <= 1e17, default mu = 1
	`RAND('WALD', shape, mu)`	where (shape >= 1e–18) and (mu >= 1e–10) and (1e–21 <= shape/mu <= 1e17)
Weibull	`RAND('WEIBull', shape)`	where 0.02 <= shape <= 1e13, default scale = 1
	`RAND('WEIBull', shape, scale)`	where (0.02 <= shape <= 1e13) and (1e–100 <= scale <= 1e20)

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note Except for T and F, you can minimally identify any distribution by its first four characters.

parameter-1, ...parameter-k
are optional numeric constants, variables, or expressions that specify the values of *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

See "Details" on page 1008

Details

Generating Random Numbers

The RAND function generates random numbers from various continuous and discrete distributions. Wherever possible, the simplest form of the distribution is used.

The RAND function uses the Mersenne-Twister random number generator (RNG) that was developed by Matsumoto and Nishimura (1998). Other random number generators can be specified by the STREAMINIT function.

The RAND function is started with a single seed. However, the state of the process cannot be captured by a single seed. You cannot stop and restart the generator from its stopping point.

For the default 32-bit Mersenne Twister, if the initial seed is exactly divisible by 8192, the RAND function uses the 2002 initialization algorithm (Matsumoto and Nishimura, 2002). Otherwise, RAND uses the 1998 initialization algorithm for compatibility with previous releases.

Reproducing a Random Number Stream

If you want to create reproducible streams of random numbers, then use the STREAMINIT function before any invocation of the RAND function to specify a seed value for random number generation. Use the STREAMINIT function once per data program.For more information, see the example in “STREAMINIT Function” on page 1096.

When called from a DS2 program, the following FCMP functions do not recognize seed values that are set in the DS2 program.

COMPCOST
COMPGED
RAND
STREAMINIT

In a DATA step, all random numbers from RAND are drawn from the same stream by default, but you can create multiple independent streams by using the CALL STREAM routine.

In PROC DS2, random numbers from RAND that are called inside an FCMP package come from a separate stream. To get reproducible random numbers, use the CALL STREAMINIT routine inside each FCMP function that calls RAND. To get independent streams, also use the CALL STREAM routine inside each FCMP function that calls RAND. If you do so, the results from DS2 will be consistent with a DATA step. For an example, see “Considerations and Limitations When Using the FCMP Package” in *SAS DS2 Programmer’s Guide*.

Duplicate Values

The RNG algorithms used by the RAND function have extremely long periods, but this does not imply that large random samples are devoid of duplicate values. With the default 32-bit Mersenne Twister algorithm, the RAND function returns at most

2^{32} distinct values. In a random uniform sample of size 10^5, the chance of drawing at least one duplicate is greater than 50%. The expected number of duplicates in a random uniform sample of size M is approximately $M^2/2^{33}$ when M is much less than 2^{32}. For example, you should expect about 115 duplicates in a random uniform sample of size $M=10^6$. These results are consequences of the famous "birthday matching problem" in probability theory.

For a 64-bit RNG, RAND('UNIFORM') can return at least $2^{53} - 1$ distinct values. For a sample size of 10^8, the chance of drawing at least one duplicate is greater than 50%. For a sample size of 10^9, the expected number of duplicates is about 55.55.

Extreme Parameter Values

Pseudorandom numbers are generated by using numerical algorithms that transform uniform random variates. For some distributions, very small or very large parameter values can result in pseudorandom variates that do not adequately follow the theoretical distribution. For other distributions, extreme parameter values can lead to numerical underflow or overflow during a computation. The RAND function attempts to identify regions of parameter space that might lead to these problems.

If the RAND function determines that extreme parameters might lead to invalid pseudorandom variates, it returns a missing value and writes a warning message to the SAS log. For example, a small value of the parameter in the chi-square distribution (such as d=0.01) can produce the following warning message and cause the RAND function to return a missing value:

```
WARNING: In the function call, RAND('CHISQUARE', 0.01), there is a risk
         of underflow that would cause the distribution of the random
         numbers to be wrong. It is recommended that you use
         RAND('CHISQUARE', df), where 0.03 <= df <= 4e9.
```

To suppress the warning and cause the RAND function to generate random numbers (despite the possibility of a poor distribution), append a question mark to the end of the name of the distribution that is specified in the first argument. An example is RAND('CHISQUARE?',.01).

Bernoulli Distribution

x=**RAND**('BERNOULLI', *p*)

Arguments

x

is an observation from the Bernoulli distribution with the following probability density function:

$$f(x) = \begin{cases} 1 & p = 0, x = 0 \\ p^x(1-p)^{1-x} & 0 < p < 1, x = 0, 1 \\ 1 & p = 1, x = 1 \end{cases}$$

Range *x*=0, 1

p

is a numeric probability of success.

Range $0 \le p \le 1$

Beta Distribution

x = **RAND**('BETA', *a*, *b*)

Arguments

x

is an observation from the Beta distribution with the following probability density function:

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$

Range $0 < x < 1$

a

is a numeric shape parameter.

Range $a > 0$

b

is a numeric shape parameter.

Range $b > 0$

Binomial Distribution

x = **RAND**('BINOMIAL', *p*, *n*)

Arguments

x

is an observation that is a whole number from the Binomial distribution with the following probability density function:

$$f(x) = \begin{cases} 1 & p = 0, x = 0 \\ \binom{n}{x} p^x (1-p)^{n-x} & 0 < p < 1, x = 0, \ldots, n \\ 1 & p = 1, x = n \end{cases}$$

Range $x = 0, 1, \ldots, n$

p

is a numeric probability of success.

Range $0 \le p \le 1$

n

is a parameter that counts the number of independent Bernoulli trials. This argument must be a whole number.

Range $n = 1, 2, \ldots$

Cauchy Distribution

x = **RAND**('CAUCHY')

Arguments

x

is an observation from the Cauchy distribution with the following probability density function:

$$f(x) = \frac{1}{\pi\left(1 + x^2\right)}$$

Range $-\infty < x < \infty$

Chi-Square Distribution

x = **RAND**('CHISQUARE', *df*)

Arguments

x

is an observation from the Chi-Square distribution with the following probability density function:

$$f(x) = \frac{2^{-df/2}}{\Gamma\left(\frac{df}{2}\right)} x^{df/2-1} e^{-x/2}$$

Range $x > 0$

df

is a numeric degrees of freedom parameter.

Range $df > 0$

Erlang Distribution

x = **RAND**('ERLANG', *a*)

Arguments

x

is an observation from the Erlang distribution with the following probability density function:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

Range $x > 0$

a

is a numeric shape parameter. This argument must be a whole number.

Range $a = 1, 2, \ldots$

Exponential Distribution

x = **RAND**('EXPONENTIAL' ,[σ])

Arguments

x

is an observation from the Exponential distribution with the following probability density function:

$$f(x) = e^{-x}$$

Range $x > 0$

σ

is a scale parameter.

Default 1

Range σ > 0

Extreme Value Distribution

x=**RAND**('EXTRVALUE', [μ, σ, ξ])

Arguments

x

is an observation from the extreme value distribution, which has the following cumulative probability distribution:

$$F(x) = \exp(t(x))$$
$$where: t(x) = (1 + \xi(x - \mu)/\sigma)^{-1/\xi} \; for \; \xi \neq 0 \; and \; t(x) = \exp(-(x - \mu)/\sigma) \; for \; \xi = 0$$

When ξ=0, the distribution is a Type 1 distribution, sometimes called a Gumbel-type distribution. The random values of x are in the interval (–∞, ∞). When ξ=0, it is more efficient to use the GUMBEL option in the RAND function. When ξ>0, the distribution is a Type 2 distribution, sometimes called a Fréchet-type distribution. The random values of x are in the interval (μ– σ/ξ, ∞). When ξ<0, the distribution is a Type 3 distribution, sometimes called a Weibull-type distribution. The random values of x are in the interval (–∞, μ– σ/ξ).

μ

is a numeric location parameter.

Default 0

σ

is a numeric scale parameter.

Default 0

Range σ>0

ξ

is a numeric shape parameter.

Default 1

F Distribution

x = **RAND**('F', *n*, *d*)

Arguments

x

is an observation from the F distribution with the following probability density function:

$$f(x) = \frac{\Gamma\left(\frac{n+d}{2}\right)}{\Gamma\left(\frac{n}{2}\right)\Gamma\left(\frac{d}{2}\right)} \frac{n^{n/2} d^{d/2} x^{n/2-1}}{(d+nx)^{(n+d)/2}}$$

Range *x* > 0

n

is a numeric numerator degrees of freedom parameter.

Range *n* > 0

d

is a numeric denominator degrees of freedom parameter.

Range *d* > 0

Gamma Distribution

x = **RAND**('GAMMA', *a* [,λ])

Arguments

x

is an observation from the Gamma distribution with the following probability density function:

$$f(x) = \frac{x^{a-1}}{\lambda a \Gamma(a)} \exp(-x/\lambda)$$

Range *x* > 0

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range *a* > 0

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ>1

Geometric Distribution

x = **RAND**('GEOMETRIC', *p*)

Arguments

x

is a count that denotes the number of trials that are needed to obtain one success. This argument must be a whole number. *X* is an observation from the geometric distribution with the following probability density function:

$$f(x) = \begin{cases} (1-p)^{x-1}p & 0 < p < 1, x = 1, 2, \ldots \\ 1 & p = 1, x = 1 \end{cases}$$

Range *x* = 1, 2, ...

p

is a numeric probability of success.

Range 0 < *p* ≤ 1

Gompertz Distribution

x=**RAND**('GOMPERTZ',[λ,σ])

Arguments

x

is an observation from the Gompertz distribution and has the following cumulative probability distribution:

$$F(x) = 1 - \exp\left(-\lambda \exp\left(1/\sigma\right) - 1\right)$$

The random values of *x* are in the interval (0, ∞).

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

σ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range σ>0

Gumbel Distribution

x=RAND('GUMBEL',[μ, σ])

x

is an observation from the Gumbel distribution, and has the following cumulative probability distribution:

$$F(x) = \exp(-\exp(-(x-\mu)/\sigma))$$

The random values of x are in the interval (-∞, ∞). The Gumbel distribution is an extreme value distribution. The distribution is skewed to the right.

μ

is a numeric constant, variable, or expression that specifies a numeric location parameter.

Default 0

σ

is a numeric constant, variable, or expression that specifies a numeric scale parameter.

Default 1

Range σ>0

Hypergeometric Distribution

x = **RAND**('HYPER', *N*, *R*, *n*)

Arguments

x

is an observation from the hypergeometric distribution with the following probability density function. This argument must be a whole number.

$$f(x) = \frac{\binom{R}{x}\binom{N-R}{n-x}}{\binom{N}{n}}$$

Range *x* = max(0, (*n* – (*N* – *R*))), ..., min(*n*, *R*)

N

is a population size parameter. This argument must be a whole number.

Range *N* = 1, 2, ...

R

is a number of items in the category of interest. This argument must be a whole number.

Range *R* = 0, 1, ..., *N*

n

is a sample size parameter. This argument must be a whole number.

Range *n* = 1, 2, ..., *N*

The hypergeometric distribution is a mathematical formalization of an experiment in which you draw *n* balls from an urn that contains *N* balls, *R* of which are red. The hypergeometric distribution is the distribution of the number of red balls in the sample of *n*.

Integer Distribution

x=**RAND**('INTEGER',*a*,[*b*])

Arguments

x

is a random value from the discrete uniform distribution on a finite set of integers. If you specify one integer parameter, *a*, then *x* is drawn uniformly at random from the set {1,2,..., a-1, a}. If you specify two integer parameters, *a* and *b* with a≤b, then *x* is drawn uniformly at random from the set {a,a+1,..., b-1, b}.

a

is an integer parameter. If you specify only one numeric parameter, *a* is an upper limit for the random values. If you specify two parameters, *a* is a lower limit.

b

is an integer parameter that specifies the upper limit for the random values.

Laplace Distribution

x=**RAND**('LAPLACE',[*θ*,*λ*])

Arguments

x

is an observation from the Laplace distribution and has the following probability density function:

$$f(x) = \frac{1}{2\lambda}\exp\left(-\left|x-\theta\right|/\lambda\right)$$

θ

is an optional location parameter.

Default 0

λ

is an optional scale parameter

Default 0

Range λ>0

Logistic Distribution

x=**RAND**('LOGISTIC',[*θ*,*λ*])

Arguments

x

is an observation from the logistic distribution, which has the following probability density function:

$$f\left(x\right) = \frac{\exp(-(x-\theta)/\lambda)}{\lambda(1+\exp(-(x-\theta)/\lambda))^2}$$

θ

is an optional scale parameter.

Default 0

λ

is an optional scale parameter.

Default 0

Range $\lambda > 0$

Lognormal Distribution

x = **RAND**('LOGNORMAL',[*θ*,*λ*])

Arguments

x

is an observation from the lognormal distribution with the following probability density function:

$$f(x) = \frac{1}{x\lambda\sqrt{2\,\pi}}\exp\left(\frac{-(\ln(x)-\theta)^2}{2\lambda^2}\right)$$

If the random variable *x* is lognormally distributed, then log (*x*) is normally distributed with mean θ and standard deviation λ.

Range $x > 0$

θ

is a numeric constant, variable, or expression that specifies a log-scale parameter.

Default 0

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

Range $\lambda > 0$

Negative Binomial Distribution

x = **RAND**('NEGBINOMIAL', *p*, *k*)

Arguments

x

is an observation from the negative binomial distribution with the following probability density function. This argument must be a whole number.

$$f(x) = \begin{cases} \binom{x+k-1}{k-1}(1-p)^x p^k & 0 < p < 1, x = 0, 1, \ldots \\ 1 & p = 1, x = 0 \end{cases}$$

Range $x = 0, 1, \ldots$

k

is a numeric parameter that is the number of successes. Integer and non-integer *k* values are allowed.

Range $k = 1, 2, \ldots$

p

is a numeric probability of success.

Range $0 < p \leq 1$

The negative binomial distribution is the distribution of the number of failures before *k* successes occur in sequential independent trials, all with the same probability of success, *p*.

Normal Distribution

x=**RAND**('NORMAL' [, *θ*, *λ*])

Arguments

x

is an observation from the normal distribution with a mean of θ and a standard deviation of λ that has the following probability density function:

$$f(x) = \frac{1}{\lambda\sqrt{2\,\pi}}\ exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

Range $-\infty < x < \infty$

θ

is the mean parameter.

Default 0

λ

is the standard deviation parameter.

Default 1

Range $\lambda > 0$

Pareto Distribution

x=**RAND**('PARETO', *a*,[*k*])

Arguments

x

is an observation from the Pareto distribution and has the following probability density function.

$$f\left(x\right) = \frac{a}{k}\left(\frac{k}{x}\right)a + 1$$

a

is a shape parameter.

Range *a*>0

k

is an optional scale parameter.

Default 1

Range *k*>0

Poisson Distribution

x = **RAND**('POISSON', *m*)

Arguments

x

is an observation from the distribution with the following probability density function. This argument must be a whole number.

$$f(x) = \frac{m^x e^{-m}}{x!}$$

Range *x* = 0, 1, ...

m

is a numeric mean parameter.

Range *m* > 0

Shifted Gompertz Distribution

x=**RAND**('SHGOMPERTZ', [*η*,*τ*])

Arguments

x

is an observation from the shifted Gompertz distribution and has the following cumulative probability distribution.

$$F\left(x\right) = \left(1 - \exp\left(-\tau x\right)\right)\exp\left(-\eta\exp\left(-\tau x\right)\right)$$

The random values of x are in the interval (0, ∞). As η→0, the shifted Gompertz distribution approaches the exponential distribution with shape parameter 1/τ.

η

is a shape parameter.

Default 1

Range $\eta>0$

τ

is an inverse scale parameter.

Default 1

Range $\tau>0$

T Distribution

x = **RAND**('T', *df*)

Arguments

x

is an observation from the T distribution with the following probability density function:

$$f(x) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{df\,\pi}\,\Gamma\left(\frac{df}{2}\right)}\left(1+\frac{x^2}{df}\right)^{-\frac{df+1}{2}}$$

Range $-\infty < x < \infty$

df

is a numeric degrees of freedom parameter.

Range $df > 0$

Tabled Distribution

x = **RAND**('TABLE', *p1*, *p2*, ...)

Arguments

x

is an observation from one of the following distributions. This argument must be a whole number.

If $\sum_{i=1}^{n} p_i < 1$, then *x* is an observation from this probability density function:

$$f(i) = p_i, \quad i = 1, 2, \ldots, n$$

and

$$f(n+1) = 1 - \sum_{i=1}^{n} p_i$$

If $\sum_{i=1}^{n} p_i \geq 1$ for some index *n*, then *x* is an observation from this probability density function:

$f(i) = p_i, \quad i = 1, 2, \ldots, n-1$

and

$f(n) = 1 - \sum_{i=1}^{n-1} p_i$

p1, p2, ...

are numeric probability values.

Range 0 ≤ *p1, p2, ...* ≤ 1

Restriction The maximum number of probability parameters depends on your operating environment, but the maximum number of parameters is at least 32,767.

The tabled distribution takes on the values 1, 2, ..., *n* with specified probabilities.

Note: By using the FORMAT statement, you can map the set {1, 2, ..., *n*} to any set of *n* or fewer elements.

Triangular Distribution

x = **RAND**('TRIANGLE', *h*)

Arguments

x

is an observation from the triangular distribution with the following probability density function:

$$f(x) = \begin{cases} \frac{2x}{h} & 0 \leq x \leq h \\ \frac{2(1-x)}{1-h} & h < x \leq 1 \end{cases}$$

In this equation, 0 ≤ *h* ≤ 1.

Range 0 ≤ *x* ≤ 1

Note The distribution can be easily shifted and scaled.

h

is the horizontal location of the peak of the triangle.

Range 0 ≤ *h* ≤ 1

Uniform Distribution

x = **RAND**('UNIFORM', [a,b])

Arguments

x

is an observation from the continuous uniform distribution in the interval (a,b). For a< b, the probability density function on (a,b) is:

$$f(x) = \frac{1}{|b-a|}$$

If you do not specify any parameters, then the interval (0,1) is used. If you specify one parameter, then the interval (0,c) is used, where c is the specified parameter. If you specify two parameters, then a < x < b, where a and b are the specified parameters.

Range 0 < *x* < 1

The RAND function uses the Mersenne-Twister random number generator (RNG) that was developed by Matsumoto and Nishimura (1998). Other random number generators can be specified by the STREAMINIT function.

The RAND function is started with a single seed. However, the state of the process cannot be captured by a single seed. You cannot stop and restart the generator from its stopping point.

For the default 32-bit Mersenne Twister, if the initial seed is exactly divisible by 8192, the RAND function uses the 2002 initialization algorithm (Matsumoto and Nishimura, 2002). Otherwise, RAND uses the 1998 initialization algorithm for compatibility with previous releases.

Wald (Inverse Gaussian) Distribution

x=**RAND**('WALD', *λ*,[*θ*])

x=**RAND**('IGAUSS', *λ*,[*θ*])

Arguments

x

is an observation from the Wald distribution and has the following probability density function:

$$f\left(x\right) = \left(\frac{\lambda}{2\pi x^3}\right)^{1/2} \exp\left(-\frac{\lambda(x-\theta)2}{2\theta^2 x}\right)$$

The random values of x are in the interval (0,∞). Many references, including the MCMC procedure in SAS/STAT software, list θ as the first parameter for the inverse Gaussian distribution. However, θ is the last parameter for the RAND, PDF, CDF, and QUANTILE functions because it is an optional parameter.

λ

is a shape parameter.

Range *λ*>0

θ

is the mean parameter.

Default 1

Range $\theta > 0$

Weibull Distribution

x = **RAND**('WEIBULL', *a*, *b*)

Arguments

x

is an observation from the Weibull distribution with the following probability density function:

$$f(x) = \frac{a}{b^a} x^{a-1} e^{-\left(\frac{x}{b}\right)^a}$$

Range $x \geq 0$

a

is a numeric shape parameter.

Range $a > 0$

b

is a numeric scale parameter.

Range $b > 0$

Example

The following program illustrates the RAND function:

```
data test (overwrite=yes);
   dcl double a b c d e f g h i j k l m n o p q r s t u v;
   dcl double w x y z aa bb cc;
   method run();
      a=rand('BERN', .75);
      b=rand('BETA', 3, 1);
      c=rand('BINOM', 0.75, 10);
      d=rand('CAUCHY');
      e=rand('CHISQ', 22);
      f=rand('ERLANG', 7);
      g=rand('EXTRVAL', 0, 1);
      h=rand('EXPO');
      i=rand('F', 12, 322);
      j=rand('GAMMA', 7.25);
      k=rand('GEOM', 0.02);
      l=rand('GOMP', 1, 0.5);
      m=rand('GUMBEL', 0, 2);
      n=rand('HYPER', 10, 3, 5);
      o=rand('INTEGER', 1, 10);
      p=rand('LAPLACE');
      q=rand('LOGIST');
```

```
      r=rand('LOGN');
      s=rand('NEGB', 0.8, 5);
      t=rand('NORMAL');
      u=rand('PARETO', 3, 1);
      v=rand('POISSON', 6.1);
      w=rand('SHGOMP', 0.5, 1.2);
      x=rand('T', 4);
      y=rand('TABLE', .2, .5);
      z=rand('TRIANGLE',0.7);
      aa=rand('UNIFORM');
      bb=rand('WALD', 0.25, 2.1);
      cc=rand('WEIB', 1, 2);
      put a;
      put b;
      put c;
      put d;
      put e;
      put f;
      put g;
      put h;
      put i;
      put j;
      put k;
      put l;
      put m;
      put n;
      put o;
      put p;
      put q;
      put r;
      put s;
      put t;
      put u;
      put v;
      put w;
      put x;
      put y;
      put z;
      put aa;
      put bb;
      put cc;
   end;
enddata;
run;
```

SAS writes the following output to the log. Because the function is a random number generator, the results vary every time the program is run.

```
0
0.74302422399001
8
-0.53507631686494
16.8091595104245
9.85226977170385
-0.23569561070954
3.91526159715128
0.70109395516368
4.49313340782904
43
0.13847585083914
4.71625869848403
2
8
-0.64278704678732
2.77758410988081
0.65580499717406
4
0.58012595905153
2.07213926731126
3
0.43334467603266
0.38949035606593
2
0.27075405231828
0.01891799294389
0.08049855030189
4.57316490321742
```

References

Matsumoto, M., and T. Nishimura. “Mersenne Twister with Improved Initialization.” 2002. Available Mersenne Twister.

Fishman, George S. 1996. *Monte Carlo: Concepts, Algorithms, and Applications*. New York, USA: Springer-Verlag.

Fushimi, M., and S. Tezuka.1983. . “The k-Distribution of Generalized Feedback Shift Register Pseudorandom Numbers.” *Communications of the ACM* 26: 516–523.

Gentle, James E. 1998. *Random Number Generation and Monte Carlo Methods*. New York, USA: Springer-Verlag.

Lewis, T. G., and W. H. Payne. 1973 . “Generalized Feedback Shift Register Pseudorandom Number Algorithm.” *Journal of the ACM* 20: 456–468.

Matsumoto, M., and Y. Kurita. 1992 . “Twisted GFSR Generators.” *ACM Transactions on Modeling and Computer Simulation* 2: 179–194.

Matsumoto, M., and Y. Kurita. 1994 . “Twisted GFSR Generators II.” *ACM Transactions on Modeling and Computer Simulation* 4: 254–266.

Matsumoto, M., and T. Nishimura. 1998 . “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator.” *ACM Transactions on Modeling and Computer Simulation* 8: 3–30.

Ripley, Brian D. 1987. *Stochastic Simulation*. New York, USA: Wiley.

Robert, C. P., and G. Casella. 1999. *Monte Carlo Statistical Methods*. New York, USA: Springer-Verlag.

Ross, Sheldon M. 199. *Simulation*. San Diego, USA: Academic Press.

RANEXP Function

Returns a random variate from an exponential distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('EXPONENTIAL') function on page 1004 instead.

RANGAM Function

Returns a random variate from a gamma distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('GAMMA') function on page 1004 instead.

RANGE Function

Returns the difference between the largest and the smallest values.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

RANGE(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The RANGE function returns the difference between the largest and the smallest of the non-null or nonmissing arguments.

Example

The following program illustrates the RANGE function:

```
data test(overwrite=yes);
   dcl double x0 x1 x2 x3 x4;
   method run();
      x0=range(., .);
      x1=range(-2, 6, 3);
      x2=range(2, 6 ,3, .);
      x3=range(1, 6, 3, 1);
      x4=range(of x1-x3);
      put x0= x1= x2= x3= x4=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x0=. x1=8 x2=4 x3=5 x4=4
```

RANK Function

Returns the position of a character in the ASCII collating sequence.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

RANK(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The RANK function returns a whole number that represents the position of the character in the ASCII collating sequence. When more than one character is specified, the RANK function returns the position in the ASCII collating sequence for the first character.

Note: Any program that uses the RANK function with characters above ASCII 127 is not portable. (The hexadecimal notation is **'7F'x**.) The program is not portable because these characters are national characters and they vary from country to country.

Example

The following program illustrates the RANK function:

```
data test(overwrite=yes);
   dcl varchar x;
   method run();
      x=rank('A');
      put x;
   end;
enddata;
run;
```

SAS writes the following ASCII output to the log.

```
65
```

See Also

Functions:

- "BYTE Function" on page 353

RANNOR Function

Returns a random variate from a normal distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('NORMAL') function on page 1004 instead.

RANPOI Function

Returns a random variate from a Poisson distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('POISSON') function on page 1004 instead.

RANTBL Function

Returns a random variate from a tabled probability distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('TABLE') function on page 1004 instead.

RANTRI Function

Returns a random variate from a triangular distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('TRIANGLE') function on page 1004 instead.

RANUNI Function

Returns a random variate from a uniform distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('UNIFORM') function on page 1004 instead.

REPEAT Function

Repeats a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

REPEAT(*expression*, *n*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

n
specifies the number of times to repeat *expression*.

Restriction *n* must be greater than or equal to 0.

Data type BIGINT, DOUBLE

Details

The REPEAT function returns a character value consisting of the first argument repeated *n* times. Thus, the first argument appears *n*+1 times in the result.

Example

The following program illustrates the REPEAT function:

```
data test(overwrite=yes);
   dcl varchar(10) x;
   method run();
      x=repeat('ONE', 2);
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
ONEONEONE
```

REVERSE Function

Reverses a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

REVERSE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The REVERSE function returns a character value with the last character in the expression is the first character in the result, the next-to-last character in the expression is the second character in the result, and so on.

Note: Trailing blanks in the expression become leading blanks in the result.

Example

The following program illustrates the REVERSE function:

```
data test(overwrite=yes);
   dcl varchar backward;
   method run();
      backward=reverse('xyz
');
```

```
      put backward=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
backward= zyx
```

RIGHT Function

Right aligns a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

RIGHT(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The RIGHT function returns an argument with trailing blanks moved to the start of the value. The argument's length does not change.

Example

The following programs illustrate the RIGHT function:

```
data test(overwrite=yes);
   dcl char(10) a b c;
```

```
      method run();
         a='Due Date  ';
         b=put(a, $10.);
         c=put(right(a), $10.);
         put b;
         put c;
      end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
Due Date
  Due Date
```

```
   data test(overwrite=yes);
      dcl char(12) a;
      dcl char(15) b;
      method run();
         a='Due Date  ';
         b='*'||right(a);
         put a= $12. b= $15.;
      end;
   run;
```

SAS writes the following output to the log:

```
a=Due Date     b=*    Due Date
```

See Also

Functions:

- “COMPRESS Function” on page 460
- “LEFT Function” on page 781
- “TRIM Function” on page 1151

RMS Function

Returns the root mean square.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

RMS(*expression* [, ...*expression*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the root mean square of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The root mean square is calculated as follows.

$$\sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}$$

Example

The following program illustrates the RMS function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3;
   method run();
      x1=rms(1,
7);
      x2=rms(., 1, 5,
11);

      x3=rms(of x1-x2);
      put x1=;
      put x2=;
      put x3=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=5
x2=7
x3=6.0827625302982
```

ROUND Function

Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

ROUND(*expression* [, *rounding-unit*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value, to be rounded.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

rounding-unit
specifies a positive numeric expression that specifies the rounding unit.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

Basic Concepts

The ROUND function rounds the first argument to a value that is very close to a multiple of the second argument. The results might not be an exact multiple of the second argument.

Differences between Binary and Decimal Arithmetic

Computers use binary arithmetic with finite precision. If you work with numbers that do not have an exact binary representation, computers often produce results that differ slightly from the results that are produced with decimal arithmetic.

For example, the decimal values 0.1 and 0.3 do not have exact binary representations. In decimal arithmetic, 3*0.1 is exactly equal to 0.3, but this equality is not true in binary arithmetic.

As the following example shows, if **a** is a float and **b** is a REAL, there is a difference between the two values.

```
data _null_;
   dcl float a diff;
   dcl real b;
   method run();
      a=0.3;
      b=3*0.1;
      diff=a-b;
      put a=;
      put b=;
      put diff=;
   end;
enddata;
run;
```

The following lines are written to the SAS log:

```
a=0.3
b=0.30000001192092
diff=-1.192092896618E-8
```

Operating Environment Information: The example above was executed in the Windows environment. If you use other operating environments, the results is slightly different.

The Effects of Rounding

Rounding by definition finds an exact multiple of the rounding unit that is closest to the value to be rounded. For example, 0.33 rounded to the nearest tenth equals 3*0.1 or 0.3 in decimal arithmetic. In binary arithmetic, 0.33 rounded to the nearest tenth equals 3*0.1, and not 0.3, because 0.3 is not an exact multiple of one tenth in binary arithmetic.

The ROUND function returns the value that is based on decimal arithmetic, even though this value is sometimes not the exact, mathematically correct result. In the example `ROUND(0.33,0.1)`, ROUND returns 0.3 and not 3*0.1.

Expressing Binary Values

If the characters "0.3" appear as a constant in a DS2 program, the value is computed as 3/10. To be consistent with the standard informat, `ROUND(0.33,0.1)` computes the result as 3/10, and the following statement produces the results that you would expect.

```
if round(x,0.1) = 0.3 then
```

```
   ... more DS2 statements ...
```

However, if you use the variable Y instead of the constant 0.3, as the following statement shows, the results might be unexpected depending on how the variable Y is computed.

```
if round(x,0.1) = y then
   ... more DS2 statements ...
```

If ROUND reads Y as the characters "0.3" using the standard informat, the result is the same as if a constant 0.3 appeared in the IF statement. If ROUND reads Y with a different informat, or if a program other than SAS reads Y, then there is no guarantee that the characters "0.3" would produce a value of exactly 3/10. Imprecision can also be caused by computation involving numbers that do not have exact binary representations, or by porting tables from one operating environment to another that has a different floating-point representation.

If you know that Y is a decimal number with one decimal place, but are not certain that Y has exactly the same value as would be produced by the standard informat, it is better to use the following statement:

```
if round(x,0.1) = round(y,0.1) then
   ... more DS2 statements ...
```

Testing for Approximate Equality

You should not use the ROUND function as a general method to test for approximate equality. Two numbers that differ only in the least significant bit can round to different values if one number rounds down and the other number rounds up. Testing for approximate equality depends on how the numbers have been computed. If both numbers are computed to high relative precision, you could test for approximate equality by using the ABS and the MAX functions, as the following example shows.

```
if abs(x-y) <= 1e-12 * max( abs(x), abs(y) ) then
   ... more DS2 statements ...
```

Producing Expected Results

In general, `ROUND(expression, rounding-unit)` produces the result that you expect from decimal arithmetic if the result has no more than nine significant digits and any of the following conditions are true:

- The rounding unit is an integer.
- The rounding unit is a power of 10 greater than or equal to 1e-15. [1]
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```
data rounding(overwrite=yes);
   method run();
      d1 = round(1234.56789,100)      - 1200;
      d2 = round(1234.56789,10)       - 1230;
```

1. If the rounding unit is less than one, ROUND treats it as a power of 10 if the reciprocal of the rounding unit differs from a power of 10 in at most the three or four least significant bits.

```
      d3 = round(1234.56789,1)       - 1235;
      d4 = round(1234.56789,.1)      - 1234.6;
      d5 = round(1234.56789,.01)     - 1234.57;
      d6 = round(1234.56789,.001)    - 1234.568;
      d7 = round(1234.56789,.0001)   - 1234.5679;
      d8 = round(1234.56789,.00001)  - 1234.56789;
      d9 = round(1234.56789,.1111)   - 1234.5432;
      /* d10 has too many decimal places in the value for */
      /* rounding-unit.                                   */
      d10 = round(1234.56789,.11111) - 1234.54321;
   end;
enddata;
run;
```

The following output shows the results.

Output 7.19 *Results of Rounding Based on the Value of the Rounding Unit*

Obs	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10
1	0	0	0	0	0	0	0	0	0	0.012345

Operating Environment Information: The example above was executed in a z/OS environment. If you use other operating environments, the results is slightly different.

When the Rounding Unit Is the Reciprocal of an Integer

When the rounding unit is the reciprocal of an integer [1], the ROUND function computes the result by dividing by the integer. Therefore, you can safely compare the result from ROUND with the ratio of two integers, but not with a multiple of the rounding unit. Here is an example:

```
data rounding2(overwrite=yes);
   drop pi unit;
   method run();
      pi = arcos(-1);
      unit=1/7.;
      d1=round(pi,unit) - 22/7.;
      d2=round(pi, unit) - 22*unit;
   end;
enddata;
run;
```

The following output shows the results.

1. ROUND treats the rounding unit as a reciprocal of an integer if the reciprocal of the rounding unit differs from an integer in at most the three or four least significant bits.

Output 7.20 *Results of Rounding by the Reciprocal of an Integer*

The SAS System

Obs	d1	d2
1	0	1.3323E-15

Operating Environment Information: The example above was executed in an z/OS environment. If you use other operating environments, the results is slightly different.

Computing Results in Special Cases

The ROUND function computes the result by multiplying an integer by the rounding unit when all of the following conditions are true:

- The rounding unit is not an integer.
- The rounding unit is not a power of 10.
- The rounding unit is not the reciprocal of an integer.
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```
data _null_;
   method run();
      difference=round(1234.56789,.11111) - 11111*.11111;
      put difference=;
   end;
enddata;
run;
```

The following line is written to the SAS log:

```
difference=0
```

Operating Environment Information: The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

Computing Results When the Value Is Halfway between Multiples of the Rounding Unit

When the value to be rounded is approximately halfway between two multiples of the rounding unit, the ROUND function rounds up the absolute value and restores the original sign. For example:

```
data test(overwrite=yes);
   method run ();
      do i=8 to 17;
         value=0.5 - 10**(-i);
         round=round(value);
```

```
            output;
         end;
         do i=8 to 17;
            value=-0.5 + 10**(-i);
            round=round(value);
            output;
         end;
      end;
enddata;
run;
```

The following output shows the results.

Output 7.21 *Results of Rounding When Values Are Halfway between Multiples of the Rounding Unit*

The SAS System

Obs	I	VALUE	ROUND
1	8	0.50000	0
2	9	0.50000	0
3	10	0.50000	0
4	11	0.50000	0
5	12	0.50000	0
6	13	0.50000	1
7	14	0.50000	1
8	15	0.50000	1
9	16	0.50000	1
10	17	0.50000	1
11	8	-0.50000	0
12	9	-0.50000	0
13	10	-0.50000	0
14	11	-0.50000	0
15	12	-0.50000	0
16	13	-0.50000	-1
17	14	-0.50000	-1
18	15	-0.50000	-1
19	16	-0.50000	-1
20	17	-0.50000	-1

Operating Environment Information: The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

The approximation is relative to the size of the value to be rounded, and is computed in a manner that is shown in the following example. This example code will not always produce results exactly equivalent to the ROUND function.

```
data testfile(overwrite=yes);
```

```
   method run();
      do i = 1 to 17;
         value = 0.5 - 10**(-i);
         epsilon = min(1e-6, value * 1e-12);
         temp = value + .5 + epsilon;
         fraction = modz(temp, 1);
         round = temp - fraction;
         output;
      end;
   end;
enddata;
run;

proc print data=testfile noobs;
   format value 19.16;
run;
```

Comparisons

The ROUND function is the same as the ROUNDE function except that when the first argument is halfway between the two nearest multiples of the second argument, ROUNDE returns an even multiple. ROUND returns the multiple with the larger absolute value.

The ROUNDZ function returns a multiple of the rounding unit without trying to make the result match the result that is computed with decimal arithmetic.

Example

The following program compares the results that are returned by the ROUND function with the results that are returned by the ROUNDE function. The output was generated from the UNIX operating environment.

```
data results(overwrite=yes);
   dcl double x rounde round;
   method run();
      do x=0 to 4 by .25;
         Rounde=rounde(x);
         Round=round(x);
         output;
      end;
   end;
enddata;
run;

proc print data=results noobs;
run;
```

The following output shows the results.

Output 7.22 *Results That Are Returned by the ROUND and ROUNDE Functions*

The SAS System

Obs	X	ROUNDE	ROUND
1	0.00	0	0
2	0.25	0	0
3	0.50	0	1
4	0.75	1	1
5	1.00	1	1
6	1.25	1	1
7	1.50	2	2
8	1.75	2	2
9	2.00	2	2
10	2.25	2	2
11	2.50	2	3
12	2.75	3	3
13	3.00	3	3
14	3.25	3	3
15	3.50	4	4
16	3.75	4	4
17	4.00	4	4

See Also

Functions:

- “CEIL Function” on page 419
- “CEILZ Function” on page 421
- “FLOOR Function” on page 644
- “FLOORZ Function” on page 646
- “INT Function” on page 694
- “INTZ Function” on page 757

- "ROUNDE Function" on page 1044
- "ROUNDZ Function" on page 1047

ROUNDE Function

Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

ROUNDE(*expression* [, *rounding-unit*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value and that is to be rounded.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

rounding-unit
is a positive, numeric expression that specifies the rounding unit.

Default 1

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ROUNDE function rounds the first argument to the nearest multiple of the second argument.

Comparisons

The ROUNDE function is the same as the ROUND function except that when the first argument is halfway between the two nearest multiples of the second argument, ROUNDE returns an even multiple. ROUND returns the multiple with the larger absolute value.

Example

The following program compares the results that are returned by the ROUNDE function with the results that are returned by the ROUND function.

```
data results(overwrite=yes);
   dcl double x rounde round;
   method run();
      do x=0 to 4 by .25;
         rounde=rounde(x);
         round=round(x);
         output;
      end;
   end;
enddata;
run;
```

The following output shows the results.

Output 7.23 *Results That Are Returned by the ROUNDE and ROUND Functions*

The SAS System

Obs	X	ROUNDE	ROUND
1	0.00	0	0
2	0.25	0	0
3	0.50	0	1
4	0.75	1	1
5	1.00	1	1
6	1.25	1	1
7	1.50	2	2
8	1.75	2	2
9	2.00	2	2
10	2.25	2	2
11	2.50	2	3
12	2.75	3	3
13	3.00	3	3
14	3.25	3	3
15	3.50	4	4
16	3.75	4	4
17	4.00	4	4

See Also

Functions:

- “CEIL Function” on page 419
- “CEILZ Function” on page 421
- “FLOOR Function” on page 644
- “FLOORZ Function” on page 646
- “INT Function” on page 694
- “INTZ Function” on page 757

- "ROUND Function" on page 1035
- "ROUNDZ Function" on page 1047

ROUNDZ Function

Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

ROUNDZ(*expression* [, *rounding-unit*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

rounding-unit
specifies any valid expression that evaluates to a numeric expression and that specifies the rounding unit.

Default 1

Requirement Only positive values are valid.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The ROUNDZ function rounds the first argument to the nearest multiple of the second argument.

Comparisons

The ROUNDZ function is the same as the ROUND function with these exceptions:

- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument. ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples.
- When the rounding unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUNDZ does not fuzz the result. ROUND performs extra computations, called fuzzing, to try to make the result agree with decimal arithmetic.

Examples

Example 1: Comparing Results from the ROUNDZ and ROUND Functions

The following program compares the results that are returned by the ROUNDZ and the ROUND function.

```
data test (overwrite=yes);
   dcl double i value roundz round;
   method run();
      do i=10 to 17;
         Value=2.5 - 10**(-i);
         Roundz=roundz(value);
         Round=round(value);
         output;
      end;
      do i=16 to 12 by -1;
         value=2.5 + 10**(-i);
         roundz=roundz(value);
         round=round(value);
         output;
      end;
   end;
enddata;
run;

proc print data=test;
format value 19.16;
quit;
```

The following output shows the results.

Output 7.24 *Results That Are Returned by the ROUNDZ and ROUND Functions*

The SAS System

Obs	i	Value	Roundz	Round
1	10	2.4999999999000000	2	2
2	11	2.4999999999900000	2	2
3	12	2.4999999999990000	2	3
4	13	2.4999999999999000	2	3
5	14	2.4999999999999900	2	3
6	15	2.4999999999999900	2	3
7	16	2.5000000000000000	2	3
8	17	2.5000000000000000	2	3
9	16	2.5000000000000000	2	3
10	15	2.5000000000000000	3	3
11	14	2.5000000000000100	3	3
12	13	2.5000000000001000	3	3
13	12	2.5000000000010000	3	3

Example 2: Sample Output from the ROUNDZ Function

The following program illustrates the ROUNDZ function:

```
data _null_;
   dcl double x a b c d e;
   method run();
      x=223.456;
      a=roundz(x,1);
      b=roundz(x, .01);
      c=roundz(x, 100);
      d=roundz(x);
      e=roundz(x, .3);
      put a= b= c= d= e=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=223 b=223.46 c=200 d=223 e=223.5
```

See Also

Functions:

- “ROUND Function” on page 1035
- “ROUNDE Function” on page 1044

SAVING Function

Returns the future value of a periodic saving.

Category: Financial

Restriction: This function is not supported on the CAS server.

Returned data type: DOUBLE

Syntax

SAVING(*f*, *p*, *r*, *n*)

Arguments

f

is numeric, the future amount (at the end of *n* periods).

Range $f \geq 0$

Data type DOUBLE

p

is numeric, the fixed periodic payment.

Range $p \geq 0$

Data type DOUBLE

r

is numeric, the periodic interest rate expressed as a decimal.

Range $r \geq 0$

Data type DOUBLE

n
is an integer, the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

Details

The SAVING function returns the missing argument in the list of four arguments from a periodic saving. The arguments are related by the following equation:

$$f = \frac{p(1+r)\left((1+r)^n - 1\right)}{r}$$

One missing argument must be provided. It is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Example

A savings account pays a 5% nominal annual interest rate, compounded monthly. For a monthly deposit of $100, the number of payments that are needed to accumulate at least $12,000, can be expressed as follows:

```
data test (overwrite=yes);
   dcl double a;
   method run();
      a=saving(12000, 100, .05/12, .);
      put  'a= ' a;
   end;
enddata;
run;
```

The value that is returned is 97.18 months. The fourth argument is set to missing, which indicates that the number of payments is to be calculated. The 5% nominal annual rate is converted to a monthly rate of 0.05/12. The rate is the fractional (not the percentage) interest rate per compounding period.

See Also

Functions:

- “SAVINGS Function” on page 1052

SAVINGS Function

Returns the balance of a periodic savings by using variable interest rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

SAVINGS(*base-date*, *initial-deposit-date*, *deposit-amount*, *deposit-number*, *deposit-interval*, *compounding-interval*, *date-1*, *rate-1*[, ...*date-n*, *rate-n*])

Arguments

base-date
specifies the value that is returned is the balance of the savings at the base date.

Requirement *Base-date* is a SAS date.

Data type DOUBLE

initial-deposit-date
specifies the date of the first deposit. Subsequent deposits are at the beginning of subsequent deposit intervals.

Requirement *Initial-deposit-date* is a SAS date.

Data type DOUBLE

deposit-amount
specifies the value of each deposit. All deposits are assumed constant.

Data type DOUBLE

deposit-number
specifies the number of deposits.

Data type DOUBLE

deposit-interval
specifies the frequency at which deposits are made.

Requirement *Deposit-interval* is a SAS interval.

Data type CHAR

compounding-interval
specifies the compounding interval.

Requirement *Compounding-interval* is a SAS interval.

Data type CHAR

date
specifies the time at which *rate* takes effect. Each date is paired with a rate.

Requirement *Date* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate as numeric percentage that starts on *date*. Each rate is paired with a date.

Data type DOUBLE

Details

The following details apply to the SAVINGS function:

- The values for rates must be between –99 and 120.
- *Deposit-interval* cannot be 'CONTINUOUS'.
- The list of date-rate pairs does not need to be in chronological order.
- When multiple rate changes occur on a single date, the SAVINGS function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the *initial-deposit-date* and the *base-date*.

Example

- If you deposit $300 monthly for two years into an account that compounds quarterly at an annual rate of 4%, the balance of the account after five years can be expressed as follows:

```
data _null_;
   dcl double bd idd d amount_base1;
   method run();
      bd=to_double(date'2005-01-01');
      idd=to_double(date'2000-01-01');
      d=to_double(date'2000-01-01');
      amount_base1=savings(bd, idd, 300, 24, 'month', 'qtr', d, 4.00);
```

```
      put amount_base1;
   end;
enddata;
run;
```

The following line is written to the SAS log.

```
8458.79415896917
```

- If the interest rate increases by a quarter-point each year, then the balance of the account could be expressed as follows:

```
 data _null_;
   dcl double bd idd d1 d2 d3 d4 d5 amount_base2;
   method run();
      bd= to_double(date'2005-01-01');
      idd= to_double(date'2000-01-01');
      d1= to_double(date'2000-01-01');
      d2= to_double(date'2001-01-01');
      d3= to_double(date'2002-01-01');
      d4= to_double(date'2003-01-01');
      d4= to_double(date'2004-01-01');
      amount_base2 = savings(bd, idd, 300, 24, 'month', 'qtr', d1, 4.00, d2,
         4.25, d3, 4.50, d4, 4.75, d5, 5.0);
      put amount_base2;
   end;
enddata;
run;
```

The following line is written to the SAS log.

```
8623.09024586998
```

- To determine the balance after one year of deposits, the following program sets amount_base3 to the desired balance:

```
data _null_;
   dcl double bd idd d amount_base3;
   method run();
      bd= to_double(date'2001-01-01');
      idd= to_double(date'2000-01-01');
      d= to_double(date'2000-01-01');
      amount_base3 = savings(bd, idd, 300, 24, 'month', 'qtr', d, 4);
      put amount_base3;
   end;
enddata;
run;
```

The following line is written to the SAS log.

```
3978.69037121739
```

The SAVINGS function ignores deposits after the base date, so the deposits after the reference date do not affect the value that is returned.

See Also

Functions:

- "SAVING Function" on page 1050

SCAN Function

Returns the *n*th word from a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

SCAN(*expression*, *n* [, *delimiters*[, *modifier*]])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

n
is a nonzero numeric expression that specifies the number of the word in the character expression that you want SCAN to select. The following rules apply:

- If *n* is positive, SCAN counts words from left to right in the character string.
- If *n* is negative, SCAN counts words from right to left in the character string.
- If *n* is greater than the number of words in *expression*, SCAN returns a blank value.

delimiters
specifies any valid expression that evaluates or can be coerced to a character string and that SCAN uses as word separators in the expression.

Default:

Requirement: If *delimiter* is a constant, enclose *delimiter* in single quotation marks.

Interactions: ASCII default delimiters are: blank ! $ % & ( ) * + , – . / ; < |. In environments without the ^ character, SCAN uses the ~ character instead.

EBCDIC default delimiters are: blank ! $ % & ( ) * + , – . / ; < ¬ | ¢.

Specifying a modifier can change the characters in *delimiter*. For example, if you specify the K modifier in the *modifier* argument, all characters that are not in *delimiter* are used as delimiters.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: You can add more characters to *delimiter* by using other modifiers.

See: "Using Default Delimiters in ASCII and EBCDIC Environments" on page 1058

"DS2 Expressions" in *SAS DS2 Programmer's Guide*

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the SCAN function. Blanks are ignored. Use the following characters as modifiers:

a or A — adds alphabetic characters to the list of characters.

b or B — scans backward from right to left instead of from left to right, regardless of the sign of the *count* argument.

c or C — adds control characters to the list of characters.

d or D — adds digits to the list of characters.

f or F — adds an underscore and English letters to the list of characters.

g or G — adds graphic characters to the list of characters. Graphic characters are characters that, when printed, produce an image on paper.

h or H — adds a horizontal tab to the list of characters.

i or I — ignores the case of the characters.

k or K — causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.

l or L — adds lowercase letters to the list of characters.

m or M — specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.

n or N — adds digits, an underscore, and English letters to the list of characters.

o or O processes the *charlist* and *modifier* arguments only once, rather than every time the SCAN function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure can make SCAN run faster when you call it in a loop where the *character-list* and *modifier* arguments do not change. The O modifier applies separately to each instance of the SCAN function in your SAS code, and does *not* cause all instances of the SCAN function to use the same delimiters and modifiers.

p or P adds punctuation marks to the list of characters.

q or Q ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.

r or R removes leading and trailing blanks from the word that SCAN returns. If you specify the Q and R modifiers, the SCAN function first removes leading and trailing blanks from the word. Then, if the word begins with a quotation mark, SCAN also removes one layer of quotation marks from the word.

s or S adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T trims trailing blanks from the *string* and *charlist* arguments. If you want to remove trailing blanks from only one character argument instead of both character arguments, use the TRIM function instead of the SCAN function with the T modifier.

u or U adds uppercase letters to the list of characters.

w or W adds printable (writable) characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Restriction This argument is supported only in SAS Viya and on the CAS server.

Tip If the *modifier* argument is a character constant, enclose the argument in quotation marks. Specify multiple modifiers in a single set of quotation marks. A *modifier* argument can also be expressed as a character variable or expression.

Details

Definitions of “Delimiter” and “Word”

A *delimiter* is any of several characters that are used to separate words. You can specify the delimiters in the *delimiter* and *modifier* arguments.

If you specify the Q modifier, delimiters inside substrings that are enclosed in quotation marks are ignored.

In the SCAN function, “word” refers to a substring that has all of these characteristics:

- It is bounded on the left by a delimiter or the beginning of the string.
- It is bounded on the right by a delimiter or the end of the string.
- It contains no delimiters.

A word can have a length of zero if there are delimiters at the beginning or end of the string, or if the string contains two or more consecutive delimiters. However, the SCAN function ignores words that have a length of zero unless you specify the M modifier.

Note: The definition of "word" is the same in the SCAN and COUNTW functions.

Using Default Delimiters in ASCII and EBCDIC Environments

If you use the SCAN function with only two arguments, then the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.

- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

If you use the *modifier* argument without specifying any characters as delimiters, then the only delimiters that are used are delimiters that are defined by the *modifier* argument. In this case, the lists of default delimiters for ASCII and EBCDIC environments are not used. In other words, modifiers add to the list of delimiters that are explicitly specified by the *delimiter* argument. Modifiers do not add to the list of default modifiers.

The Length of the Result

Leading delimiters before the first word in the expression do not affect SCAN. If there are two or more contiguous delimiters, SCAN treats them as one.

In DS2, if the SCAN function returns a value to a variable that has not yet been given a length, and then that variable is given the length of the first argument. If you need the SCAN function to assign to a variable a value that is different from the length of the first argument, use a DECLARE statement for that variable before the statement that uses the SCAN function.

The minimum length of the word that is returned by the SCAN function depends on whether the M modifier is specified. See "Using the SCAN Function with the M Modifier" on page 1059. See also "Using the SCAN Function without the M Modifier" on page 1059.

Using the SCAN Function with the M Modifier

If you specify the M modifier, the number of words in a string is defined as one plus the number of delimiters in the string. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you specify the M modifier, the SCAN function returns a word with a length of zero if one of these conditions is true:

- The string begins with a delimiter and you request the first word.
- The string ends with a delimiter and you request the last word.
- The string contains two consecutive delimiters and you request the word that is between the two delimiters.

Using the SCAN Function without the M Modifier

If you do not specify the M modifier, the number of words in a string is defined as the number of maximal substrings of consecutive non-delimiters. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you do not specify the M modifier, the SCAN function acts in these ways:

- It ignores delimiters at the beginning or end of the string.
- It treats two or more consecutive delimiters as if they were a single delimiter.

If the string contains no characters other than delimiters, or if you specify a count that is greater in absolute value than the number of words in the string, then the SCAN function returns one of the following items:

- a single blank when you call the SCAN function from a DATA step
- a string with a length of zero when you call the SCAN function from the macro processor

Using Null Arguments

The SCAN function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Processing SBCS and DBCS Data

The SCAN function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria:

- If *expression* is not declared as VARCHAR and you are processing single-byte data, then SCAN processes SBCS.
- If *string* is declared as VARCHAR and you are processing multibyte data, then SCAN processes DBCS.

Examples

Example 1: Finding the First and Last Words in a String

This example scans a string for the first and last words:

- A negative count instructs the SCAN function to scan from right to left.
- Leading and trailing delimiters are ignored because the M modifier is not used.
- In the last observation, all characters in the string are delimiters.

```
data test (overwrite=yes);
   dcl varchar(8) first_word last_word;
   dcl varchar(29) string1 string2 string3 string4;
   dcl double i;
   method run();
      string1='Jack and Jill';
      first_word=scan(string1, 1);
      last_word=scan(string1, -1);
      put first_word= last_word=;
      string2='& Bob & Carol & Ted & Alice &';
      first_word=scan(string2, 1);
      last_word=scan(string2, -1);
      put first_word= last_word=;
      string3='Leonardo';
      first_word=scan(string3, 1);
      last_word=scan(string3, -1);
      put first_word= last_word=;
      string4='! $ % & ( ) * , - . / ;';
      first_word=scan(string4, 1);
      last_word=scan(string4, -1);
      put first_word= last_word=;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
first_word=Jack last_word=Jill
first_word=Bob last_word=Alice
first_word=Leonardo last_word=Leonardo
first_word=  last_word=
```

Example 2: Finding All Words in a String without Using the M Modifier

This example scans a string from left to right until the word that is returned is blank. Because the M modifier is not used, the SCAN function does not return any words that have a length of zero. Because blanks are included among the default delimiters, the SCAN function returns a blank word only when the count exceeds the number of words in the string. Therefore, the loop can be stopped when SCAN returns a blank word.

```
proc ds2;
data test(drop=(string) overwrite=yes);
   dcl varchar(47) string word;
```

```
   dcl double count;
   method run();
      string=' The quick brown fox jumps over the lazy dog.   ';
      do until(word=' ');
         count+1;
         word=scan(string, count);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=test;
run;
```

Obs	word	count
1	The	1
2	quick	2
3	brown	3
4	fox	4
5	jumps	5
6	over	6
7	the	7
8	lazy	8
9	dog	9
10		10

Example 3: Finding All Words in a String by Using the M and O Modifiers

This example shows the results of using the M modifier with a comma as a delimiter. With the M modifier, leading, trailing, and multiple consecutive delimiters cause the SCAN function to return words that have a length of zero. Therefore, do not end the loop by testing for a blank word. Instead, use the COUNTW function with the same modifiers and delimiters to count the words in the string.

The O modifier is used for efficiency because the delimiters and modifiers are the same in every call to the SCAN and COUNTW functions.

```
proc ds2;
data comma(keep=(count word) overwrite=yes);
   dcl varchar(45) string;
   dcl varchar(20) delim modif word;
   dcl double nwords count;
   method run();
      string=',leading, trailing,and multiple,,delimeters,,';
      delim=',';
      modif='mo';
      nwords=countw(string, delim, modif);
```

```
      put nwords=;
      do count=1 to nwords;
         word=scan(string, count, delim, modif);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=comma;
run;
```

Obs	word	count
1		1
2	leading	2
3	trailing	3
4	and multiple	4
5		5
6	delimeters	6
7		7
8		8

Example 4: Using Comma-Separated Values, Substrings in Quotation Marks, and the O and R Modifiers

This example uses the SCAN function with the O modifier and a comma as a delimiter, both with and without the R modifier.

The O modifier is used for efficiency because in each call of the SCAN or COUNTW function, the delimiters and modifiers do not change. The O modifier applies separately to each of the two instances of the SCAN function:

- The first instance of the SCAN function uses the same delimiters and modifiers every time SCAN is called.
- The second instance of the SCAN function uses the same delimiters and modifiers every time SCAN is called.
- The first instance of the SCAN function does not use the same modifiers as the second instance, but this fact has no bearing on the use of the O modifier.

```
proc ds2;
data test(keep=(word word_r count) overwrite=yes);
   dcl varchar(40) string;
   dcl varchar(20) word word_r;
   dcl varchar delim modif;
   dcl double count nwords;
   method run();
      string='He said, "She said, ""No!""", not "Yes!"';
      delim=',';
      modif='oq';
```

```
      nwords=countw(string, delim, modif);
      do count=1 to nwords;
         word=scan(string, count, delim, modif);
         word_r=scan(string, count, delim, modif||'r');
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=test;
run;
```

Obs	word	word_r	count
1	He said	He said	1
2	"She said, ""No!"""	She said, "No!"	2
3	not "Yes!"	not "Yes!"	3

Example 5: Finding Substrings of Digits by Using the D and K Modifiers

This example finds substrings of digits. The character-list argument is null. Consequently, the list of characters is initially empty. The D modifier adds digits to the list of characters. The K modifier treats all characters that are not in the list as delimiters. Therefore, all characters except digits are delimiters.

```
proc ds2;
data test(keep=(count digits) overwrite=yes);
   dcl char(25) string digits;
   dcl double count;
   method run();
      string='Call (800) 555-1234 now!';
      do until(digits=' ');
         count+1;
         digits=scan(string, count, '', 'dko');
         output;
      end;

   end;
enddata;
run;
quit;

proc print data=test;
run;
```

Obs	digits	count
1	800	1
2	555	2
3	1234	3
4		4

Example 6: Finding All Words in a String by Using the M and O Modifiers

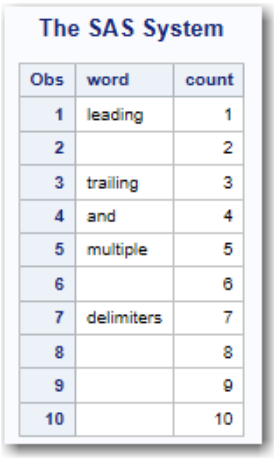
This example shows the results of using the M modifier with a comma as a delimiter. With the M modifier, leading, trailing, and multiple consecutive delimiters cause the SCAN function to return words that have a length of zero. Therefore, do not end the loop by testing for a blank word. Instead, use the COUNTW function with the same modifiers and delimiters to count the words in the string.

The O modifier is used for efficiency because the delimiters and modifiers are the same in every call to the SCAN and COUNTW functions.

Note: This example is valid only in SAS Viya or on the CAS server.

```
proc ds2 sessref=casauto;
data comma (keep= (count word) overwrite=yes);
   dcl char(45) string;
   dcl char(12) delim modif word;
   dcl double nwords count;
   method run();
      string=',leading, trailing,and multiple,,delimiters,,';
      delim=',';
      modif='mo';
      nwords=countw(string, delim, modif);
      do count=1 to nwords;
         word=scan(string, count, delim, modif);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=work.comma;
run;
quit;
```



The SAS System

Obs	word	count
1	leading	1
2		2
3	trailing	3
4	and	4
5	multiple	5
6		6
7	delimiters	7
8		8
9		9
10		10

SDF Function

Returns a survival function.

Categories: CAS
Probability

See: "CDF Function" on page 370

Syntax

SDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Arguments

distribution
is a character string that identifies the distribution.

Note: The arguments for each of the SDF distribution functions are identical to those of the corresponding CDF distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	`'BERNOULLI'`
Beta	`'BETA'`
Binomial	`'BINOMIAL'`
Cauchy	`'CAUCHY'`

Distribution	Argument
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL'|'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'
Wald (inverse Gaussian)	'WALD'|'IGAUSS'
Weibull	'WEIBULL'

Note Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Data type DOUBLE

Details

The SDF function computes the survival function (upper tail) from various continuous and discrete distributions. For more information, see the individual distributions listed in the table above.

The SDF function for the Conway-Maxwell-Poisson distribution has the following form:

SDF('CONMAXPOI',*y*,*λ*,*v*)

y
is a nonnegative, whole number that represents counts data.

λ
is similar to the mean, as in the Poisson distribution.

v
is a dispersion parameter.

The SDF function returns the probability that the counts value is greater than *y*.

For more information, see "Conway-Maxwell-Poisson" distribution in the PDF function on page 898.

Example

The following program illustrates the SDF function:

```
proc ds2;
data _null;
   dcl double y;
   method run();
      y=sdf('BERN', 0, .25);
      put 'Bern dist:      ' y;
      y=sdf('BETA', 0.2, 3,4);
      put 'Beta dist:      ' y;
      y=sdf('BINOM', 4, .5, 10);
      put 'Binom dist:     ' y;
      y=sdf('CAUCHY', 2);
      put 'Cauchy dist:    ' y;
```

```
      y=sdf('CHISQ', 11.264, 11);
      put 'Chisq dist:     ' y;
      y=sdf('CONMAXPOI', 12, 2.3, .4);
      put 'Conmaxpoi dist: ' y;
      y=sdf('EXPO', 1);
      put 'Expo dist:      ' y;
      y=sdf('F', 3.32, 2,3);
      put 'F dist:         ' y;
      y=sdf('GAMMA', 1, 3);
      put 'Gamma dist:     ' y;
      y=sdf('GENPOISSON', .9, 1, .7);
      put 'Genpoisson dist:' y;
      y=sdf('GEOMETRIC', .5, .3);
      put 'Geometric dist: ' y;
      y=sdf('HYPER', 2, 200, 50, 10);
      put 'Hyper dist:     ' y;
      y=sdf('LAPLACE',1);
      put 'Laplace dist:   ' y;
      y=sdf('LOGISTIC', 1);
      put 'Logistic dist:  ' y;
      y=sdf('LOGNORMAL', 1);
      put 'LogNormal dist: ' y;
      y=sdf('NEGB', 1, .5,2);
      put 'NegB dist:      ' y;
      y=sdf('NORMAL', 1.96);
      put 'Normal dist:    ' y;
      y=sdf('NORMALMIX', 2.3, 3, .33, .33, .34, .5, 1.5, 2.5, .79,
1.6, 4.3);
      put 'Normal mix dist:' y;
      y=sdf('PARETO', 1, 1);
      put 'Pareto dist:    ' y;
      y=sdf('POISSON', 2, 1);
      put 'Poisson dist:   ' y;
      y=sdf('T', .9, 5);
      put 'T dist:         ' y;
      y=sdf('TWEEDIE', .8, 5);
      put 'Tweedie dist:   ' y;
      y=sdf('UNIFORM', 0.25);
      put 'Uniform dist:   ' y;
      y=sdf('WALD', 1, 2);
      put 'Wald dist:      ' y;
      y=sdf('WEIBULL', 1, 2);
      put 'Weibull dist:   ' y;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
Bern dist:       0.25
Beta dist:       0.90112
Binom dist:      0.62304687499999
Cauchy dist:     0.14758361765043
Chisq dist:      0.42141867068269
Conmaxpoi dist:  0.19705138765343
Expo dist:       0.36787944117144
F dist:          0.17360663977568
Gamma dist:      0.9196986029286
Genpoisson dist: 0.63212055882855
Geometric dist:  0.7
Hyper dist:      0.47632659187826
Laplace dist:    0.18393972058572
Logistic dist:   0.26894142136999
LogNormal dist:  0.5
NegB dist:       0.5
Normal dist:     0.02499789514822
Normal mix dist: 0.28186912768685
Pareto dist:     1
Poisson dist:    0.08030139707139
T dist:          0.20468560017231
Tweedie dist:    0.40823708356802
Uniform dist:    0.75
Wald dist:       0.37230216184474
Weibull dist:    0.36787944117144
```

See Also

Functions:

- “CDF Function” on page 370
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Function” on page 886

SEC Function

Returns the secant.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

SEC(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value that expressed in radians.

Restriction *expression* cannot be an odd multiple of PI/2.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Comparisons

The SEC function is related to the COS function:

```
sec(x)=1/cos(x)
```

Example

```
data _null_;
  dcl double x y z;
  method run();
     x=sec(0.5);
     y=sec(0);
     z=sec(3.14159/3);
     put x= y= z=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=1.13949392732454 y=1 z=1.99999693590391
```

See Also

Functions:

- "COS Function" on page 475
- "COT Function" on page 477
- "CSC Function" on page 489
- "SIN Function" on page 1080
- "TAN Function" on page 1121

SECOND Function

Returns the second from a SAS time or datetime value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

SECOND(*time* | *datetime*)

Arguments

time
specifies any valid expression that represents a SAS time value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The SECOND function produces a numeric value that represents a specific second of the minute. The result can be any number that is >= 0 and < 60.

Examples

Example 1: Basic SECOND Function Usage

The following program illustrates the SECOND function:

```
data test(overwrite=yes);
   dcl double a b c d e f;
   method run();
```

```
      a=hms(3,19,24);
      b=second(a);
      put b=;
      c=hms(6,25,65);
      d=second(c);
      put d=;
      e=hms(3,19,60);
      f=second(e);
      put f=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
b=24
d=5
f=0
```

Example 2: Displaying the Second of the Minute That Is Associated with the Current Date and Time

This program specifies the second of the minute that is associated with datetime. The first three digits for the milliseconds, 876, are relevant.

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=second(datetime());
      put x=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=18.8769989013671
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "HOUR Function" on page 682
- "MINUTE Function" on page 824

SHA256 Function

Returns the result of the message digest of a specified string.

Categories: CAS
Character

Returned data type: BINARY

Syntax

SHA256('*string*')

Arguments

string
specifies a character constant, variable, or expression.

Data type: BINARY, CHAR

Tips: Enclose a literal string of characters in single quotation marks.

For scalar character variable arguments, the initial character set encoding that is specified in the DECLARE statement is used to transcode the variable before it is passed to the SHA256 function. For binary arguments, the binary value is treated as a character string and the session encoding is used to transcode the value before it is passed to the SHA256 function.

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The SHA256 function converts a message string, based on the SHA256 algorithm, to a 256-bit hash value.

The SHA256 function does not format its own output. Use the $BINARY*w*. or $HEX*w*. format to view readable results.

You can use the SHA256 function to track changes in your data sets. The SHA256 function can generate a digest of a set of column values in a table record. This digest could be treated as the signature of the record and be used to track changes that are made to the record. If the digest from the new record matches the existing digest of a table record, then the two records are the same. If the digest is different,

then a column value in the record has changed. The new changed record could then be added to the table along with a new surrogate key because the record represents a change to an existing keyed value.

The SHA256 function can be useful when you are developing shell scripts for software installation, file comparison, and detection of file corruption and tampering.

Example: Generating Results with the SHA256 Function

This program generates results that are returned by the SHA256 function.

```
data _null_;
   dcl binary(64) y z;
   method init();
      y=sha256('abc');
      z=sha256('access method');
      put y=$HEX64.;
      put z=$HEX64.;
    end;
enddata;
run;
```

For ASCII systems, SAS writes the following output to the log:

```
y=BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
z=F2758E91725621F59F2F80D15DE8824560EDC471EBE40A83BA6D1259B1605915
```

For EBCDIC systems, SAS writes the following output to the log:

```
y=B58EA6D31995A4D8CE092EB718DDFA58B6CEF2288B41FAF1DCD52FF3D6D8FA01
z=D7EE088DAF6B029BADCC2DD01984867F0A3C342D60719DA7B478721E3E778F63
```

SHA256HEX Function

Returns the result of the message digest of a specified string and converts the string to hexadecimal representation.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR
Notes:	The SHA256HEX function verifies the data integrity and authentication of a message. UTF-8 text is recommended for the SHA256HEX function arguments to ensure consistency across encodings.

Syntax

SHA256HEX('*string*', *string_indicator*);

Arguments

string
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

string_indicator
indicates whether the argument *string* is regular characters or hexadecimal representation characters.

0 indicates that the expression in the argument *string* is regular characters.

1 indicates that the expression in the argument *string* is hexadecimal representation characters.

Note: There must be an even number of hexadecimal representation characters, and they must all be between 0–9, a–f, A–F. Blanks in the hexadecimal representation string are ignored.

Data type DOUBLE

Details

The Basics

The SHA256HEX function converts a string, based on the SHA256 algorithm, to a 256-bit hash value. Then, the function converts the data to a hexadecimal representation format.

z/OS Specifics: In the z/OS operating environment, because the SHA256HEX function might be operating on EBCDIC data, the message digest is different from the ASCII equivalent. For example, SHA256HEX('ABC') on an EBCDIC system means that SHA256HEX receives the bytes 'C1C2C3'x, and the digest is 5202BF40821662BF1AD7D9C9B558056775D9D6BF8AA1C00492BCA8556B02772F. On an ASCII system, 'ABC' is '414243'x, and the digest is B5D4045C3F466FA91FE2CC6ABE79232A1A57CDF104F7A26E716E0A1E2789DF78.

Using the SHA256HEX Function

You can use the SHA256HEX function to track changes in your data sets. The SHA256HEX function can generate a digest of a set of column values in a table record. This digest could be treated as the signature of the record and be used to track changes that are made to the record. If the digest from the new record matches the existing digest of a table record, then the two records are the same. If the digest is different, then a column value in the record has changed. The new changed record could then be added to the table along with a new surrogate key because the record represents a change to an existing keyed value.

The SHA256HEX function can be useful when you are developing shell scripts or Perl programs for software installation, file comparison, and detection of file corruption and tampering.

Comparisons

The SHA256 function does not format its own output, so you must use the $BINARYw. or $HEXw. formats to view readable results. The SHA256HEX function formats its output, so you do not have to use the $BINARY or $HEX formats.

Example: Generating Results with the SHA256HEX Function

This example generates results that are returned by the SHA256HEX function.

```
data _null_;
   dcl char(64) y z;
   method run();
      y=sha256hex('abc');
      z=sha256hex('access method');
      put y=;
      put z=;
   end;
enddata;
run;
```

For ASCII systems, SAS writes the following output to the log:

```
y=BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
z=F2758E91725621F59F2F80D15DE8824560EDC471EBE40A83BA6D1259B1605915
```

For EBCDIC systems, SAS writes the following output to the log:

```
y=B58EA6D31995A4D8CE092EB718DDFA58B6CEF2288B41FAF1DCD52FF3D6D8FA01
z=D7EE088DAF6B029BADCC2DD01984867F0A3C342D60719DA7B478721E3E778F63
```

See Also

Functions:

- "SHA256HMACHEX Function" on page 1077

SHA256HMACHEX Function

Returns the result of the message digest of a specified string by using the Hash-based Message Authentication (HMAC) algorithm.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Notes: The SHA256HMACHEX function verifies the data integrity and authentication of a message.

See the following article for more information about the Hash-based message authentication code (HMAC).

UTF-8 text is recommended for the SHA256HMACHEX function arguments to ensure consistency across encodings.

Syntax

SHA256HMACHEX('*key*', '*message*' [*string-indicator*]);

Arguments

key
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

message
specifies a secret key padded to the right with extra zeros to the input block size of the hash function.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

string_indicator
indicates whether the key and message are provided in hexadecimal representation.

0 the arguments *key* and *message* are not represented in hexadecimal representation.

1 the argument *message* is represented in hexadecimal representation.

2 the argument *key* is represented in hexadecimal representation.

3 the arguments *key* and *message* are represented in hexadecimal representation.

Note: This argument is useful when the SHA256HMACHEX function is being called repeatedly and the result of a previous call is used as the key in a subsequent call. The following code demonstrates this functionality:

```
length digest $64;
digest = sha256hmachex('mykey', 'mymessage', 0);
digest = sha256hmachex(digest, 'my new message', 2);
```

Data type DOUBLE

Details

The SHA256HMACHEX function converts a string, based on the SHA256 algorithm, to a 256-bit hash value.

See the following article for more information about the Hash-based message authentication code (HMAC).

Example: Generating Results with the SHA256HMACHEX Function

This example generates results that are returned by the SHA256HMACHEX function.

```
data _null_;
   dcl char(200) digest;
   method run();
      digest = SHA256HMACHEX('key',
      'The quick brown fox jumps over the lazy dog', 0);
      if digest=

upcase('f7bc83f430538424b13298e6aa6fb143ef4d59a14946175997479dbc2d1a3cd
8')
         then
            put 'matched';
         else
            put 'not matched';
   end;
enddata;
```

```
run;
```

```
matched
```

SIGN Function

Returns a number that indicates the sign of a numeric value expression.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

SIGN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type: All numeric types

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The SIGN function returns the following values:

-1
if *expression* < 0

0
if *expression* = 0

1
if *expression* > 0.

Example

The following program illustrates the SIGN function:

```
data test(overwrite=yes);
```

```
   dcl double x y z;
   method run();
      x=sign(-5);
      y=sign(5);
      z=sign(0);
      put x= y= z=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=-1 y=1 z=0
```

SIN Function

Returns the trigonometric sine.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

SIN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the SIN function:

```
data test(overwrite=yes);
   dcl double x y z;
   method run();
      x=sin(0.5);
      y=sin(0);
      z=sin(3.14159/4);
```

```
   put x= y= z=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=0.4794255386042 y=0 z=0.70710631209355
```

SINH Function

Returns the hyperbolic sine.

Categories: CAS
Hyperbolic

Returned data type: DOUBLE

Syntax

SINH(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The SINH function returns the hyperbolic sine of the argument, which is given by the following equation.

$$e^{argument} - e^{-argument}/2$$

Example

The following program illustrates the SINH function:

```
data test(overwrite=yes);
```

```
      dcl double x y z;
      method run();
         x=sinh(0);
         y=sinh(1);
         z=sinh(-1);
         put x= y= z=;
      end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
x=0 y=1.1752011936438 z=-1.1752011936438
```

SKEWNESS Function

Returns the skewness.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

SKEWNESS(*expression-1, expression-2, expression-3* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least three non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If all non-null or nonmissing arguments have equal values, the skewness is mathematically undefined and the SKEWNESS function returns a null or missing value.

Example

The following program illustrates the SKEWNESS function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3 x4;
   method run();
      x1=skewness(0, 1, 1);
      x2=skewness(2, 4, 6, 3, 1);
      x3=skewness(2, 0, 0);
      x4=skewness(of x1-x3);
      put x1=;
      put x2=;
      put x3=;
      put x4=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=-1.73205080756887
x2=0.59012865638436
x3=1.73205080756887
x4=-0.9530977136649
```

SLEEP Function

For a specified period of time, suspends the execution of a program that invokes this function.

Categories: CAS
Special

Returned data type: DOUBLE

Syntax

SLEEP(*number-of-time-units* [, *time-unit*])

Arguments

number-of-time-units
specifies any valid expression that evaluates to a numeric value and that specifies the number of units of time for which you want to suspend execution of a program.

Range $n \geq 0$

Restriction Negative values are invalid.

Data type DOUBLE

Tip If you use a fraction for the *n* argument, the *unit* argument is required if you want to suspend execution for a fraction of a second. For example, `SLEEP(.25);` does not suspend execution. `SLEEP(1.25);` suspends execution for 1 second. `SLEEP(1.25, 1);` suspends execution for 1.25 seconds. `SLEEP(.25,1)` suspends execution for .25 seconds.

time-unit

specifies that the unit of time in seconds is a multiple of 10, and that is applied to *number-of-time-units*. For example, 1 corresponds to 1 second, and .001 to a millisecond, and 5 corresponds to 5 seconds.

Default 1 in a Windows PC environment, .001 in other environments

Data type DOUBLE

Example This code is in Windows:

```
x=sleep(10)
```

This code is in UNIX:

```
x=sleep(10000);
```

Both code examples produce the same result. If you want to use the SLEEP function in a portable manner between Windows and UNIX, this code suspends execution of the program for 10 seconds:

```
x=sleep(10,1);
```

Details

The SLEEP function suspends the execution of a program that invokes this function for a period of time that you specify.

The SLEEP function suspends the execution of a program and returns the value of the first argument.

If you are using a GUI environment such as SAS Studio, then the SLEEP function uses the default *time-unit* value. A window appears that indicates how long SAS is going to sleep.

When you submit a program that calls the SLEEP function, the SLEEP window appears, telling you when SAS is going to wake up. Your SAS session remains inactive until the sleep period is over. To cancel the call to the SLEEP function, use the CTRL+BREAK attention sequence.

You should use a null data program to call the SLEEP function; follow this data program with the rest of the SAS program. Using theSLEEP function in this manner enables you to use the CTRL+BREAK attention sequence to interrupt the SLEEP function and to continue with the execution of the rest of your SAS program.

Examples

Example 1: Suspending Execution for a Specified Period of Time

The following program delays the execution for 20 seconds:

```
data payroll;
   ...DS2 statements...
   time_slept=sleep(20,1);
   ...more DS2 statements...
enddata;
```

Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following program tells SAS to suspend the execution until June 15, 2011, at midnight. DS2 calculates the length of the suspension based on the target date and the date and time that the code begins to execute.

```
data budget;
   ...DS2 statements...;
   sleeptime=datetime() - dhms(mdy(06,15,2011),00,00,00);
   time_calc=sleep(sleeptime,1);
   ...more DS2 statements...;
enddata;
```

SMALLEST Function

Returns the *k*th smallest non-null or nonmissing value.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

SMALLEST(*k*, *expression* [, ...*expression*])

Arguments

k

specifies any valid expression that evaluates to a numeric value to return.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression
specifies any valid expression that evaluates to a numeric value to be processed.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

If *k* is null or missing, less than zero, or greater than the number of values, the result is a null or missing value.

Comparisons

The SMALLEST function differs from the ORDINAL function in that SMALLEST ignores null and missing values, but ORDINAL counts null and missing values.

Example

This example compares the values that are returned by the SMALLEST function with values that are returned by the ORDINAL function.

```
proc ds2;
data comparison;
   dcl double smallest_num having label 'SMALLEST Function';
   dcl double ordinal_num having label 'ORDINAL Function';
   dcl double k;
   method run();
      do k = 1 to 4;
         smallest_num = smallest(k, 456, 789, .Q, 123);
         ordinal_num  = ordinal (k, 456, 789, .Q, 123);
         output;
      end;
   end;
enddata;
run;
quit;
proc print data=comparison label;
   var k smallest_num ordinal_num;
   title 'Results From the SMALLEST and the ORDINAL Functions';
run;
```

Output 7.25 *Comparison of Values: The SMALLEST and the ORDINAL Functions*

Results From the SMALLEST and the ORDINAL Functions

Obs	K	SMALLEST Function	ORDINAL Function
1	1	123	Q
2	2	456	123
3	3	789	456
4	4	.	789

See Also

Functions:

- "LARGEST Function" on page 774
- "ORDINAL Function" on page 883
- "PCTL Function" on page 884

SQLEXEC Function

Executes a FedSQL statement to create, delete, or update a table or to insert rows into a table.

Category: Special

Restriction: This function is not supported on the CAS server.

Syntax

SQLEXEC('*sql-text*')

Arguments

'sql-text'
is a valid FedSQL statement that inserts into, updates, creates, or deletes rows from a table.

CAUTION

Only FedSQL statements can be used with the SQLEXEC function. DBMS-specific SQL cannot be used. For more information, see *SAS FedSQL Language Reference*.

Requirement	The FedSQL statement must be enclosed in single quotation marks (’).
Note	The statement can be a string literal, a string value generated from an expression, or a string value that is stored in a variable.

Details

The following items apply to the SQLEXEC function:

- Use the SQLEXEC function when a FedSQL statement is to be executed only once.
- Allocate, prepare, execute, and free are performed at run time.
- The SQLEXEC function does not support parameters.
- The SQLEXEC function does not support the return of a result set. It cannot be used with a SELECT statement.
- SQLEXEC is similar to the SQL EXECUTE IMMEDIATE statement or the JDBC Statement.executeUpdate (*string*) method.

An SQLSTMT package enables you to execute a FedSQL more than one time, to use parameters, and to access a result set. For more information, see “Using the SQLSTMT Package” in *SAS DS2 Programmer’s Guide*.

Examples

Example 1: SQLEXEC Function

Here is an example of using the SQLEXEC function:

```
tablename='testdata';
name='Jane Doe';
age=25;

s='create table ' || tablename || '(name char(50), age int)';
sqlexec(s);

s='insert into ' || tablename || ' values(''' || name || ''',' || age
||')';
sqlexec(s);
```

Example 2: Use SQLEXEC Function to Create an Empty Table

The following program creates five tables. Each table has three columns and no rows.

```
data test(overwrite=yes);
   declare int i;
   declare int rc;
   method init();
      do i = 1 to 5;
       rc = sqlexec('create table testdata' || i ||
          '(x double, y double, z double)');
       if (rc ne 0) then put 'TEST FAILED';
   end;
end;
enddata;
run;
```

See Also

"Comparing the SQLSTMT Package and the SQLEXEC Function" in *SAS DS2 Programmer's Guide*

SQRT Function

Returns the square root of a value.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

SQRT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the SQRT function:

```
data test(overwrite=yes);
   dcl double a b c;
   method run();
      a=sqrt(36);
      b=sqrt(25);
      c=sqrt(4.4);
      put a=;
      put b=;
      put c=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=6
b=5
c=2.0976176963403
```

SQUANTILE Function

Returns the quantile from a distribution when you specify the right probability (SDF).

Categories:	CAS Quantile
Restriction:	This function is not supported on the CAS server.
Returned data type:	DOUBLE
See:	"SDF Function" on page 1065

Syntax

SQUANTILE(*'distribution'*, *probability*, *parameter-1, ..., parameter-k*)

Arguments

distribution
is a character constant, variable, or expression that identifies the distribution.

Note: The arguments for each of the SQUANTILE Distribution functions are identical to those of the corresponding CDF Distribution functions.

Here are valid distributions:

Distribution	Argument
Bernoulli	'BERNOULLI'
Beta	'BETA'
Binomial	'BINOMIAL'
Cauchy	'CAUCHY'
Chi-Square	'CHISQUARE'
Conway-Maxwell-Poisson	'CONMAXPOI'
Exponential	'EXPONENTIAL'
F	'F'
Gamma	'GAMMA'
Generalized Poisson	'GENPOISSON'
Geometric	'GEOMETRIC'
Hypergeometric	'HYPERGEOMETRIC'
Laplace	'LAPLACE'
Logistic	'LOGISTIC'
Lognormal	'LOGNORMAL'
Negative binomial	'NEGBINOMIAL'
Normal	'NORMAL' | 'GAUSS'
Normal mixture	'NORMALMIX'
Pareto	'PARETO'
Poisson	'POISSON'
T	'T'
Tweedie	'TWEEDIE'
Uniform	'UNIFORM'

Distribution	Argument
Wald (inverse Gaussian)	'WALD' | 'IGAUSS'
Weibull	'WEIBULL'

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

probability
is a numeric constant, variable, or expression that specifies the value of a random variable.

Data type DOUBLE

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

Data type DOUBLE

Details

The SQUANTILE function computes the quantile from the specified continuous or discrete distribution, based on the probability value that is provided. For more information, see the individual distributions noted in the table above.

The Conway-Maxwell-Poisson distribution of the SQUANTILE function returns the counts value *y* that is the smallest, whole number whose SDF value is less than *p*. The syntax for the Conway-Maxwell-Poisson distribution in the SQUANTILE function has the following form:

SQUANTILE('CONMAXPOI', *p*, *λ*, *v*)

p
is a real number between 0 and 1, inclusively.

λ
is similar to the mean, as in the Poisson distribution.

v
is a dispersion parameter.

For more information, see "Conway-Maxwell-Poisson" distribution in the PDF function on page 898.

Examples

Example 1: Using the LOGISTIC Distribution

```
data test(overwrite=yes);
   dcl double p sdf;
   dcl varchar(10) dist;
   method init();
      dist='logistic';
      sdf=squantile(dist, 1.e-20);
      put sdf=;
      p=sdf(dist, sdf);
      put p= /* p will be 1.e-20 */;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
sdf=46.0517018598809
p=1E-20
```

Example 2: Using the Conway-Maxwell-Poisson Distribution

```
data test(overwrite=yes);
   dcl double y;
   method init();
      y=squantile('conmaxpoi',.2,2.3,.4);
      put y=;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
y=12
```

See Also

Functions:

- “CDF Function” on page 370
- “LOGCDF Function” on page 798
- “LOGPDF Function” on page 801
- “LOGSDF Function” on page 804
- “PDF Function” on page 886
- “SDF Function” on page 1065
- “QUANTILE Function” on page 998

STD Function

Returns the standard deviation.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

STD(*expression-1, expression-2* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the STD function:

```
data test(overwrite=yes);
   dcl double val1 val2 val3 val4;
   method run();
      val1=2;
      val2=4;
      val3=6;
      val4=8;
      output;
   end;
enddata;
run;


data test2(overwrite=yes);
   dcl double x1 x2 x3 x4;
   method init();
      set test;
```

```
      x1=std(val1,val3);
      x2=std(val1,val3, .);
      x3=std(val1,val2,val3, 3, 1);
      x4=std(of x1-x3);
      put x1= x2= x3= x4=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=2.82842712474619 x2=2.82842712474619 x3=1.92353840616713 x4=0.52243774525827
```

STDERR Function

Returns the standard error of the mean.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

STDERR(*expression-1, expression-2* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the STDERR function.

```
data test(overwrite=yes);
   dcl double val1 val2 val3 val4;
   method run();
      val1=2;
```

```
      val2=4;
      val3=6;
      val4=8;
      output;
   end;
enddata;
run;


data test2(overwrite=yes);
   dcl double x1 x2 x3 x4;
   method init();
      set test;
      x1=stderr(val1,val3);
      x2=stderr(val1,val3, .);
      x3=stderr(val1,val2,val3, 3, 1);
      x4=stderr(of x1-x3);
      put x1=;
      put x2=;
      put x3=;
      put x4=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=2
x2=2
x3=0.86023252670426
x4=0.37992249109857
```

STREAMINIT Function

Specifies a random-number generator and seed value for generating random numbers.

Categories: CAS
Random Number

Returned data type: DOUBLE

Syntax

Form 1: **STREAMINIT**([*seed*])

Form 2: **STREAMINIT**([*RNG*]

Form 3: **STREAMINIT**([*RNG*, *seed*]

Arguments

seed

a numeric expression that specifies the value used to initialize a stream of pseudorandom numbers.

Range The range of seed values depends on the RNG.

Data type DOUBLE

Note For each thread, all the streams are initialized by using the seed value in the first call to the STREAMINIT function. Subsequent calls to the STREAMINIT function from the same thread are ignored.

RNG

a character expression that specifies the random-number generator (RNG) for generating random or pseudorandom numbers in subsequent calls to the RAND function. The character expression is not case sensitive. The following generators are supported:

RNG	Description
MTHybrid	Hybrid 1998/2002 32-bit Mersenne twister. Default method.
MT1998	1998 32-bit Mersenne twister. Deprecated.
MT32 | MT2002	2002 32-bit Mersenne twister.
MT64	64-bit Mersenne twister.
PCG | PCG64i	64-bit permuted congruential generator.
TF2 | THREEFRY2x64	Threefry 2x64-bit counter-based RNG based on the Threefish encryption function in the Random123 library.
TF4 | THREEFRY4x64	Threefry 4x64-bit counter-based RNG based on the Threefry method in the Random123 library.
HARDWARE	Any supported hardware-based random number generator.
RDRAND	Intel hardware-based RdRand instructions.

For more information about random-number generators, see "Using Random-Number Functions and CALL Routines in the DATA Step" in *SAS Functions and CALL Routines: Reference*.

Restriction The HARDWARE RNG is available only on Intel processors that support the RdRand instruction: Ivy Bridge and later processors.

Tips For each thread, all the streams are initialized by using the seed value in the first call to the STREAMINIT function. Subsequent calls to the STREAMINIT function from the same thread are ignored.

If you do not specify an *RNG*, the RAND function uses the MTHYBRID RNG to compute streams of (pseudo) random numbers.

The PCG RNG provides a good combination of statistical quality, speed, cycle length, and multiple independent streams.

Details

Basics

A sequence of random or pseudorandom values that are generated by an *RNG* is called a *stream*. The STREAMINIT function uses a positive seed to initialize an *RNG*. Within each thread, the first call to the STREAMINIT function initializes the *RNG*; subsequent calls in the same thread do not reinitialize the stream. The *RNG* remains in effect in each thread for the remainder of the DS2 program.

An *RNG* generates random values from the uniform distribution. The RAND function uses one or more uniform variates from the *RNG* to construct a value from a probability distribution. Each call to the RAND function uses at least one value from the stream. The call updates the internal state of the *RNG*.

To generate a reproducible stream of pseudorandom values, call the STREAMINIT function with a positive seed value and specify any of the pseudorandom *RNG*s. If the STREAMINIT function is called before the RAND function, the resulting stream of random numbers is reproducible. Every time you run the DS2 program, it generates the same stream.

If you specify a hardware-based *RNG*, seed values are ignored. Random numbers from a hardware *RNG* are not reproducible. If you run a SAS program that uses a hardware *RNG* multiple times, you get different random numbers every time you run the program.

SAS computes a default seed value if you use a pseudorandom generator and call any of these routines or function:

- Call the STREAMINIT function with a missing zero or negative *seed* argument.
- Call the STREAMINIT function without the *seed* argument.
- Call the RAND function without previously calling the STREAMINIT function.

If the STREAMINIT function initializes the stream or streams, it returns the value of the seed. If the stream or streams have already been initialized by calling the STREAMINIT function or the RAND function, then the STREAMINIT function returns 0.

When called from a DS2 program, the following FCMP functions do not recognize seed values that are set in the DS2 program.

COMPCOST
COMPGED
RAND
STREAMINIT

In a DATA step, all random numbers from RAND are drawn from the same stream by default, but you can create multiple independent streams by using the CALL STREAM routine.

In PROC DS2, random numbers from RAND that are called inside an FCMP package come from a separate stream. To get reproducible random numbers, use the CALL STREAMINIT routine inside each FCMP function that calls RAND. To get independent streams, also use the CALL STREAM routine inside each FCMP function that calls RAND. If you do so, the results from DS2 will be consistent with a DATA step. For an example, see "Considerations and Limitations When Using the FCMP Package" in *SAS DS2 Programmer's Guide*.

Range of *seed* Values

For MTHybrid, MT32, and MT1998, the seed value can be an integer from 1 to $2^{32}-1=4294967295$.

For MT64, PCG, Threefry2x64, and Threefry4x64, the seed value can be an integer from 1 to $2^{64}-1025=18446744073709549568$. Some large integers do not have an exact representation in double-precision floating-point numbers.

If the seed argument is missing or less than or equal to zero, SAS generates a seed value as described in "Using Random-Number Functions and CALL Routines in the DATA Step" in *SAS Functions and CALL Routines: Reference*.

If a positive seed value is out of range, the mantissa of the seed argument is scaled to an integer that is within range.

Examples

Example 1: Specify a Seed Value to Create a Reproducible Stream of Pseudorandom Numbers with the RAND Function

The following program illustrates how to specify a seed value with the STREAMINIT function to create a reproducible stream of pseudorandom numbers with the RAND function.

```
proc ds2;
data random (overwrite=yes);
   dcl double i x1;
   method run();
      streaminit(123);
      do i=1 to 10;
         x1=rand('cauchy');
```

```
            output;
         end;
      end;
   enddata;
   run;
   quit;

   proc print data=random;
      id i;
   run;
   quit;
```

Output 7.26 *Number String Seeded with the STREAMINIT Function*

The SAS System

i	x1
1	-0.17593
2	3.76106
3	1.23427
4	0.49095
5	-0.05094
6	0.72496
7	-0.51646
8	7.61304
9	0.89784
10	1.69348

Example 2: Use the 64-bit Mersenne Twister Random-Number Generator

The following program illustrates how to specify a seed value with the MT64 RNG to create a stream of pseudorandom numbers with the RAND function.

```
proc ds2;
data random (overwrite=yes drop=(MT64));
   dcl double i x1;
   method run();
      streaminit('MT64', 123);
      do i=1 to 10;
         x1=rand('cauchy');
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=random;
```

```
   id i;
run;
quit;
```

Output 7.27 *Number String Seeded with the MT64 Random-Number Generator*

The SAS System

i	x1
1	-3.3370
2	1.8546
3	1.0090
4	-2.1520
5	-27.0630
6	-2.6697
7	-7.6492
8	0.2939
9	3.7349
10	-0.5185

Example 3: Specify a Seed and Use the 64-bit Mersenne Twister Random-Number Generator

The following program shows a seed value with the MT64 RNG to create a stream of pseudorandom numbers with the RAND function.

```
proc ds2;
data random (overwrite=yes drop=(MT64));
   dcl double i x1;
   method run();
      streaminit('MT64', 128645);
      do i=1 to 10;
         x1=rand('cauchy');
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=random;
   id i;
run;
quit;
```

Output 7.28 *Number String with the Seed Number and the 64-bit Mersenne Twister Random-Number Generator*

The SAS System

i	x1
1	0.00658
2	0.55544
3	9.67566
4	0.54930
5	0.44583
6	-2.88661
7	1.90463
8	0.01001
9	-0.01532
10	0.72421

See Also

Functions:

- "RAND Function" on page 1004

STRIP Function

Returns a character string with all leading and trailing blanks removed.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

STRIP(*expression*)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The STRIP function returns the argument with all leading and trailing blanks removed. If the argument is blank, STRIP returns a string with a length of zero.

If the value that is trimmed is shorter than the length of the receiving variable, SAS pads the value with new trailing blanks.

Note: The STRIP function is useful for concatenation because the concatenation operator does not remove trailing blanks.

Comparisons

The following list compares the STRIP function with the TRIM function:

- For blank character strings, the STRIP and TRIM functions both return a string with a length of zero.
- For strings that lack leading blanks, the STRIP and TRIM functions return the same value.

Example

The following program shows the results of using the STRIP function to delete leading and trailing blanks.

```
proc ds2;
data lengthn (overwrite=yes);
   dcl char(8) string;
   method init();
      string='abcd    '; output;
      string='  abcd  '; output;
      string='    abcd'; output;
      string='abcdefgh'; output;
      string='x y x'; output
   end;
enddata;
run;
```

```
data stripstring (overwrite=yes);
   dcl varchar(10) original;
   dcl varchar(10)  stripped;
   method run();
      set lengthn;
      original = '*' || string || '*';
      stripped = '*' || strip(string) || '*';
   end;
enddata;
run;

proc print data=stripstring;
run;
```

Output 7.29 *Results from the STRIP Function*

The SAS System

Obs	ORIGINAL	STRIPPED	STRING
1	*abcd *	*abcd*	abcd
2	* abcd *	*abcd*	abcd
3	* abcd*	*abcd*	abcd
4	*abcdefgh*	*abcdefgh*	abcdefgh
5	*x y x *	*x y x*	x y x

See Also

Functions:

- “CAT Function” on page 354
- “CATS Function” on page 361
- “CATT Function” on page 364
- “CATX Function” on page 366
- “LEFT Function” on page 781
- “TRIM Function” on page 1151

SUBSTR (left of =) Function

Replaces a substring of content in a character variable.

Categories: CAS
Character

Syntax

SUBSTR (*character-variable*, *position-expression* [, *length-expression*]) = *characters-to-replace*

Arguments

character-variable
specifies character variable.

Restriction *character-variable* must be a variable or an array reference. It cannot be an expression or a literal.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

position-expression
specifies any valid expression that evaluates or can be coerced to an integer and that specifies the character position of the first character in the substring in the *character-variable* that is to be replaced.

Requirement *position-expression* must be greater than or equal to 1.

Data type BIGINT. Other types are coerced to BIGINIT.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

length-expression
specifies any valid expression that evaluates or can be coerced to an integer and that specifies the character length of the substring in *character-variable* that is to be replaced. If you do not specify *length-expression*, *length-expression* defaults to the character length of the substring in *character-variable* that extends from the character position that you specify to the end of the string.

Restriction *length-expression* cannot be larger than the character length of the substring in *character-variable* that extends from *position-expression* to the maximum length of *character-variable*.

Data type BIGINT. Other types are coerced to BIGINIT.

Note *length-expression* can be larger than the character length of the substring in *character-variable* that is to be replaced.

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

characters-to-replace
specifies any valid expression that evaluates or can be coerced to a character string and represents the string that replaces a substring in *character-variable*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

When you use the SUBSTR function on the left side of an assignment statement, a substring of the content in the character variable is replaced with the character string value on the right side.

Note: In the following discussion, length refers to character-based length as opposed to byte-based length.

If *length-expression* is less than or equal to the length of the substring that extends from *position-expression* to the end of the string in *character-variable*, the substring starting at *position-expression* and extending *length-expression* characters is replaced. If *length-expression* is greater than the length of the substring that extends from *position-expression* to the end of the string in *character-variable*, the substring starting at *position-expression* and extending to the end of the string is replaced.

If the length of characters-to-replace is longer than *length-expression*, characters-to-replace is truncated to *length-expression* characters before being used as the substring replacement in *character-variable*. If the length of characters-to-replace is shorter than *length-expression*, *characters-to-replace* is blank padded to *length-expression* characters before being used as the substring replacement in *character-variable*.

If the length of the resulting character string after the substring replacement is longer than the maximum length of *character-variable*, the resulting character string is truncated to the maximum length of *character-variable*.

For a VARCHAR *character-variable*, if *position-expression* is greater than the current length of *character-variable* but less than or equal to the maximum length of *character-variable*, the *position-expression* argument is considered valid. The SUBSTR (left of =) function behaves as if blank padding were added to the *character-variable* to extend its current length to the length of the *position-expression* argument + *length-expression* argument. The content of the blank-padded *character-variable* is then replaced with characters from the *characters-to-replace* expression.

The SUBSTR (left of =) function does not modify *character-variable* if any of the following are true:

- *character-variable* is a null value (ANSI mode)

- *length-expression* is less than 1
- *length-expression* is a missing (SAS mode) or null value (ANSI mode)
- *length-expression* is larger than the length of the substring in *character-variable* that extends from *position-expression* to the maximum length of *character-variable*
- *position-expression* is less than 1
- *position-expression* is a missing (SAS mode) or null value (ANSI mode)
- *position-expression* is larger than the length of the substring in *character-variable*

Note: In addition, a warning is written to the log stating that the argument is invalid.

Example

The following programs illustrate the SUBSTR (left of=) function.

```
/*** replaces 3 characters starting at position 1 ***/
data test (overwrite=yes);
   dcl char mystring;
   method run();
      mystring='kidnap';
      substr(mystring, 1, 3)='cat';
      put mystring=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
mystring=catnap
```

```
/**** mystring is not declared ****/
/****  and is assigned a length of 1 ****/
/**** replaces 1 character starting at position 1 ****/
data test (overwrite=yes);
   method run();
      mystring=' ';
      substr(mystring, 1)='cat';
      put mystring=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
c
```

```
/* position=missing value */
```

```
/* returns mystring because length-expression is missing */
data test (overwrite=yes);
   dcl char mystring;
   method run();
      mystring='kidnap';
      substr(mystring,.)='cat';
      put mystring=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
kidnap
```

```
/**** The VARCHAR variable, mystring, has an initial length of 3 ****/
/**** is blank padded to 7 characters ****/
/**** (position-expression + length-expression) ****/
/**** 3 characters from the characters-to-replace expression ****/
/**** are replaced in mystring starting at position 4 ****/
data test (overwrite=yes);
   dcl varchar(10) mystring;
   method run();
      mystring='cat';
      substr(mystring, 4, 3)='napper';
      put mystring=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
catnap
```

See Also

Functions:

- “SUBSTR (right of =) Function” on page 1108
- “SUBSTRN Function” on page 1111

SUBSTR (right of =) Function

Extracts a substring from an argument.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

SUBSTR(*character-expression*, *position-expression* [, *length-expression*])

Arguments

character-expression
specifies a character constant, variable, or expression.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

position-expression
specifies a numeric constant, variable, or expression that is the beginning character position.

Requirement *position-expression* must be greater than or equal to zero.

Data type BIGINT

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

length-expression
specifies a numeric constant, variable, or expression that is the length of the substring to extract. If you do not specify *length-expression*, the SUBSTR (right of) function returns the substring that extends from the character position that you specify to the end of the string.

Data type BIGINT

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The following information applies to the SUBSTR function:

- The SUBSTR (right of =) function returns a missing (SAS mode) or null value (ANSI mode) if any of the following are true:
  - *position-expression* is less than 1 or greater than the length of *character-expression*
  - *character-expression* is a missing (SAS mode) or null value (ANSI mode)
  - *position-expression* is a missing (SAS mode) or null value (ANSI mode)

Note: With any of these conditions, a warning is written to the log stating that the second argument is invalid.

- The SUBSTR (right of =) function returns the substring from *position-expression* to the end of the *character-expression* if *length-expression* meets any of the following conditions:
  - is less than 1 (beginning in SAS V9.4M6)
  - is not specified
  - is greater than the length of the substring from *position-expression* to the end of the *character-expression*
  - is missing (SAS mode) or a null value (ANSI mode)

Note: With the last three conditions, a warning is written to the log stating that the third argument is invalid.

Example

The following programs illustrate the SUBSTR (right of =) function:

```
/**** selects all chars starting at position 5 to the end of the
string ****/
data test (overwrite=yes);
   dcl char(12) a b;
   method run();
      a='chsh234960b3';
      b=substr(a,5);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
234960b3
```

```
/**** selects 6 chars starting at position 5 ****/
data test (overwrite=yes);
   dcl char(12) a b;
   method run();
      a='chsh234960b3';
      b=substr(a,5, 6);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
234960
```

```
/**** selects all chars starting at position 5 to the end of the
string ****/
/**** because of the invalid third argument ****/
```

```
data test (overwrite=yes);
   dcl char(12) a b;
   method run();
      a='chsh234960b3';
      b=substr(a, 5, 15);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
234960b3
WARNING: Argument 3 to function SUBSTR is invalid.
```

```
/**** selects all chars starting at position 5 to the end of the
string ****/
/**** because of the invalid third argument ****/
data test (overwrite=yes);
   dcl char(12) a b;
   method run();
      a='chsh234960b3';
      b=substr(a, 5, -5);
      put b;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
234960b3
WARNING: Argument 3 to function SUBSTR is invalid.
```

See Also

Functions:

- “SUBSTR (left of =) Function” on page 1105
- “SUBSTRN Function” on page 1111

SUBSTRN Function

Returns a substring, allowing a result with a length of zero.

Categories:	CAS Character
Returned data type:	CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

SUBSTRN(*expression*, *position-expression*[, *length-expression*])

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string or numeric value.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR, DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

position-expression
specifies any valid expression that evaluates to a BIGINT and that specifies the position of the first character in the substring.

Data type BIGINT

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

length-expression
specifies any valid expression that evaluates to a BIGINT and that specifies the length of the substring.

Data type BIGINT

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The Basics

The following information applies to the SUBSTRN function:

- The SUBSTRN function returns the substring from *position-expression* to the end of the *character-expression* if *length-expression* meets either of the following conditions:
  - is not specified
  - is greater than the length of the substring from *position-expression* to the end of the *expression*
- The SUBSTRN function returns a missing (SAS mode) or null value (ANSI mode) with a length of zero if any of the following conditions are true:
  - *position-expression* is a missing (SAS mode) or null value (ANSI mode)
  - *length-expression* is a missing (SAS mode) or null value (ANSI mode)

Note: In addition, an error is written to the log stating that the argument is invalid.

- The SUBSTRN function returns a missing value (SAS mode) or an empty string (ANSI mode) with a length of zero if any of the following conditions are true:
  - *expression* is a missing (SAS mode) or null value (ANSI mode)
  - *length-expression* is less than 1
  - *position-expression* is less than 1 or greater than the length of *character-expression*

Comparisons

The following table lists comparisons between the SUBSTRN and the SUBSTR functions:

Condition	Function	Result
the value of *position-expression* is nonpositive	SUBSTRN	returns a missing value (SAS mode) or empty string (ANSI mode) of length 0
the value of *position-expression* is nonpositive	SUBSTR	returns a missing (SAS mode) or null value (ANSI mode) of length 0
the value of *length-expression* is nonpositive	SUBSTRN	returns a missing value (SAS mode) or empty string (ANSI mode) of length 0
the value of *length-expression* is nonpositive	SUBSTR	returns the substring from *position-expression* to the end of the *character-expression*
the value of the *length-expression* is a missing (SAS mode) or null value (ANSI mode)	SUBSTRN	returns a missing or null value with a length of zero and writes an error to the log that the third argument is invalid
the value of the *length-expression* is a missing (SAS mode) or null value (ANSI mode)	SUBSTR	returns substring from *position-expression* to the end of the *expression* and writes a warning to the log that the third argument is invalid

Examples

Example 1: Manipulating Strings with the SUBSTRN Function

The following program shows how to manipulate strings with the SUBSTRN function.

```
proc ds2;
data test (overwrite=yes);
   dcl char(6) string result;
   dcl double position length;
   retain string 'abcd';
   drop string;
   method run();
      do position=-1 to 6;
         do length=max(-1,-position) to 7-position;
             result=substrn(string, position, length);
             output;
         end;
      end;
   end;
enddata;
run;
quit;

proc print data=test;
run;
```

Output 7.30 *Partial Output from the SUBSTRN Function*

Obs	result	position	length
1		-1	1
2		-1	2
3	a	-1	3
4	ab	-1	4
5	abc	-1	5
6	abcd	-1	6
7	abcd	-1	7
8	abcd	-1	8
9		0	0
10		0	1
11	a	0	2
12	ab	0	3
13	abc	0	4
14	abcd	0	5
15	abcd	0	6
16	abcd	0	7
17		1	-1
18		1	0
19	a	1	1
20			2

Example 2: Comparison between the SUBSTR and SUBSTRN Functions

The following program compares the results of using the SUBSTR function and the SUBSTRN function when the first argument is numeric.

```
data _null_;
   dcl char substr_result substrn_result;
   method run();
      substr_result='*' || substr('   1234.5678',2,6) || '*';
      put substr_result=;
      substrn_result='*' || substrn('1234.5678',2,6) || '*';
      put substrn_result=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
substr_result=*  1234*
substrn_result=*234.56*
```

Example 3: Using SUBSTRN with a VARCHAR Argument

The following program uses the SUBSTRN function in a string with a VARCHAR data type.

```
proc ds2;
 data _null_;
    dcl varchar(100000) inarg inarg2 inarg3;
    dcl double l;
    method init();
       inarg = 'x';
       inarg2 = repeat(inarg,30000-1)||'abc';
       l=length(inarg2);
       inarg3 = substrn(inarg2,29998,5);
       put l= inarg3=;
       inarg2 = repeat(inarg,90000-1)||'abc';
       l=length(inarg2);
       inarg3 = substrn(inarg2,89998,5);
       put l= inarg3= ' (should be xxxab)';
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
l=30003 inarg3=xxxab
l=90003 inarg3=xxxab  (should be xxxab)
```

See Also

Functions:

- “SUBSTR (left of =) Function” on page 1105
- “SUBSTR (right of =) Function” on page 1108

SUM Function

Returns the sum of the non-null or nonmissing arguments.

Categories:	CAS Descriptive Statistics
Returned data type:	BIGINT, DECIMAL, DOUBLE

Syntax

SUM(*expression-1*, *expression-2* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	At least two arguments are required.
Data type	BIGINT, DECIMAL, DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

Null and missing values are ignored and not included in the computation. If all of the arguments have missing values, the result is a missing value. If all the arguments have a null value, the result is a null value.

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Example

The following program illustrates the SUM function:

```
data test(overwrite=yes);
   dcl double u v w x y z;
   dcl double val1 val2 val3 val4;
   method run();
      val1=4;
      val2=3;
      val3=9;
      val4=8;
      u=sum(4,3,9,8);
      v=sum(val1, val2, val3, val4);
      w=sum(val1, val2, val3, val4, .);
      x=sum(of val1-val2, of val3-val4);
      y=sum(of val1-val2);
      z=sum(of val:);
      put u= v= w= x= y= z=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
u=24 v=24 w=24 x=24 y=7 z=24
```

SUMABS Function

Returns the sum of the absolute values of the nonmissing arguments.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

SUMABS(*value-1*[, ...*value-n*])

Arguments

value
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

If all arguments have null or missing values, then the result is a null or missing value. Otherwise, the result is the sum of the absolute values of the nonmissing values.

Examples

Example 1: Calculating the Sum of Absolute Values

The following program returns the sum of the absolute values of the nonmissing arguments.

```
data _null_;
   dcl double x;
   method run();
      x=sumabs(1,.,-2,0,3,.q,-4);
      put x=;
```

```
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
x=10
```

Example 2: Calculating the Sum of Absolute Values When You Use a Variable List

The following program uses a variable list and returns the sum of the absolute value of the nonmissing arguments.

```
data _null_;
   dcl double x1 x2 x3 x4 x5 x;
   method run();
      x1 = 1;
      x2 = 3;
      x3 = 4;
      x4 = 3;
      x5 = 1;
      x = sumabs(x1, x2, x3, x4, x5);
   put x=;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
x=12
```

SYSGET Function

Returns the value of the specified operating environment variable.

Categories: CAS
Special

Restriction: This function is supported in CAS only for users with administrative-level capabilities.

Syntax

Windows and UNIX:

SYSGET('*environment-variable*')

z/OS:

SYSGET(*operating-environment-variable*)

Arguments

environment-variable
is a character constant, variable, or expression with a value that is the name of an environment variable under Windows and UNIX.

Requirement This argument must be enclosed in single quotation marks.

operating-environment-variable
is a character constant, variable, or expression with a value that is the name of a simulated environment variable under z/OS.

Details

General Information

The SYSGET function returns the value of an environment variable as a character string. For example, this statement returns the value of the HOME environment variable under UNIX:

```
here=sysget('HOME');
```

If the SYSGET function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

If the value of the operating environment variable is truncated or the variable is not defined in the operating environment, SYSGET displays a warning message in the SAS log.

SYSGET Specifics under UNIX

The case of the value that you supply in the *environment-variable* argument must agree with the case of the variable that is stored in the UNIX operating environment.

SYSGET Specifics under z/OS

z/OS does not have native environment variables, but SAS supports three types of simulated environment variables that SYSGET can access.

- variables that have been created through the SET system option
- variables that have been defined in a TKMVSENV file
- under TSO, variables in the calling REXX exec or CLIST

SYSGET searches for the specified *operating-environment-variable* in each of these three locations, in the order specified in the preceding list. If the specified variable is not found in any of the locations, then the error message `NOTE: Invalid argument to the function SYSGET` is generated, and _ERROR_ is set to 1.

Names of TKMVSENV variables are case-sensitive, but names of SET, REXX, and CLIST variables are not case-sensitive.

Example: Obtain Environment Variable Values under Windows

This example obtains the value of the PATH environment variable in the Windows environment:

```
data _null_;
   dcl char(256) x;
   method run();
      x = sysget('PATH');
      put x;
   end;
enddata;
run;
```

The following line is written to the SAS log.

```
C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS
\System32\WindowsPowerShell\v1.0
\;C:\Program Files\SASHome\SASFoundation\9.4\ets\sasexe;C:\Program
Files\SASHome\Secure\ccme4;C:\Program Files\SASHome\x86\Secure\ccme4;C:\Program
Files (x86)\PRISM;
```

TAN Function

Returns the tangent.

Categories: CAS
Trigonometric

Returned data type: DOUBLE

Syntax

TAN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value in radians.

Restriction *expression* cannot be an odd multiple of π /2

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the TAN function:

```
data test(overwrite=yes);
   dcl double x1 x2 x3;
   method run();
      x1=tan(0.5);
      x2=tan(0);
      x3=tan(3.14159/3);
      put x1= x2= x3=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=0.54630248984379 x2=0 x3=1.73204726945457
```

See Also

Functions:

- "TANH Function" on page 1122

TANH Function

Returns the hyperbolic tangent.

Categories: CAS
Hyperbolic

Returned data type: DOUBLE

Syntax

TANH(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction *expression* cannot be an odd multiple of π/2

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The TANH function returns the hyperbolic tangent of the argument, which is given by the following equation.

$$\frac{e^{argument} - e^{-argument}}{e^{argument} + e^{-argument}}$$

Example

The following program illustrates the TANH function:

```
data test (overwrite=yes);
   dcl double a b c;
   method run();
      a=tanh(0);
      b=tanh(0.5);
      c=tanh(-0.5);
      put 'a= ' a;
      put 'b= ' b;
      put 'c= ' c;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=  0
b=  0.46211715726
c=  -0.46211715726
```

See Also

Functions:

- "TAN Function" on page 1121

TIME Function

Returns the current time of day as a numeric SAS time value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

TIME( )

Details

The TIME function does not take any arguments. It produces the current time in the form of a SAS time value.

Example

The following programs illustrate the TIME function:

```
data test(overwrite=yes);
   dcl double current;
   method run();
      current=time();
      put current=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
current=40486.0569992065
```

```
data test(overwrite=yes);
   dcl double current;
   method run();
      current=time();
      put current=time.;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
current=11:14:46
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "DATE Function" on page 500
- "DATETIME Function" on page 504

TIMEPART Function

Extracts a time value from a SAS datetime value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

TIMEPART(*datetime*)

Arguments

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the TIMEPART function:

```
data test(overwrite=yes);
   dcl double dttm tm;
   dcl char(10) x;
```

```
      method run();
         dttm=datetime();
         put dttm;
         x=put(timepart(dttm), time.);
         put x;
      end;
   enddata;
   run;
```

SAS writes the following output to the log:

```
1866291827.48899
14:23:47
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "DATEPART Function" on page 503

TIMEVALUE Function

Returns the equivalent of a reference amount at a base date by using variable interest rates.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

TIMEVALUE(*base-date*, *reference-date*, *reference-amount*, *compounding-interval*, *date-1*, *rate-1*[, ...*date-n*, *rate-n*])

Arguments

base-date
specifies the time value of the *reference-amount* at the *base-date*.

Requirement *Base-date* is a SAS date.

Data type DOUBLE

reference-date
specifies the date of *reference-amount*.

Requirement *Reference-date* is a SAS date.

Data type DOUBLE

reference-amount
specifies the amount at the *reference-date*.

Data type DOUBLE

compounding-interval
specifies the compounding interval.

Requirement *Compounding-interval* is a SAS interval.

Data type CHAR

date
specifies the time at which *rate* takes effect. Each date is paired with a rate.

Requirement *Date* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate as numeric percentage that starts on *date*. Each rate is paired with a date.

Data type DOUBLE

Details

The following details apply to the TIMEVALUE function:

- The values for rates must be between –99 and 120.
- The list of date-rate pairs does not need to be sorted by date.
- When multiple rate changes occur on a single date, the TIMEVALUE function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the *reference-date* and the *base-date*.

Example

- You can express the accumulated value of an investment of $1,000 at a nominal interest rate of 10% compounded monthly for one year as the following:

```
data _null_;
   dcl double bd rd d amount_base1;
   method run();
      bd= to_double(date'2001-01-01');
      rd= to_double(date'2000-01-01');
      d= to_double(date'2000-01-01');
      amount_base1 = timevalue(bd, rd, 1000, 'month', d, 10);
      put amount_base1;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1104.71306744129
```

- If the interest rate jumps to 20% halfway through the year, the resulting calculation would be as follows:

```
data _null_;
   dcl double bd rd d1 d2 amount_bases;
   method run();
      bd= to_double(date'2001-01-01');
      rd= to_double(date'2000-01-01');
      d1= to_double(date'2000-01-01');
      d2= to_double(date'2000-07-01');
      amount_base2 = timevalue(bd, rd, 1000, 'month', d1, 10, d2, 20);
      put amount_base2;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
1160.63657783831
```

- The date-rate pairs do not need to be sorted by date. This flexibility allows amount_base2 and amount_base3 to assume the same value:

```
data _null_;
   dcl double bd rd d1 d2 amount_base3;
   method run();
      bd= to_double(date'2001-01-01');
      rd= to_double(date'2000-01-01');
      d1= to_double(date'2000-07-01');
      d2= to_double(date'2000-01-01');
      amount_base3 = timevalue(bd, rd, 1000, 'month', d1, 20, d2, 10);
      put amount_base3;
end;
enddata;
run;
```

SAS writes the following output to the log:

```
1160.63657783831
```

TINV Function

Returns a quantile from the *t* distribution.

Categories: CAS
Quantile

Returned data type: DOUBLE

Syntax

TINV(*p*, *df*[, *nc*])

Arguments

p
specifies any valid expression that evaluates to a numeric probability.

Range $0 < p < 1$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

df
specifies any valid expression that evaluates to a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

nc
specifies any valid expression that evaluates to a numeric noncentrality parameter.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The TINV function returns the pth quantile from the Student's *t* distribution with degrees of freedom *df* and a noncentrality parameter *nc*. The probability that an observation from a *t* distribution is less than or equal to the returned quantile is *p*.

TINV accepts a noninteger degree of freedom parameter *df*. If the optional parameter *nc* is not specified or is 0, the quantile from the central *t* distribution is returned.

CAUTION
For large values of nc, the algorithm can fail. In that case, a missing value is returned.

Comparisons

TINV is the inverse of the PROBT function.

Example

The following program illustrates the TINV function:

```
data _null_;
  dcl double x y;
  method run();
     x=tinv(.95, 2);
     y=tinv(.95, 2.5, 3);
     put x y;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=2.91998558035372
y=11.0338336251942
```

See Also

Functions:

- "PROBT Function" on page 976

TNONCT Function

Returns the value of the noncentrality parameter from the Student's *t* distribution.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

TNONCT(*x*, *df*, *prob*)

Arguments

x
is a numeric random variable.

Data type DOUBLE

df
is a numeric degrees of freedom parameter.

Range *df* > 0

Data type DOUBLE

prob
is a probability.

Range 0 < *prob* < 1

Data type DOUBLE

Details

The TNONCT function returns the nonnegative noncentrality parameter from a noncentral *t* distribution whose parameters are *x*, *df*, and *nc*. A Newton-type algorithm is used to find a root *nc* of the equation

$$P_t(x|df, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_t(x|df,nc) = \frac{1}{\Gamma\left(\frac{df}{2}\right)} \int_0^{\infty} v^{\frac{df}{2}-1} e^{-v} \int_{-\infty}^{x\sqrt{\frac{2v}{df}}} e^{-\frac{(u-nc)^2}{2}} du dv$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

The following program computes the noncentrality parameter from the *t* distribution.

```
proc ds2;
data work /overwrite=yes;
   dcl double x df nc prob ncc;
   method init();
      x=2;
      df=4;
      do nc=1 to 3 by .5;
         prob=probt(x, df, nc);
         ncc=tnonct(x, df, prob);
         output;
      end;
   end;
enddata;
run;
quit;

proc print data=work;
run;
```

Output 7.31 *Output from the Computations of the Noncentrality Parameter for the t Distribution*

The SAS System

Obs	x	df	nc	prob	ncc
1	2	4	1.0	0.76457	1.00000
2	2	4	1.5	0.61893	1.50000
3	2	4	2.0	0.45567	2.00000
4	2	4	2.5	0.30115	2.50000
5	2	4	3.0	0.17702	3.00000

See Also

Functions:

- “CNONCT Function” on page 430
- “FNONCT Function” on page 650

TO_DATE Function

Returns a DATE value from a DOUBLE value that specifies a SAS date value.

Categories: CAS
Date and Time

Returned data type: DATE

Syntax

TO_DATE(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

A DOUBLE value that specifies a SAS date value represents the number of days between January 1, 1960, and a specified date. SAS can perform calculations on dates ranging from A.D. 1582 to A.D. 19,900. Dates before January 1, 1960, are negative numbers; dates after January 1, 1960, are positive numbers.

- SAS date values account for all leap year days, including the leap year day in the year 2000.
- SAS date values can reliably tell you what day of the week a particular day fell on as far back as September 1752, when the calendar was adjusted by dropping several days. SAS day-of-the-week and length-of-time calculations are accurate in the future to A.D. 19,900.

Example

The following program converts a DOUBLE value that specifies a SAS date value, to a DATE value.

```
/* SAS date to date */
data _null_;
   dcl date da;
   dcl double d;
   method init();
      d = 22096;
      da = to_date(d);
      put d=YYMMDD10. da=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
d=2020-06-30 da=2020-06-30
```

See Also

Functions:

- “TO_DOUBLE Function” on page 1134
- “TO_TIME Function” on page 1137
- “TO_TIMESTAMP Function” on page 1139

TO_DOUBLE Function

Returns a DOUBLE value that specifies a SAS date, time, or datetime value, from a DATE, TIME, or TIMESTAMP value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

TO_DOUBLE (*date* | *time* | *timestamp*)

Arguments

date
is a date constant, variable, or expression.

time
is a time constant, variable, or expression.

timestamp
is a timestamp constant, variable, or expression.

Details

The following list describes the values that can be returned by the TO_DOUBLE function:

- A SAS date value represents the number of days between January 1, 1960, and a specified date. SAS can perform calculations on dates ranging from A.D. 1582 to A.D. 19,900. Dates before January 1, 1960, are negative numbers; dates after January 1, 1960, are positive numbers.
  - SAS date values account for all leap year days, including the leap year day in the year 2000.
  - SAS date values can reliably tell you what day of the week a particular day fell on as far back as September 1752, when the calendar was adjusted by dropping several days. SAS day-of-the-week and length-of-time calculations are accurate in the future to A.D. 19,900.
- A SAS time value represents the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.
- A timestamp is a record of the date and time at which a certain event occurred.
- A SAS datetime value represents the number of seconds between January 1, 1960, and an hour/minute/second within a specified date.

Examples

Example 1: Converting a TIMESTAMP Value to a DOUBLE Value

The following program converts a TIMESTAMP value to a DOUBLE value that specifies a SAS datetime value.

```
/* Timestamp to SAS datetime */
data _null_;
   dcl timestamp ts;
   dcl double d_asd;
   dcl date d;
   method init();
      ts = timestamp '2019-10-19 16:51:36.0625';
      d_asd = to_double(ts);
      put ts=;
```

```
      put d_asd;
      put d_asd=DATETIME28.9;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
ts=2019-10-19 16:51:36.062500000
1887123096.0625
d_asd=19OCT2019:16:51:36.062500000
```

Example 2: Converting a Date Value to a SAS Date Value

The following program converts a DATE value to a DOUBLE value that specifies a SAS date value:

```
/* Date to SAS date */
data _null_;
   dcl date da;
   dcl double d;
   method init();
      da = date '2019-10-19';
      d = to_double(da);
      put d;
      put d=YYMMDD10.;
      put da=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
21841
d=2019-10-19
da=2019-10-19
```

Example 3: Converting a Time Value to a SAS Time Value

The following program converts a TIME value to a SAS time value:

```
/* Time to SAS time */
data _null_;
   dcl time t;
   dcl double d;
   method init();
      t = time '16:51:36.0625';
      d = to_double(t);
      put d=TIME18.9 t=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
d=16:51:36.062500000 t=16:51:36.062500000
```

Example 4: Converting a NULL Timestamp to a SAS Datetime Value

The following program converts a NULL TIMESTAMP to a SAS datetime value:

```
/* NULL timestamp to SAS datetime */
data _null_;
   dcl timestamp ts;
   dcl double d;
   method init();
      ts = null;
      d = to_double(ts);
      put d=DATETIME28.9 ts=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
d=                          . ts=
```

See Also

Functions:

- “TO_DATE Function” on page 1133
- “TO_TIME Function” on page 1137
- “TO_TIMESTAMP Function” on page 1139

TO_TIME Function

Returns a TIME value from a DOUBLE value that specifies a SAS time value.

Categories: CAS
Date and Time

Returned data type: TIME

Syntax

TO_TIME(*date*)

Arguments

date

specifies any valid expression that represents a SAS time value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

A SAS time value represents the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.

Example

The following program converts a SAS time value to a formatted time value.

```
/* SAS time to time */
data _null_;
   dcl time t;
   dcl double d;
   method init();
      d = 45911.68;
      t = to_time(d);
      put d=TIME18.9 t=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
d=12:45:11.680000000 t=12:45:11.680000000
```

See Also

Functions:

- "TO_DATE Function" on page 1133
- "TO_DOUBLE Function" on page 1134
- "TO_TIMESTAMP Function" on page 1139

TO_TIMESTAMP Function

Returns a TIMESTAMP value from a DOUBLE value that specifies a SAS time value.

Categories: CAS
Date and Time

Returned data type: TIMESTAMP

Syntax

TO_TIMESTAMP(*date*)

Arguments

date
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

A SAS datetime value is a DOUBLE value that represents the number of seconds between January 1, 1960, and an hour/minute/second within a specified date.

Examples

Example 1: Converting a SAS Datetime Value to a Timestamp Value

The following program converts a SAS datetime value to a timestamp value.

```
/* SAS datetime to timestamp */
data _null_;
   dcl timestamp ts;
   dcl double d;
   method init();
      d = 1845773470.3;
      ts = to_timestamp(d);
      put d=DATETIME28.9 ts=;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
d=28JUN2018:02:51:10.299999952 ts=2018-06-28 02:51:10.299999952
```

Example 2: Converting a SAS Datetime Value That Is Missing

The following program shows how SAS handles the conversion of a missing SAS datetime value.

```
/* Missing SAS datetime to timestamp */
data _null_;
   dcl timestamp ts;
   dcl double d;
   method init();
      d = .;
      ts = to_timestamp(d);
      put d=DATETIME28.9 ts=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
d=                          . ts=
```

See Also

Functions:

- “TO_DATE Function” on page 1133
- “TO_DOUBLE Function” on page 1134
- “TO_TIME Function” on page 1137

TODAY Function

Returns the current date as a numeric SAS date value.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

TODAY( )

Details

The TODAY function does not take any arguments. It produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

For more information about how DS2 handles dates, see "DS2 Expressions" in *SAS DS2 Programmer's Guide*.

Example

The following program illustrates the TODAY function:

```
data test(overwrite=yes);
   dcl double td;
   dcl char(10) tday;
   method run();
      td=today();
      tday=put(td, date.);
      put td;
      put tday;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
21600
20FEB19
```

TRANSLATE Function

Replaces specific characters in a character expression.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRANSLATE(*expression*, *to-characters*, *from-characters*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string. *expression* contains the original character value.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

to-characters
specifies the characters that you want TRANSLATE to use as substitutes.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

from-characters
specifies the characters that you want TRANSLATE to replace.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Values of *to-characters* and *from-characters* correspond on a character-by-character basis; TRANSLATE changes the first character in *from-characters* to the first character in *to-characters*, and so on. If *to-characters* has fewer characters than *from-characters*, TRANSLATE changes the extra *from-characters* characters to blanks. If *to-characters* has more characters than *from-characters*, TRANSLATE ignores the extra *to-characters*.

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this scan, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANWRD function differs from TRANSLATE in that it scans for words (or patterns of characters) and replaces those words with a second word (or pattern of characters).

Example

The following program illustrates the TRANSLATE function.

```
data _null_;
   dcl char(10) x y;
   method run();
      x=translate('XYZW', 'AB', 'VW');
      y='AABBAABABB';
      y=translate(y, '12', 'AB');
      put x;
      put y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
XYZB
1122112122
```

See Also

Functions:

- “TRANSTRN Function” on page 1143
- “TRANWRD Function” on page 1146

TRANSTRN Function

Replaces or removes all occurrences of a substring in a character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRANSTRN(*source-expression*, *target-expression*, *replacement-expression*)

Arguments

source-expression
specifies any valid expression that evaluates or can be coerced to a character string and whose characters you want to translate.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

target-expression
specifies any valid expression that evaluates or can be coerced to a character string and whose characters are searched for in *source-expression*.

Requirement The length for *target-expression* must be greater than zero.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

replacement-expression
specifies any valid expression that evaluates or can be coerced to a character string and that replaces *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The TRANSTRN function replaces or removes all occurrences of a given substring within a character string. The TRANSTRN function does not remove trailing blanks in the *target-expression* string and the *replacement-expression* string. To remove all occurrences of *target*, specify *replacement-expression* as TRIMN(' ').

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this scan, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANSTRN function differs from TRANSLATE in that TRANSTRN scans for substrings and replaces those substrings with a second substring.

Examples

Example 1: Replacing All Occurrences of a Word

In this program, the TRANSTRN function is used to replace **Mrs.** and **Miss** with **Ms.**

```
data _null_;
   dcl char(20) name;
   method run();
      name='Mrs.  Joan Smith';
      name=transtrn(name, 'Mrs.', 'Ms.');
      put name;
      name='Miss Alice Cooper';
      name=transtrn(name, 'Miss', 'Ms.');
      put name;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Ms.  Joan Smith
Ms. Alice Cooper
```

Example 2: Removing Blanks from the Search String

In this program, the TRIM function is used with *target* to exclude blanks. If you did not include the TRIM function, the TRANSTRN function would not replace the source string because the target string contains blanks.

```
data test (overwrite=yes);
   dcl char(10) target;
   dcl char(3) replacement;
   dcl char(8) salelist salelist2;
   method run();
      salelist='CATFISH';
      target='FISH';
      replacement='NIP';
      salelist2=transtrn(salelist, trim(target), replacement);
      put salelist2;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
CATNIP
```

Example 3: Zero Length in the Third Argument of the TRANSTRN Function

The following program illustrates the results of the TRANSTRN function when the third argument, *replacement*, has a length of zero. In DS2, a character constant that consists of two quotation marks with a blank in between them represents a single

blank, and not a zero-length string. In the following program, the results for *string1* are different from the results for *string2*.

```
data _null_;
   dcl char string1 string2;
   method run();
      string1='*' || transtrn('abcxabc', 'abc', trimn(' ')) || '*';
      put string1=;
      string2='*' || transtrn('abcxabc', 'abc', ' ') || '*';
      put string2=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
string1=*x*
string2=* x *
```

See Also

Functions:

- "TRANSLATE Function" in *SAS Functions and CALL Routines: Reference*
- "TRANWRD Function" on page 1146

TRANWRD Function

Replaces all occurrences of a substring in a character string.

Categories: CAS
Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRANWRD(*source-expression*, *target-expression*, *replacement-expression*)

Arguments

source-expression
specifies any valid expression that evaluates or can be coerced to a character string whose characters you want to replace.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

target-expression
specifies any valid expression that evaluates or can be coerced to a character string and that is searched for in *source-expression*.

Requirement The length of the *target-expression* must be greater than zero.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

replacement-expression
specifies any valid expression that evaluates or can be coerced to a character string and that replaces *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See “DS2 Expressions” in *SAS DS2 Programmer’s Guide*

Details

The TRANWRD function replaces all occurrences of a given substring (or a pattern of characters) within a character string. The TRANWRD function does not remove trailing blanks in the *target-expression* string and the *replacement-expression* string.

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANWRD function differs from TRANSLATE in that it scans for substrings (or patterns of characters) and replaces those substrings with a second substring (or pattern of characters).

Examples

Example 1: Replacing All Occurrences of a Word

The following program illustrates replacing all occurrences of a substring using the TRANWRD function:

```
data test(overwrite=yes);
   dcl char(16) text1 text2 nametrn1 nametrn2;
   method run();
      text1='Mrs. Jane Doe';
      text2='Miss Joan Smith';
      nametrn1=tranwrd(text1, 'Mrs.', 'Ms.');
      nametrn2=tranwrd(text2, 'Miss', 'Ms.');
      put nametrn1;
      put nametrn2;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
Ms. Jane Doe
Ms. Joan Smith
```

Example 2: Removing Blanks from the Search String by Changing the Data Type

This program illustrates incorrect data type declarations. The TRANWRD function does not replace the source string because TARGET is declared as `char(10)` and the TRANWRD function searches for the character string `'pail '` and not `''pail''`.

```
data _null_;
   dcl char(100) text finaltext;
   dcl char(10) target;
   dcl varchar(10) rplc;
   method run();
      text='Believe and act as if it were impossible to pail.
          -Charles F. Kettering';
      target='pail';
      rplc='fail';
      finaltext=tranwrd(text, target, rplc);
      put finaltext=;
   end;
enddata;
run;
```

This line is written to the SAS log.

```
finaltext=Believe and act as if it were impossible to pail. -Charles F. Kettering
```

By changing the data type declaration to `VARCHAR(10)`, trailing blanks are excluded from the search:

```
data _null_;
```

```
   dcl char(100) text finaltext;
   dcl varchar(10) target;
   dcl varchar(10) rplc;
   method run();
      text='Believe and act as if it were impossible to pail.
         -Charles F. Kettering';
      target='pail';
      rplc='fail';
      finaltext=tranwrd(text, target, rplc);
      put finaltext=;
   end;
enddata;
run ;
```

This line is written to the SAS log.

```
finaltext=Believe and act as if it were impossible to fail. -Charles F. Kettering
```

Example 3: Using TRIM to Remove Blanks from the Search String

In this program, the TRANWRD function does not replace the source string because the target string contains blanks.

```
data test(overwrite=yes);
   dcl char prod target replacement salelist;
   method run();
      prod='CATFISH';
      target='FISH';
      replacement='NIP';
      prod=tranwrd(prod, target, replacement);
      put prod;
   end;
enddata;
run;
```

The DECLARE statement pads target with blanks to the length of 8, which causes the TRANWRD function to search for the character string 'FISH ' in PROD. Because the search fails, this line is written to the SAS log: CATFISH

You can use the TRIM function to exclude trailing blanks from a target or replacement variable. Use the TRIM function with target:

```
prod=tranwrd(prod, trim(target), replacement;
put prod;
```

SAS writes the following output to the log:

```
CATNIP
```

Example 4: Removing Repeated Commas

You can use the TRANWRD function to remove repeated commas in text and replace the repeated commas with a single comma. In this program, the TRANWRD function is used twice: to replace three commas with one comma and to replace the ending two commas with a period:

```
data test(overwrite=yes);
   dcl char(70) mytxt newtext newtext2;
   method run();
      mytxt='If you exercise your power to vote,,,then your opinion
          will be heard,,';
      newtext=tranwrd(mytxt, ',,,', ',');
      newtext2=tranwrd(newtext, ',,' , '.');
      put mytxt=;
      put newtext=;
      put newtext2=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
mytxt=If you exercise your power to vote,,,then your opinion will be heard,,
newtext=If you exercise your power to vote,then your opinion will be heard,,
newtext2=If you exercise your power to vote,then your opinion will be heard.
```

See Also

Functions:

- “TRANSLATE Function” on page 1141
- “TRANSTRN Function” on page 1143

TRIGAMMA Function

Returns the value of the trigamma function.

Categories: CAS
Mathematical

Returned data type: DOUBLE

Syntax

TRIGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction Nonpositive integers are invalid.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The TRIGAMMA function returns the derivative of the digamma function. For *expression* > 0, the TRIGAMMA function is the second derivative of the lgamma function.

Example

The following program illustrates the TRIGAMMA function:

```
data test(overwrite=yes);
   dcl double x;
   method run();
      x=trigamma(3);
      put x=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x=0.39493406684822
```

See Also

Functions:

- "DIGAMMA Function" on page 520
- "LGAMMA Function" on page 790

TRIM Function

Removes trailing blanks from a character expression, and returns a string with a length of zero if the expression is missing.

Categories: CAS
Character

Alias: TRIMN

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRIM('*expression*')

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The TRIM function copies a character argument, removes trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIM returns a string with a length of zero. TRIM is useful after concatenating because concatenation does not remove trailing blanks.

Note: The TRIM function removes both blanks and whitespace characters as defined by the Unicode standard. Consequently, the TRIM function also handles DBCS blanks and shift out/shift in (SO/SI) escape codes. For more information about the Unicode whitespace character standard, see Wikipedia: Unicode character property.

Examples

Example 1: Removing Trailing Blanks

The following program illustrates the TRIM function:

```
data test(overwrite=yes);
   dcl char(17) string result;
   method run();
      string='Testscore   ';
      result=trim(string)||'File.xls';
      put result;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log:

```
TestscoreFile.xls
```

Example 2: Concatenating a Blank Character Expression

The following program illustrates how to use the TRIM function to concatenate a blank character.

```
data new(overwrite=yes);
   dcl char x z y;
   method run();
      x='A'||trim(' ')||'B';
      z=' ';
      y='>'||trim(z)||'<';
      put x;
      put y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
AB
><
```

See Also

Functions:

- “COMPRESS Function” on page 460
- “LEFT Function” on page 781
- “STRIP Function” on page 1102

TRUNC Function

Truncates a numeric value to a specified length.

Categories:	CAS Truncation
Returned data type:	DOUBLE

Syntax

TRUNC(*expression*, *length-expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

length-expression
specifies any valid expression that evaluates to a numeric value.

Range 3–8

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The TRUNC function truncates a full-length numeric expression (stored as a DOUBLE) to a smaller number of bytes, as specified in *length-expression* and pads the truncated bytes with 0s. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them.

Example

The following program illustrates the TRUNC function:

```
data test(overwrite=yes);
   dcl double i x;
   method run();
      do i=3 to 8;
         x=trunc(3.1,i);
         put x;
      end;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
3.099609375
3.09999847412109
3.09999999403953
3.09999999997671
3.0999999999999
3.1
```

UNIFORM Function

Returns a random variate from a uniform distribution.

Note: In SAS 9.4M5, this function is no longer supported. Use the RAND('UNIFORM') function on page 1004 instead.

UPCASE Function

Converts all letters in an argument to uppercase.

Categories: CAS
Character

Alias: UPPER

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

UPCASE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The UPCASE function copies a character expression, converts all lowercase letters to uppercase letters, and returns the altered value as a result.

Comparisons

The LOWCASE function converts all letters in an argument to lowercase letters. The UPCASE function converts all letters in an argument to uppercase letters.

Example

The following program illustrates the UPCASE function:

```
proc ds2;
data names(overwrite=yes);
   dcl char(13) name;
   method run();
      name='SaraH'; output;
      name='Mylinda'; output;
      name='Ryan'; output;
      name='MaRk T. Smith'; output;
   end;
enddata;
run;
quit;

proc ds2;
data test(overwrite=yes);
   dcl char(13) ucname;
   method run();
      set names;
      ucname=upcase(name);
      put ucname;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
SARAH
MYLINDA
RYAN
MARK T. SMITH
```

See Also

Functions:

- “LOWCASE Function” on page 806

URLDECODE Function

Returns a string that was decoded using the URL escape syntax.

Categories: CAS
Web Tools

Syntax

URLDECODE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

The URL escape syntax is used to hide characters that might otherwise be significant when used in a URL.

A URL escape sequence can be one of the following:

- a plus sign, which is replaced by a blank.
- a sequence of three characters beginning with a percent sign and followed by two hexadecimal characters, which is replaced by a single character that has the specified hexadecimal value.

expression can be decoded using either SAS session encoding or UTF-8 encoding.

Operating Environment Information: In operating environments that use EBCDIC, SAS performs an extra translation step after it recognizes an escape sequence. The specified character is assumed to be an ASCII encoding. SAS uses the transport-to-local translation table to convert this character to an EBCDIC character in operating environments that use EBCDIC. For more information, see the TRANTAB option .

Character Verification

The URLDECODE and URLENCODE functions do not verify that the bytes that are produced by the escape sequences are valid characters based on the encoding.

Length of Returned Variable

If the URLDECODE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

Example

The following program decodes three URL strings.

```
data _null_;
   dcl varchar(15) x1 x2 x3;
   method run();
      x1=urldecode('abc+def');
      x2=urldecode('why%3F');
      x3=urldecode('%41%42%43%23%31');
      put x1= x2= x3=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=abc def x2=why? x3=ABC#1
```

See Also

Functions:

- “URLENCODE Function” on page 1158

URLENCODE Function

Returns a string that was encoded using the URL escape syntax.

Categories: CAS
Web Tools

Syntax

URLENCODE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

expression can be encoded using either SAS session encoding or UTF-8 encoding.

The URLENCODE function encodes characters that might otherwise be significant when used in a URL. This function encodes all characters except for the following:

- all alphanumeric characters
- dollar sign ($)
- hyphen (-)
- underscore (_)
- at sign (@)
- period (.)
- exclamation point (!)
- asterisk (*)
- open parenthesis (( ) and close parenthesis ( ))
- comma (,)

Note: The encoded string might be longer than the original string. Ensure that you consider the additional length when you use this function.

Character Verification

The URLDECODE and URLENCODE functions do not verify that the bytes that are produced by the escape sequences are valid characters based on the encoding.

Example

The following program encodes a URL string.

```
data _null_;
   dcl varchar(7) x1;
   dcl varchar(9) x2;
   method run();
      x1=urlencode('abc def');
      put x1=;
      x2=urlencode('abc def');
      put x2=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
x1=abc%20d
x2=abc%20def
```

See Also

Functions:

- "URLDECODE Function" on page 1157

USS Function

Returns the uncorrected sum of squares.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

USS(*expression* [, ...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.
Data type	DOUBLE
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the USS function:

```
proc ds2;
data values(overwrite=yes);
   dcl double x1 x2 x3 x4;
   method run();
      x1=4;
      x2=2;
      x3=3.5;
      x4=6;
      output;
   end;
enddata;
run;
quit;

proc ds2;
data new (overwrite=yes);
   dcl double val1 val2 val3;
   method run();
      set values;
      val1=uss(of x1-x4);
      put val1;
      val2=uss(of x1-x4, .);
      put val2;
      val3=uss(of val1-val2);
      put val3;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
68.25
68.25
9316.125
```

See Also

Functions:

■ “CSS Function” on page 490

UUIDGEN Function

Returns the short form of a Universally Unique Identifier (UUID).

Categories: CAS
Special

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

UUIDGEN( )

Without Arguments

The UUIDGEN function has no arguments.

Details

The UUIDGEN function returns a UUID (a unique value) for each call. The default result is 36 characters long and it looks like this:

```
5ab6fa40-426b-4375-bb22-2d0291f43319
```

Example

The following program returns a UUID. Note that a variable declaration of 36 characters is required.

```
data _null_;
   dcl char(36) x;
   method run();
      x=uuidgen();
      put x;
   end;
enddata;
run;
```

SAS writes the following output to the log. Each UUID is unique.

```
25C752D5-AFA1-4932-BEE6-39E4006C2AAB
```

See Also

Other References:

- "Universal Unique Identifiers" in *SAS Language Reference: Concepts*

VAR Function

Returns the variance.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

VAR(*expression-1*, *expression-2* [,...*expression-n*])

Arguments

expression
specifies any valid expression that evaluates to a numeric value. The argument list can consist of a variable list.

Requirement: At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Example

The following program illustrates the VAR function:

```
proc ds2;
data new (overwrite=yes);
   vararray double a[4];
   dcl double x1 x2 x3;

   method init();
      a:=(4, 2, 3.5, 6);
   end;
```

```
      method run();
         x1=var(of a1-a4);
         put x1;
         x2=var(of a1, a4, .);
         put x2;
         x3=var(of x1-x2);
         put x3;
      end;
   enddata;
   run;
   quit;
```

SAS writes the following output to the log:

```
2.72916666666666
2
0.26584201388888
```

VERIFY Function

Returns the position of the first character that is unique to an expression.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

VERIFY(*target-expression*, *search–expression*)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string that is to be searched.

Requirement: Literal character strings must be enclosed in single quotation marks.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

search-expression
specifies any valid expression that evaluates or can be coerced to a character string.

Requirement	Literal character strings must be enclosed in single quotation marks.
Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
See	"DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The VERIFY function returns the position of the first character in *target-expression* that is not present in *search-expression*. If there are no characters in *target-expression* that are unique from those in *search-expression*, VERIFY returns a 0.

Comparisons

The INDEX function returns the position of the first occurrence of *search-expression* that is present in *target-expression* where the VERIFY function returns the position of the first character in *target-expression* that does not contain *search-expression*.

Example

The following programs illustrate the VERIFY function:

```
data test(overwrite=yes);
   dcl double z;
   dcl char x y;
   method run();
      x='abc';
      y='abcdef';
      z=verify(y,x);
      put z;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
4
```

```
data test(overwrite=yes);
   dcl double z;
   dcl char x y;
   method run();
      x='abcdef';
      y='abcdef';
      z=verify(y,x);
      put z;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0
```

See Also

Functions:

- "INDEX Function" on page 685

VFORMAT Function

Returns the format that is associated with the specified variable.

Categories: CAS
Variable Information

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

VFORMAT(*variable* | *variable-list*[*i*] | *variable-array*[*i*])

Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

variable-array
specifies a collection of homogenous DS2 variables.

See "Variable Arrays" in *SAS DS2 Programmer's Guide*

i
specifies the element number of the named variable list or variable array.

Details

VFORMAT returns the complete format name, which includes the width and the period (for example, $CHAR20.).

Example

The following program returns the format of a character variable:

```
data _null_;
   dcl varchar(7) str having format $char7.;
   dcl varchar a;
   method run();
      a = vformat(str);
      put a=;
   end;
enddata;
run;
quit;
```

SAS writes the following output to the log:

```
a=$char7.
```

See Also

Functions:

- “VINARRAY Function” on page 1167
- “VINFORMAT Function” on page 1169
- “VLABEL Function” on page 1171
- “VLENGTH Function” on page 1172
- “VNAME Function” on page 1174
- “VTYPE Function” on page 1175

VINARRAY Function

Returns a value that indicates whether the specified variable is a member of an array.

Categories: CAS
Variable Information

Returned data type: INTEGER

Syntax

VINARRAY(*variable* | *variable-list*[*i*] | *variable-array*[*i*])

Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

variable-array
specifies a collection of homogenous DS2 variables.

See "Variable Arrays" in *SAS DS2 Programmer's Guide*

i
specifies the element number of the named variable list or variable array.

Details

VINARRAY returns a value of 1 if the given variable is a member of a variable array. Otherwise, VINARRAY returns a value of 0.

Example

The following program returns a value (0 or 1) depending on whether the specified variable is a member of an array:

```
data _null_;
   dcl  int a b c d;
   vararray int x[3] a b c;
   method init();
      dcl int c_in;
      dcl int d_in;
```

```
        c_in=vinarray(c);  /* c is in array x */
        d_in=vinarray(d);  /* d is not in an array */
        put c_in=;
        put d_in=;
    end;
enddata;
run;
```

SAS writes the following output to the log:

```
c_in=1
d_in=0
```

See Also

Functions:

- “VFORMAT Function” on page 1166
- “VINFORMAT Function” on page 1169
- “VLABEL Function” on page 1171
- “VLENGTH Function” on page 1172
- “VNAME Function” on page 1174
- “VTYPE Function” on page 1175

VINFORMAT Function

Returns the informat that is associated with the specified variable.

Categories: CAS
Variable Information

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

VINFORMAT(*variable* | *variable-list*[*i*] | *variable-array*[*i*])

Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

variable-array
specifies a collection of homogenous DS2 variables.

See "Variable Arrays" in *SAS DS2 Programmer's Guide*

i
specifies the element number of the named variable list or variable array.

Details

VINFORMAT returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

Example

The following program returns an informat that is associated with the specified variable:

```
data _null_;
   dcl varchar(10) str having informat $char5.;
   method run();
      a=vinformat(str);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=$char5.
```

See Also

Functions:

- "VINARRAY Function" on page 1167
- "VFORMAT Function" on page 1166
- "VLABEL Function" on page 1171
- "VLENGTH Function" on page 1172

- "VNAME Function" on page 1174
- "VTYPE Function" on page 1175

VLABEL Function

Returns the label that is associated with the specified variable.

Categories: CAS
Variable Information

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

VLABEL(*variable* | *variable-list*[*i*] | *variable-array*[*i*])

Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

variable-array
specifies a collection of homogenous DS2 variables.

See "Variable Arrays" in *SAS DS2 Programmer's Guide*

i
specifies the element number of the named variable list or variable array.

Details

VLABEL returns the label that is associated with the specified variable. If there is no label, VLABEL returns the variable name.

Example

The following program returns a label for a specified variable:

```
data _null_;
   dcl varchar(10) fname having label 'First Name';
   method run();
      a=vlabel(fname);
      put a=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=First Name
```

See Also

Functions:

- “VINARRAY Function” on page 1167
- “VFORMAT Function” on page 1166
- “VINFORMAT Function” on page 1169
- “VLENGTH Function” on page 1172
- “VNAME Function” on page 1174
- “VTYPE Function” on page 1175

VLENGTH Function

Returns the size of the specified variable.

Categories:	CAS Variable Information
Returned data type:	BIGINT

Syntax

VLENGTH(*variable* | *variable-list*[*i*] | *variable-array*[*i*])

Arguments

variable
specifies a value that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

variable-array
specifies a collection of homogenous DS2 variables.

See "Variable Arrays" in *SAS DS2 Programmer's Guide*

i
specifies the element number of the named variable list or variable array.

Details

The length of numeric data types is defined as the maximum number of digits used by the data type of the column, or the precision of the data. For character types, this is the length in characters of the data; for binary data types, this is the length in bytes of the data. For the TIME, TIMESTAMP, and all interval data types, this is the number of characters in the character representation of this data.

Example

The following program returns the length of the specified variable:

```
data _null_;
  dcl char(7) str;
  dcl double a;
  method run();
     str='World';
     a=vlength (str);
     put a=;
  end;
run;
```

SAS writes the following output to the log:

```
a=7
```

See Also

Functions:

- “VINARRAY Function” on page 1167
- “VFORMAT Function” on page 1166
- “VINFORMAT Function” on page 1169
- “VLABEL Function” on page 1171
- “VNAME Function” on page 1174
- “VTYPE Function” on page 1175

VNAME Function

Returns the name of the specified variable.

Categories: CAS
Variable Information

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

VNAME(*variable* | *variable-list*[*i*] | *variable-array*[*i*])

Arguments

variable
specifies a variable that can be expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See “Variable Lists” in *SAS DS2 Programmer’s Guide*

variable-array
specifies a collection of homogenous DS2 variables.

See “Variable Arrays” in *SAS DS2 Programmer’s Guide*

i
specifies the element number of the named variable list or variable array.

Example

The following program returns the name of the specified variable:

```
data _null_;
   vararray int x[3] a b c;
   dcl char y;
   method run();
      y=vname(x[1]);
      put y=;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
y=a
```

See Also

Functions:

- “VINARRAY Function” on page 1167
- “VFORMAT Function” on page 1166
- “VINFORMAT Function” on page 1169
- “VLABEL Function” on page 1171
- “VLENGTH Function” on page 1172
- “VTYPE Function” on page 1175

VTYPE Function

Returns the full name of the data type that is associated with a variable.

Categories: CAS
Variable Information

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

VTYPE(*variable* | *variable-list*[*i*] | *variable-array*[*i*])

Arguments

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Data type All data types

variable-list
specifies a collection of DS2 variables.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

variable-array
specifies a collection of homogenous DS2 variables.

See "Variable Arrays" in *SAS DS2 Programmer's Guide*

i
specifies the element number of the named variable list or variable array.

Details

VTYPE returns the data type name for the data type of the variable.

Example

The following program returns the name of the data type that is associated with the specified variable.

```
data _null_;
  dcl double d;
  dcl timestamp t;
  dcl nvarchar(10) n;
  method run();
     declare char(20) a b c;
     a=vtype(d);
     put a=;
     b=vtype(t);
     put b=;
     c=vtype(n);
     put c=;
  end;
enddata;
run;
```

SAS writes the following output to the log:

```
a=double
b=timestamp
c=nvarchar
```

See Also

Functions:

- “VINARRAY Function” on page 1167
- “VFORMAT Function” on page 1166
- “VINFORMAT Function” on page 1169
- “VLABEL Function” on page 1171
- “VLENGTH Function” on page 1172
- “VNAME Function” on page 1174

WEEK Function

Returns the week-number value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

WEEK([*sas-date*], ['*descriptor*'])

Arguments

sas-date
specifies the SAS date value. If the *sas-date* argument is not specified, the WEEK function returns the week-number value of the current date.

Data type DOUBLE

descriptor
specifies the value of the descriptor. The following descriptors can be specified in uppercase or lowercase characters.

U
specifies the number-of-the-week within the year. Sunday is considered the first day of the week. The number-of-the-week value is represented as a

decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'u')` is 53. U is the default value.

Tip The U and W descriptors are similar, except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

See "The U Descriptor" on page 1178

V

specifies the number-of-the-week whose value is represented as a decimal number in the range 1–53. Monday is considered the first day of the week and week 1 of the year is the week that includes both January 4 and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year.

See "The V Descriptor" on page 1179

W

specifies the number-of-the-week within the year. Monday is considered the first day of the week. The number-of-the-week value is represented as a decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'w')` is 53.

Tip The U and W descriptors are similar except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

See "The W Descriptor" on page 1179

Default U

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note If you specify a descriptor other than U, V, or W, a note is written to the SAS log to indicate that the argument is invalid. The function returns NULL.

Details

The Basics

The WEEK function reads a SAS date value and returns the week number. The WEEK function is not dependent on locale, and uses only the Gregorian calendar in its computations.

The U Descriptor

The WEEK function with the U descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of

53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year is represented as 05.

Sunday is considered the first day of the week. For example, the value of `week('01jan2013'd, 'u')` is 0.

The V Descriptor

The WEEK function with the V descriptor reads a SAS date value and returns the week number. The number-of-the-week is represented as a decimal number in the range 01–53. The decimal number has a leading zero and a maximum value of 53. Weeks begin on a Monday, and week 1 of the year is the week that includes both January 4 and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. In the following example, 01jan2014 and 31dec2013 occur in the same week. The first day (Monday) of that week is 30dec2013. Therefore, `week('01jan2014'd, 'v')` and `week('30dec2013'd, 'v')` both return a value of 53. This means that both dates occur in week 53 of the year 2013.

The W Descriptor

The WEEK function with the W descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year would be represented as 05.

Monday is considered the first day of the week. Therefore, the value of `week('30dec2013'd, 'w')` is 1.

Comparisons of Descriptors

U is the default descriptor. Its range is 0-53, and the first day of the week is Sunday. The V descriptor has a range of 1-53 and the first day of the week is Monday. The W descriptor has a range of 0-53 and the first day of the week is Monday.

The following list describes the descriptors and an associated week:

- Week 0:

  U indicates the days in the current Gregorian year before week 1.

  V does not apply.

  W indicates the days in the current Gregorian year before week 1.

- Week 1:

  U begins on the first Sunday in a Gregorian year.

  V begins on the Monday between December 29 of the previous Gregorian year and January 4 of the current Gregorian year. The first ISO week can span the previous and current Gregorian years.

  W begins on the first Monday in a Gregorian year.

- End of Year Weeks:

U specifies that the last week (52 or 53) in the year can contain less than 7 days. A Sunday to Saturday period that spans 2 consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

V specifies that the last week (52 or 53) of the ISO year contains 7 days. However, the last week of the ISO year can span the current Gregorian and next Gregorian year.

W specifies that the last week (52 or 53) in the year can contain less than 7 days. A Monday to Sunday period that spans two consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

Example

The following program shows the values of the U, V, and W descriptors for the date March 1, 2019.

```
data _null_;
   dcl double sasdate x y z;
   method run();
      sasdate=to_double(date'2019-03-01');
      x=week(sasdate, 'u');
      y=week(sasdate, 'v');
      z=week(sasdate, 'w');
      put x;
      put y;
      put z;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
8
9
8
```

See Also

Functions:

- “INTNX Function” on page 732

Formats:

- “WEEKDATEw. Format” on page 199
- “WEEKDATXw. Format” on page 201
- “WEEKDAYw. Format” on page 203

WEEKDAY Function

From a SAS date value, returns a whole number that corresponds to the day of the week.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

WEEKDAY(*expression*)

Arguments

expression
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The WEEKDAY function produces a whole number that represents the day of the week, where 1 = Sunday, 2 = Monday, ..., 7 = Saturday.

For information about how DS2 handles date and time values, see "DS2 Expressions" in *SAS DS2 Programmer's Guide*.

Example

The following program illustrates the WEEKDAY function when the current day is Sunday:

```
data test(overwrite=yes);
   dcl double sasdate x;
   method run();
      sasdate=to_double(date'2019-02-24');
      x=weekday(sasdate);
      put x;
   end;
enddata;
```

```
run;
```

SAS writes the following output to the log:

```
1
```

WHICHC Function

Returns the first position of a character string from a list of character strings.

Categories: CAS
Character

Returned data type: DOUBLE

Syntax

WHICHC(*search-expression*, *expression-list-item-1*, *expression-list-item-2* [, ...*expression-list-item-n*])

Arguments

search-expression
specifies any valid expression that evaluates or can be coerced to a character string that is compared with a list of character string expressions.

Requirement: Literal character strings must be enclosed in single quotation marks.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression-list-item
specifies any valid expression that evaluates or can be coerced to a character string and that is a member of a list of character string expressions.

Requirements: Literal character strings must be enclosed in single quotation marks.

At least two expressions are required in the list.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

See: "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The WHICHC function searches the character expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHC returns its position in the expression list. If none of the expressions match the search expression, WHICHC returns a value of 0.

Example

In the following program, 'Spain' appears twice in the list. The WHICHC function returns the first position of 'Spain' in the list:

```
data test(overwrite=yes);
   dcl double position;
   method run();
      position=whichc('Spain', 'Denmark', 'Germany', 'Austria','Spain',
         'China', 'Egypt', 'Spain', 'France');
      put position;
   end;
run;
```

SAS writes the following output to the log:

```
4
```

See Also

Functions:

- "WHICHN Function" on page 1183

WHICHN Function

Returns the first position of a number from a list of numbers.

Categories:	CAS Mathematical
Returned data type:	DOUBLE

Syntax

WHICHN(*search-expression*, *expression-list-item-1*, *expression-list-item-2*

[, ...*expression-list-item-n*])

Arguments

search-expression
specifies any valid expression that evaluates to a number and that is compared with a list of numeric expressions.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

expression-list-item
specifies any valid expression that evaluates to a number and is part of a list.

Requirement At least two expressions are required in the list.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The WHICHN function searches the numeric expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHN returns its position in the expression list. If none of the expressions match the search expression, WHICHN returns a value of 0. Arguments for the WHICHN functions can be any numeric data type.

Example

In the following program, 4.5 appears two times in the list. The WHICHN function returns the first position of 4.5 in the list.

```
data test(overwrite=yes);
   dcl double position;
   method run();
      position=whichn(4.5,7.3, 8.6, 4.5, 4.5, 2.1, 6.4);
      put position;
   end;
run;
```

SAS writes the following output to the log:

```
3
```

See Also

Functions:

- "WHICHC Function" on page 1182

YEAR Function

Returns the year from a SAS date value.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

YEAR(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The YEAR function produces a four-digit numeric value that represents the year.

Example

The following program illustrates the YEAR function when the year is the current year.

```
data test(overwrite=yes);
   dcl double x thdate;
   method run();
      thdate=today();
      x=year(thdate);
      put x;
```

```
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
2019
```

See Also

- "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Functions:

- "DAY Function" on page 505
- "MONTH Function" on page 834

YIELDP Function

Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

YIELDP(*A*, *c*, *n*, *K*, k_0, *p*)

Arguments

A

specifies the face value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal annual coupon rate, expressed as a fraction.

Range $0 \leq c < 1$

Data type DOUBLE

n

specifies the number of coupons per year.

Range $n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons from settlement date to maturity.

Range $K > 0$

Data type DOUBLE

k_o

specifies the time from settlement date to the next coupon as a fraction of the annual basis.

Range $0 < k_0 \leq \frac{1}{n}$

Data type DOUBLE

p

specifies the price with accrued interest.

Range $p > 0$

Data type DOUBLE

Details

The YIELDP function is based on the following relationship:

$$P = \sum_{k=1}^{K} c(k) \frac{1}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

The YIELDP function solves for *y*.

Example

In the following program, the YIELDP function returns the yield-to-maturity of a bond that has a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, and 14 remaining coupons. The time from settlement date to next coupon date is 0.165, and the price with accrued interest is 800.

```
data _null_;
   dcl double y;
   method run();
      y=yieldp(1000,.01,4,14,.165,800);
      put y;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
0.0775031247735
```

YRDIF Function

Returns the difference in years between two dates according to specified day count conventions; returns a person's age.

Categories: CAS
Date and Time

Returned data type: DOUBLE

Syntax

YRDIF(*start-date*, *end-date*[, *basis*])

Arguments

start-date
specifies a SAS date value that identifies the starting date.

Data type DOUBLE

end-date
specifies a SAS date value that identifies the ending date.

Data type DOUBLE

basis

identifies a character constant or variable that describes how SAS calculates a date difference or a person's age. The following character strings are valid:

'30/360'

specifies a 30-day month and a 360-day year in calculating the number of years. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year.

Alias '360'

Tip If either date falls at the end of a month, it is treated as if it were the last day of a 30-day month.

'ACT/ACT'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366.

Alias 'Actual'

'ACT/360'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

'ACT/365'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 365, regardless of the actual number of days in each year.

'AGE'

specifies that a person's age will be computed.

If you do not specify a third argument, AGE becomes the default value for *basis*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Using YRDIF in Financial Applications

The Basics

The YRDIF function can be used in calculating interest for fixed income securities when the third argument, *basis*, is present. YRDIF returns the difference between two dates according to specified day count conventions.

Calculations That Use ACT/ACT Basis

In YRDIF calculations that use the *ACT/ACT* basis, both a 365-day year and 366-day year are taken into account. For example, if n365 equals the number of days between the start and end dates in a 365-day year, and n366 equals the number of days between the start and end dates in a 366-day year, the YRDIF calculation is computed as YRDIF=n365/365.0 + n366/366.0. This calculation corresponds to the commonly understood *ACT/ACT* day count basis that is documented in the financial literature. The values for *basis* also includes *30/360*, *ACT/360*, and *ACT/365*. Each has well-defined meanings that must be conformed to in calculating interest payments for specific financial instruments.

Computing a Person's Age

The YRDIF function can compute a person's age. The first two arguments, *start-date* and *end-date*, are required. If the value of *basis* is AGE, then YRDIF computes the age. The age computation takes into account leap years. No other values for *basis* are valid when computing a person's age.

Examples

Example 1: Calculating a Difference in Years Based on Basis

In the following program, YRDIF returns the difference in years between two dates based on each of the options for *basis*.

```
data test(overwrite=yes);
   dcl double sdate edate y30360 yactact yact360 yact365;
   method run();
      sdate= to_double(date'1998-10-16');
      edate= to_double(date'2010-02-06');
      y30360=yrdif(sdate, edate, '30/360');
      yactact=yrdif(sdate, edate, 'ACT/ACT');
      yact360=yrdif(sdate, edate, 'ACT/360');
      yact365=yrdif(sdate, edate, 'ACT/365');
      put y30360=;
      put yactact=;
      put yact360=;
      put yact365=;
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
y30360=11.3055555555555
yactact=11.3095890410958
yact360=11.475
yact365=11.317808219178
```

Example 2: Calculating a Person's Age

You can calculate a person's age by using three arguments in the YRDIF function. The third argument, *basis*, must have a value of AGE:

```
data _null_;
   dcl double sdate edate age;
   method run();
      sdate= to_double(date'1998-10-16');
      edate= to_double(date'2010-02-16');
      age=yrdif(sdate, edate, 'AGE');
      put age= 'years';
   end;
enddata;
run;
```

SAS writes the following results to the log:

```
age=11.3369863013698 years
```

See Also

Functions:

- "DATDIF Function" on page 497

References

"Day count convention." *Wikipedia*, 2010. Available Day count convention. Accessed on May 1, 2013..

ISDA International Swaps and Derivatives Association, Inc "EMU and Market Conventions: Recent Developments." 1998. ISDA. Available EMU and Market Conventions: Recent Developments.

Mayle, Jan. 1994. *Standard Securities Calculation Methods – Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, New York: Securities Industry Association.

YYQ Function

Returns a SAS date value from year and quarter year values.

Categories:	CAS Date and Time
Returned data type:	DOUBLE

Syntax

YYQ(*year*, *quarter*)

Arguments

year

specifies any valid expression that evaluates to a two-digit or four-digit whole number that represents the year.

Interaction The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

quarter

specifies the quarter of the year (1, 2, 3, or 4).

Data type DOUBLE

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The YYQ function returns a SAS date value that corresponds to the first day of the specified quarter. If either *year* or *quarter* is null or missing, or if the quarter value is not valid, the result is a null or missing value.

Example

The following programs illustrate the YYQ function:

```
data test(overwrite=yes);
   dcl double DateValue Date7Value Date9Value;
   method run();
      DateValue=yyq(2019,3);
      put DateValue;
      put DateValue date7.;
      put DateValue date9.;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
21731
01JUL19
01JUL2019
```

```
data test(overwrite=yes);
   dcl double StartOfQuarter;
   method run();
      StartOfQuarter=yyq(2019,4);
      put StartOfQuarter;
      put StartOfQuarter date9.;
   end;
enddata;
run;
```

SAS writes the following output to the log:

```
21823
01OCT2019
```

See Also

Concepts:

- "Dates and Times in DS2" in *SAS DS2 Programmer's Guide*

Functions:

- "QTR Function" on page 997
- "YEAR Function" on page 1185

8

DS2 Informats

Overview of Informats 1195
General Informat Syntax 1195
How Informats Are Used in DS2 1196
How to Specify Informats in DS2 1197
Validation of DS2 Informats 1197
DS2 Informat Examples 1197

Overview of Informats

An **informat** is an instruction that determines how values are read into a column. For example, the following value contains a dollar sign and commas:

```
$1,000,000
```

To remove the dollar sign ($) and commas (,) before storing the numeric value 1000000 in a column, read this value with the COMMA11. informat.

General Informat Syntax

DS2 informats have the following form:

[$] *informat* [*w*] . [*d*]

Here is an explanation of the syntax:

$
indicates a character informat; its absence indicates a numeric informat.

informat
names the informat. The informat is a SAS informat or a user-defined informat that was previously defined with the INVALUE statement in PROC FORMAT. For more information about user-defined informats, see PROC FORMAT in the *Base SAS Procedures Guide*.

w
specifies the informat width, which for most informats is the number of columns in the input data.

d
specifies an optional decimal scaling factor in the numeric informats. SAS divides the input data by 10 to the power of *d*.

Note: Even though SAS can read up to 32 decimal places when you specify some numeric informats, floating-point numbers with more than 15 decimal places might lose precision due to the limitations of the eight-byte floating-point representation used by most computers.

Informats always contain a period (.) as a part of the name. If you omit the *w* and the *d* values from the informat, SAS uses default values. If the data contains decimal points, SAS ignores the *d* value and reads the number of decimal places that are actually in the input data.

For more information about how informats work and a complete list of informats, see the *SAS Formats and Informats: Reference*.

How Informats Are Used in DS2

DS2 supports SAS informats as follows.

- Both informats supplied by SAS and user-defined informats can be associated with a column. For information about how to create your own informat in SAS, see PROC FORMAT in the *Base SAS Procedures Guide*.

  Note: To create and access user-defined informats, a SAS session must be available in order to access the SAS catalog file that stores the SAS informat definitions.

- Only the SAS data set and SPD data sets support storing and retrieving an informat with a column.
- Informats can be associated with all data types, but all data types are converted to either CHAR or DOUBLE.
- You can associate SAS informats with a column by using the HAVING clause of the DS2 DECLARE statement. For more information, see "How to Specify Informats in DS2" on page 1197.

For more information and a complete list of informats supplied by SAS, see the section on informats in the *SAS Formats and Informats: Reference*.

How to Specify Informats in DS2

In DS2, specify informats as an attribute in the HAVING clause of the DECLARE statement. For example, in the following statement, the column *y* is declared with the IEEE8.2 format and the BITS5.2 informat.

```
dcl double y having format ieee8.2 informat bits5.2;
```

Note: In DS2, an informat for a column cannot be changed or removed.

Validation of DS2 Informats

Informats are not validated by a data source or applied to a column until execution time.

When metadata for a column is requested, the informat name is returned without validation.

DS2 Informat Examples

```
dcl char(10) y having label 'varchar' format $5. informat
$charzb4.3;

dcl double ssn having format best 10.4 informat comma10.4;

dcl double(6) salary having informat uscurrency.;

dcl char(12) site having informat $city.;
```

9

DS2 Operators

Dictionary 1199
NEW Operator 1199

Dictionary

NEW Operator

Constructs an instance of a package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

See: The _NEW_ operator for predefined DS2 packages is documented in the reference section for each package.

Syntax

package-variable=**_NEW_** *package-name* ([*constructor-arguments*]);

Required Arguments

package-variable
specifies a name that can reference an instance of the package.

package-name
specifies the name of the package.

Requirement *package-name* must be a DS2 predefined package type or created with a PACKAGE statement before the _NEW_ operator is executed.

Optional Argument

constructor-arguments
specifies any constructor arguments that are passed to the constructor of the package instance.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

After you have stored methods and variables in a package by using the PACKAGE statement, you can access them by declaring and instantiating the package. If you use the _NEW_ operator to instantiate the package, you must first use the DECLARE PACKAGE statement to declare the package variable.

declare package *package-name variable-name*;

variable-name = _new_ *package-name*( );

For example, in the following lines of code, the DECLARE PACKAGE statement tells SAS that C is a variable of type COMPLEX package. The _NEW_ operator constructs an instance of the COMPLEX package and assigns it to the package variable C.

```
declare package complex c;
c = _new_ complex();
```

If you want to initialize package data, you can use a constructor. A constructor is a method that is used to instantiate a package and to initialize the package data. For example, you can provide initialization data by using parameters in the constructor syntax for the hash and hash iterator package

```
declare package hash h();
h = _new_ hash(0, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

Note: You can use the DECLARE PACKAGE statement with constructor arguments to declare and instantiate a package in one step. The example shown above would look like this.

```
declare package hash h(0, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

For more information, see "DS2 Packages" in *SAS DS2 Programmer's Guide* and "DECLARE PACKAGE Statement" on page 1231.

Comparisons

You can use the DECLARE PACKAGE statement and the _NEW_ operator, or the DECLARE PACKAGE statement alone to declare and instantiate an instance of a package.

Example

See "User-Defined Packages" in *SAS DS2 Programmer's Guide* and "Predefined DS2 Packages" in *SAS DS2 Programmer's Guide*.

See Also

- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*
- "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: FCMP Package" on page 1500
- "_NEW_ Operator: Hash Package" on page 1551
- "_NEW_ Operator: Hash Iterator Package" on page 1556
- "_NEW_ Operator: Logger Package" on page 1672
- "_NEW_ Operator: Matrix Package" on page 1734
- "_NEW_ Operator: PCRXFIND Package" on page 1768
- "_NEW_ Operator: PCRXREPLACE Package" on page 1776
- "_NEW_ Operator: SQLSTMT Package" on page 1819

Statements:

- "DECLARE PACKAGE Statement" on page 1231

10

DS2 Statements

Overview of Statements ... **1204**

Block Statements ... **1205**
- Overview of Block Statements ... 1205
- Program Block Statements ... 1205
- Program Subblock Statements ... 1205

Global Declaration Statements ... **1207**

Local Statements ... **1208**

Including Comments in a DS2 Program ... **1209**

Dictionary ... **1210**
- ARRAY Assignment Statement ... 1210
- Assignment Statement ... 1211
- BY Statement ... 1214
- CONTINUE Statement ... 1219
- DATA Statement ... 1220
- DECLARE Statement ... 1224
- DECLARE PACKAGE Statement ... 1231
- DECLARE THREAD Statement ... 1234
- DO Statement ... 1236
- DROP Statement ... 1243
- DROP PACKAGE Statement ... 1245
- DROP THREAD Statement ... 1246
- DS2_OPTIONS Statement ... 1247
- ENDDATA Statement ... 1250
- ENDPACKAGE Statement ... 1251
- ENDTHREAD Statement ... 1252
- FORWARD Statement ... 1252
- GOTO Statement ... 1253
- IF Statement: Subsetting ... 1255
- IF-THEN/ELSE Statement ... 1256
- KEEP Statement ... 1258
- Labeled Statement ... 1260
- LEAVE Statement ... 1262
- MERGE Statement ... 1264
- METHOD Statement ... 1270
- Null Statement ... 1279

OUTPUT Statement 1280
PACKAGE Statement 1286
PUT Statement 1291
RENAME Statement 1296
RETAIN Statement 1297
RETURN Statement 1300
SELECT Statement 1301
SET Statement 1305
SET FROM Statement 1310
STOP Statement 1316
Sum Statement 1316
THREAD Statement 1318
VARARRAY Statement 1323
VARLIST Statement 1329

Overview of Statements

A DS2 statement is a series of items that can include keywords, identifiers, special characters, and operators. A DS2 statement can perform the following actions:

- perform variable assignments
- influence program control
- perform input and output of data
- create methods
- call methods and functions
- specify or change the behavior of a DS2 program

All DS2 statements end with a semicolon.

Here are the statement categories:

Behavior
: use to specify or change the behavior of a DS2 program.

Block
: use to create a programming blocks. For more information, see "Programming Blocks" in *SAS DS2 Programmer's Guide*.

Block control
: use to modify program behavior in a programming block.

Global
: use anywhere in the global section of a programming block created by a DATA, PACKAGE, or THREAD statement.

Local
: use only inside a DS2 method.

Note that some statements belong to multiple categories. For example, the DECLARE statement can be both a global and a local statement.

Block Statements

Overview of Block Statements

There are five DS2 statements that differ from other statements in that they are used to group other statements in programming blocks. The block statements are as follows.

- DO...END
- METHOD...END
- PACKAGE...ENDPACKAGE
- DATA...ENDDATA
- THREAD...ENDTHREAD

Block statements can be divided further into two categories: statements that create program blocks and statements that create program subblocks. Program block statements include PACKAGE...ENDPACKGE, DATA...ENDDATA, and THREAD...ENDTHREAD. Program subblock statements include METHOD...END and DO...END.

In this documentation, these terms are used for programming blocks.

- A data programming block or **data program** refers to code bounded by DATA... ENDDATA statements.
- A package programming block or **package** refers to the stored library of variables and methods bounded by PACKAGE...ENDPACKAGE statements. The variables and methods of a package can be used by DS2 programs, threads, or other packages.
- A thread programming block, or **thread program**, refers to a stored program bounded by THREAD...ENDTHREAD statements. The thread program can be called by the SET FROM statement in a DS2 program or package.
- A DO programming block, or **DO loop**, refers to a subblock of programming statements bounded by DO...END statements.
- A method programming block or **method block** refers to a subblock of programming statements bounded by METHOD...END statements.

Each data program must have one and only one program block statement. A data program can and often has multiple subblock statements. For more information about programming blocks, see “Programming Blocks” in *SAS DS2 Programmer’s Guide*.

Program Block Statements

Program blocks are created with the DATA, PACKAGE, and THREAD statements. These statements, and their concluding END statements, are used outside of any other statement, and the program blocks that they define can contain other statements. All other statements must be used inside one of these blocks.

A data program , a package, or a thread program contains two sections: a section of global declarations followed by a section of METHOD statements. This is an example of a data program.

```
data t;
   ...global declarations
   ...METHOD statement;
enddata;
```

All global declarations must precede the first METHOD statement in the program. A syntax error results if a global declaration is found after a METHOD statement in a program.

The DATA statement creates a data program. A data program consists of the global declaration list and the METHOD statement list contained within a program created by the DATA...ENDDATA statements. For information about compiling and executing a data program, see "DS2 Procedure" in *Base SAS Procedures Guide*. For more information about the DATA statement, see the "DATA Statement" on page 1220.

A thread program is created by the THREAD...ENDTHREAD statements. A thread program consists of the global declaration list and the METHOD statement list contained within the THREAD...ENDTHREAD statements. The structure of a thread program is essentially the same as that of a data program, but is used to execute several threads in parallel. When you use a DECLARE statement in another data program or package to reference the thread, the thread program is loaded into memory. You can then use the SET FROM statement in a subsequent data program to run the program in one or more operating system threads. For more information, see the "THREAD Statement" on page 1318.

The package is defined by the PACKAGE...ENDPACKAGE statements. A package is a collection of variables and methods that can be called by a data program, a thread program, or another package. A package consists of the global declaration list and the METHOD statement list contained within a programming block created by the PACKAGE...ENDPACKAGE statements. A package is compiled and stored for later use. When you declare the package in a DS2 program, a thread program or another package, the stored package is loaded into memory. You can then access the methods and variables in the package. For more information, see the "PACKAGE Statement" on page 1286.

Program Subblock Statements

Program subblock statements are created with the METHOD or DO statements. A DS2 program normally contains several subblocks of programming statements.

Each subblock contains two sections: a section of global declaration statements followed by a section of other local statements. This is an example of a METHOD subblock.

```
method m;
   ...global declarations
   ...local statements;
end;
```

All global declaration statements must precede all other local statements in the subblock. A syntax error will result if a global declaration statement is placed after any other type of statement in a programming subblock.

Global Declaration Statements

Global declaration statements are statements that must be in the declaration section of a program created by a DATA, PACKAGE, or THREAD statement. Global declaration statements generally provide information for your DS2 program or request information or data. Generally, global declaration statements are not executable; they take effect as soon as DS2 compiles program statements.

The following table lists DS2 statements that are allowed in the declaration section of a DS2 program.

Note: The DECLARE statement can be used both globally and within a method.

- DECLARE
- DROP
- FORWARD
- KEEP
- RENAME
- RETAIN
- VARARRAY
- VARLIST

Local Statements

Local statements are statements that you can use inside a programming block created with a METHOD statement.

The following table lists DS2 method statements.

Note: All global declaration statements must proceed all other local statements in a method programming block.

Note: A METHOD statement is not a local statement. Therefore, a METHOD statement cannot be nested inside another METHOD statement.

- Assignment
- BY
- CONTINUE
- DECLARE
- DECLARE PACKAGE
- DECLARE THREAD
- DO
- GOTO
- IF, Subsetting
- IF-THEN/ELSE
- Labels
- LEAVE
- MERGE
- Null
- OUTPUT
- PUT
- RETURN
- SELECT
- SET
- SET FROM
- STOP
- Sum

Including Comments in a DS2 Program

You might want to include comments in a DS2 program to document the purpose of the program.

The DS2 language supports the following syntax for including comments in DS2 code:

```
/*comment*/
```

A DS2 comment has the following characteristics:

- It can be added anywhere in the DS2 program where a single blank space can appear, except in character constants or delimited identifiers.
- It can be any length.
- It can be embedded in DS2 statements and DS2 expressions.
- It can contain internal white space, newlines, semicolons, and quotation marks, including unmatched quotation marks.
- It is supported in all environments in which DS2 can be run.
- It cannot be nested.

The SAS macro comment syntax %*comment; is not supported by the DS2 compiler. A SAS macro comment can be embedded in DS2 programs that are submitted in the DS2 procedure. The macro facility strips SAS macro comments from the program code before the code is passed to the DS2 procedure for further processing. A SAS macro comment used in a DS2 program that is executed outside of the SAS programming runtime environment results in a syntax error and halts compilation.

You can use /.../ and *... in DS2 code for comments.

This example comment provides comments regarding a Test Library using /*...:

```
/*-------------------------------------------------------------
 *
 *                SAS  TEST  LIBRARY
 *
 *      NAME: ds2mod.sas
 *   SUPPORT: saswss
 *     TITLE: MOD function
 *      KEYS: mod function
 *   SYSTEMS: all
```

```
*  COMMENTS: This test is self-validating. A message is printed

*            in the log if an incorrect value is obtained.

-----------------------------------------------------------/
```

This example comment provides details on the eps variable: using *...

```
data _null_;
    dcl double x;
    dcl double y;  /* DS2 comments are preferred in PROC DS2; they
required in other programming environments */
    method init();
        /* multiline comment with
           an embedded semicolon ;
           an embedded single quotation mark '
           an embedded double quotation mark " */
        if (missing(x) or /* multiline comment embedded
                             in a DS2 expression */
            missing(y)) then do;
            put /* comment embedded in a DS2 statement */ 'Hello,
World';
        end;
    end;
enddata;
run;
```

Dictionary

ARRAY Assignment Statement

Assigns either a temporary array or a constant list to a temporary array.

Syntax

array-name:= *array-name*;

array-name:=(*constant-list*);

Arguments

array-name
specifies the name of the array.

constant-list
specifies a list that define the array elements.

Details

You can assign either a temporary array or a constant list to a temporary array.

When you assign one array to another array, the data types of the two arrays must be compatible (either the same or convertible). The number of dimensions do not have to be the same for the two arrays, and the total number of elements in each array do not have to be the same.

Example

The following statements are examples of array assignments.

```
ar1:=('sales', 'inv', 'profit');

ar2:=(3*3.14159, 2*'5', 2*(1,2), 99);

ar7:=ar2;
```

See Also

"Array Assignment" in *SAS DS2 Programmer's Guide*

Assignment Statement

Evaluates an expression and stores the result in a variable.

Category: Local

Syntax

variable=*expression*;

Arguments

variable
names a new or existing variable.

Range *variable* can be a variable name, array reference, or SUBSTR function.

Tip Variables that are created by the Assignment statement are not automatically retained.

expression
is any valid DS2 expression.

Tip *expression* can contain the variable that is used on the left side of the equal sign. When a variable appears on both sides of a statement, the original value on the right side is used to evaluate the expression, and the result is stored in the variable on the left side of the equal sign.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The Basics

Assignment statements evaluate the expression on the right side of the equal sign and store the result in the variable that is specified on the left side of the equal sign.

The following type conversions take place with the Assignment statement:

- If the variable has a data type and the expression's data type does not match and can be converted, the expression is converted to the variable's data type. If the expression cannot be converted, an error occurs.
- If the variable does not have a data type, then one of the following actions occurs:
  - If the expression's value is not null and has a data type of CHAR, BINARY, DATE, or TIME, then the variable is given the data type of the expression.
  - If the expression's value is not null and of numeric type, then the variable is given a data type of DOUBLE. If the expression's value is null, then the variable is given a data type of DOUBLE.
  - If the expression's type is CHAR or BINARY, then the variable is given the length of the expression's value. If the expression's value cannot be determined at compile time (for example, for VARCHAR strings), the variable is given the default length of 200.
  - If the expression's type is TIME or TIMESTAMP, then the variable is given the expression's precision.
- If an assignment statement is `a = b = 5`, then `b = 5` is an expression. If `b` is a value other than 5, then `b = 5` is evaluated to 0. Therefore, `a` is assigned a value of 0. The first equal sign (=) is an assignment operator and the second equal sign is a logical equality operator.For more information, see "Example 2: Using an Expression with Multiple Equal Signs" on page 1213.

Note: DS2 supports using `eq` as well as the equal sign. For example, `x = y < z < w;` is equivalent to `x = y < z & z < w;`. Another example is that `a = b = c =d;` equates to `a = ((b = c) & (c = d));`.

Examples

Example 1: Different Types of Expressions

These assignment statements use different types of expressions.

- ```
  name='Nagasaki';
  ```
- ```
  FullName='Mr. '||name;
  ```
- ```
  price=price+markup;
  ```
- ```
  declare int i;
  declare double d;
  declare character(200) c;

     i = 5;
     d = 1.2345;
     d = d + i;
     c = 'abc';
     c = d;
     c = '123' || '456';
     i = c;
  ```

Example 2: Using an Expression with Multiple Equal Signs

The result of these assignment statements is **a**=0. The values of b, c, and d are not changed.

```
data;
   dcl double a b c d;

   method init();
      a = b = c = d = 5;
      put _all_;
   end;
enddata;
run;
```

Example 3: Using a Long Literal Enclosed in Brackets

The following example uses nested brackets to assign a long literal string to a VARCHAR variable, s.

```
data _null_;
  method init();
      dcl varchar(100) s;
      s = [===[line 1
               line 2
               line 3 contains a double square bracket [[
               line 4]===];
      put s;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
line 1
line 2
line 3 contains a double square bracket [[
line 4
```

See Also

- "DS2 Constants" in *SAS DS2 Programmer's Guide*
- "DS2 Type Conversions for Expression Operands" in *SAS DS2 Programmer's Guide*

BY Statement

Controls the operation of a MERGE or SET statement in a DS2 program and sets up special grouping variables.

Category: Local

Restriction: The BY statement must immediately follow a MERGE or SET statement. The BY statement is optional when using a SET statement.

Tip: Trailing blanks are always ignored when combining tables with a SET or MERGE statement.

Syntax

BY [DESCENDING] *column*... [DESCENDING] *column*;

Arguments

DESCENDING
specifies that the tables are sorted in descending order by the variable that is specified. DESCENDING means largest to smallest numerically, or reverse alphabetical for character variables.

Restriction You cannot use the DESCENDING option with tables that are indexed because indexes are always stored in ascending order.

column(s)
names each column by which the table is sorted. These columns are referred to as BY variables.

Tip The table can be sorted by more than one column.

Details

How DS2 Indicates the Beginning and End of a BY Group

DS2 indicates the beginning and end of a BY group by creating two temporary variables for each BY variable: FIRST.*variable* and LAST.*variable*. The value of these variables is either 0 or 1. DS2 sets the value of FIRST.*variable* to 1 when it reads the first row in a BY group, and sets the value of LAST.*variable* to 1 when it reads the last row in a BY group. These temporary variables are available for DS2 programming but are not added to the result set.

For a complete explanation of how SAS processes grouped data and of how to prepare your data, see "Combining Tables" in *SAS DS2 Programmer's Guide*.

In a Data Program

The BY statement applies only to the SET or MERGE statement that precedes it in a data program, and only one BY statement can accompany each of these statements in a data program. An error occurs if the BY statement appears anywhere else in the data program.

Note: The BY statement recognizes the linguistic collation of data that is sorted by using the SORT procedure with the SORTSEQ=LINGUISTIC option.

Note: Assume you have a table with a column that has a character data type. If you change the column to be a numeric data type on input with a DECLARE statement, the sort order of the resulting column is not numeric. For example, assume the following character column (CHAR) s exists in a table: 9, 10, 500. If you declare s as a numeric column (DOUBLE) when you read the table with a SET and BY statement, the data is generated as output in alphanumeric order, that is, 10, 500, 9. The SET statement orders the rows in alphanumeric order before the string is converted to the numeric data type.

For more information, see "Combining Tables" in *SAS DS2 Programmer's Guide*.

Processing BY Groups

SAS assigns the following values to FIRST.*variable* and LAST.*variable*:

- FIRST.*variable* has a value of 1 under the following conditions:
  - when the current row is the first row that is read from the table.
  - when the value of the current row BY variable differs from the value of that BY variable in the previous row.
  - FIRST.*variable* has a value of 1 for any preceding variable in the BY statement.

    In all other cases, FIRST.*variable* has a value of 0.
- LAST.*variable* has a value of 1 under the following conditions:
  - when the current row is the last row that is read from the table.

- when the value of the current row BY variable differs from the value of that BY variable in the next row.
- LAST.*variable* has a value of 1 for any preceding variable in the BY statement.

  In all other cases, LAST.*variable* has a value of 0.

Examples

Example 1: Specifying One or More BY Variables

- Observations are in ascending order of the variable DEPT:

```
by dept;
```

- Observations are in alphabetical (ascending) order by CITY and, within each value of CITY, in ascending order by ZIPCODE:

```
by city zipcode;
```

Example 2: Specifying Sort Order

- Observations are in ascending order of SALESREP and, within each SALESREP value, in descending order of the values of JANSALES:

```
by salesrep descending jansales;
```

- Observations are in descending order of BEDROOMS, and, within each value of BEDROOMS, in descending order of PRICE:

```
by descending bedrooms descending price;
```

Example 3: Using a BY Statement When Combining Tables with a SET Statement

The following example creates two tables and uses a SET statement to combine the tables using the **common** column.

```
data mrg01a(overwrite=yes);
   dcl varchar(10) common animal;
   method init();
      common='a'; animal='Ant'; output;
      common='b'; animal='Bird'; output;
      common='c'; animal='Cat'; output;
      common='d'; animal='Dog';  output;
      common='e'; animal='Eagle'; output;
      common='f'; animal='Frog';  output;
   end;
enddata;
run;


data mrg01b(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Apple'; output;
```

```
      common='b'; plant='Banana'; output;
      common='c'; plant='Coconut'; output;
      common='d'; plant='Dewberry';  output;
      common='e'; plant='Eggplant'; output;
      common='f'; plant='Fig';  output;
        common='g'; plant='Grapefruit'; output;
   end;
enddata;
run;


/*  set concatenates */
data;
   method run();
      set mrg01a mrg01b; by common;
   end;
 enddata;
run;
```

The following table is generated.

common	animal	plant
a	Ant	
a		Apple
b	Bird	
b		Banana
c	Cat	
c		Coconut
d	Dog	
d		Dewberry
e	Eagle	
e		Eggplant
f	Frog	
f		Fig
g		Grapefruit

Example 4: Using a BY Statement When Combining Tables with a MERGE Statement

The following example creates two tables and uses a MERGE statement to combine the tables using the **common** column.

```
data mrg01a(overwrite=yes);
   dcl char(10) common animal;
   method init();
      common='a'; animal='Ant'; output;
```

```
      common='b'; animal='Bird'; output;
      common='c'; animal='Cat'; output;
      common='d'; animal='Dog';  output;
      common='e'; animal='Eagle'; output;
      common='f'; animal='Frog';  output;
   end;
enddata;
run;


data mrg01b(overwrite=yes);
   dcl char(10) common plant;
   method init();
      common='a'; plant='Apple'; output;
      common='b'; plant='Banana'; output;
      common='c'; plant='Coconut'; output;
      common='d'; plant='Dewberry';  output;
      common='e'; plant='Eggplant'; output;
      common='f'; plant='Fig';  output;
      common='g'; plant='Grapefruit'; output;
   end;
enddata;
run;


/*  match merge */
data;
   method run();
      merge mrg01a mrg01b; by common;
   end;
 enddata;
run;
```

The following table is generated.

common	animal	plant
a	Ant	Apple
b	Bird	Banana
c	Cat	Coconut
d	Dog	Dewberry
e	Eagle	Eggplant
f	Frog	Fig
g		Grapefruit

See Also

- "Combining Tables" in *SAS DS2 Programmer's Guide*

Statements:

- “MERGE Statement” on page 1264
- “SET Statement” on page 1305

CONTINUE Statement

Stops processing the current DO loop iteration and resumes with the next iteration.

Category: Local

Syntax

CONTINUE;

Without Arguments

The CONTINUE statement has no arguments. It resumes processing statements with the next iteration of the DO loop.

Details

The CONTINUE statement can appear only in the statement list of an iterative DO loop (for example, DO *i*=, DO WHILE, or DO UNTIL).

Comparisons

- The CONTINUE statement stops the processing of the current iteration of a DO statement and resumes program execution with the next iteration of the current DO statement.
- The LEAVE statement stops the processing of the current DO statement and resumes program execution outside of the current DO statement.

Example

This example illustrates the use of the CONTINUE statement. The DO loop iterates and prints the incremented value of `ctr`. When `ctr` is equal to 3, the CONTINUE statement causes execution to jump to the next iteration of the DO loop, and prevents `ctr` from printing.

```
data _null_;
```

```
      dcl int ctr;
      method init();
         do ctr = 1 to 5;
            if ctr = 3 then continue;
            put ctr;
         end;
      end;
   enddata;
```

The following lines are written to the SAS log.

```
1
2
4
5
```

See Also

Statements:

- “DO Statement” on page 1236
- “LEAVE Statement” on page 1262

DATA Statement

Begins a DS2 program and provides names for any output tables.

Category: Block

Alias: TABLE

Syntax

DATA [<table-expression>] [... <table-expression>] ;

... *program-body* ...

[ENDDATA ;]

<table-expression>::=

table (*table-options*)

| **_ROWSET_** (*table-options*)

| **_NULL_**

Without Arguments

If you do not specify any table names with the DATA statement, then the DS2 program returns table rows to the client application and no tables are created.

Arguments

table

specifies the name of the table. *table* can be one of these forms.

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*
- *table-name*
- *caslib.table-name*

catalog	is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.
schema	is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.
table-name	is the name of the table.
caslib.table-name	is the name of the table on the CAS server.

Notes If the table name has a dot in it and you are accessing a CAS table, you must enclose the table name in double quotation marks. Here is an example:

```
data mycaslib "tdlibref.foo";
```

If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in `data mytable;`, the table name is stored as "MYTABLE" and in `data "MyTable";`, the table name is stores as "MyTable". This is important if table and schema names in your data source are case-sensitive.

CAUTION **Using the PRESERVE_TAB_NAMES=no option on your LIBNAME statement can cause unexpected results.**

ROWSET
specifies that the DATA statement should not create a table, but it should instead return table rows to the client application.

NULL
specifies that the DATA statement should not create a table or return rows to the client application.

table-options
specifies optional arguments that the DS2 program applies when it writes rows to the output table. For more information about table options, see "DS2 Table Options" on page 1347.

Tip _NULL_ can be useful in debugging programs when using PUT statements.

Details

A DS2 program begins with the DATA statement and ends with the ENDDATA statement.

A DS2 program processes input data and produces output data. A DS2 program can run in two different ways: as a program and as a thread. When a DS2 program runs as a program, here are the results:

- Input data can include both rows from database tables and rows from DS2 program threads.
- Output data can be either database tables or rows that are returned to the client application.

When a DS2 program runs as a thread, here are the results:

- Input data can include only rows from database tables, not other threads.
- Output data includes the rows that are returned to the DS2 program that started the thread.

If you specify no table names in the DATA statement, or you specify the keyword _ROWSET_ , then the DS2 program returns table rows to the client application and no tables are created. If you specify no table names in the DATA statement, at least one global variable is required.

No rows are ever written to the _NULL_ table name. Therefore, if_NULL_ is the only table name present in the DATA statement, the DS2 program does not return any rows.

If any other table names are present, then the program creates a table for each *table*, and table rows are written to those tables. For more information, see the "OUTPUT Statement" on page 1280.

A warning is issued for tables with delimited column names that are submitted to data sources that are not case sensitive. Data sources that are not case-sensitive remove the quotation marks and treat the column name as not delimited.

Note: The MongoDB data source has special requirements for creating tables. Depending on your data, you might want to create these tables: a root table, a parent table, and a child table. You can create only root tables with DS2. To create parent and child tables, you must use FedSQL. See the information about MongoDB in *SAS/ACCESS for Relational Databases: Reference*.

Comparisons

For a comparison between packages, DS2 programs, and threads, see "Block Statements" on page 1205.

Examples

Example 1: Creating an Output Table

Use the DATA statement to create one or more output tables. You can use table options to customize the output table. The following DS2 program creates two output tables, EXAMPLE1 and EXAMPLE2 . It uses the table option DROP to prevent the column IDNumber from being written to the EXAMPLE2 table.

```
data example1 example2 (drop=(IDnumber));
   set sample;
   . . .more statements. . .
enddata;
```

Example 2: When Not Creating a Table

Usually, the DATA statement specifies at least one table name to create an output table. Using the keyword _NULL_ as the table name causes the DS2 program to execute without writing rows to a table. This example writes to the log the value of NAME and ID for each row. An output table is not created.

```
data _null_;
   set sample;
   put Name ID;
enddata;
```

See Also

Statements:

- "OUTPUT Statement" on page 1280
- "SET Statement" on page 1305

DECLARE Statement

Declares one or more DS2 variables or temporary arrays.

Categories: Global
Local

Note: Square brackets in the syntax convention indicate optional arguments. The escape character (\ ) before a square bracket indicates that the square bracket is required in the syntax. Array bounds must be contained by square brackets ([ ]).

Syntax

DECLARE [PRIVATE] { <data-type> <variable-list> [<having-clause>] } ;

< data-type >::=

<exact-numeric-type> | <approximate-numeric-type> | <binary-string-type> | <string-type>
| <date-type>

<exact-numeric-type> ::=

{**INT** | **BIGINT** | **SMALLINT** | **TINYINT**
| **DECIMAL** [(*precision* [,*scale*])] | **NUMERIC** [(*precision* [,*scale*])] }

<approximate-numeric-type> ::=

{ **DOUBLE** | **DOUBLE PRECISION** | **FLOAT** | **REAL** }

<binary-string-type>::=

BINARY(*length*) | **VARBINARY**(*length*)

<string-type>::=

NCHAR [(*character-length*)]

| **NVARCHAR** [(*character-length*)]

| **CHAR** [(*character-length*)] [CHARACTER SET *character-set-identifier*]

| **VARCHAR** [(*character-length*)] [CHARACTER SET *character-set-identifier*]

<date-type>::=

{ **TIME** | **TIMESTAMP** } [(*precision*)] | **DATE**

<variable-list>::=

{ *variable* [...*variable*] }

| { *variable* <array-declaration> [*variable*...<array-declaration>] }

<array-declaration>::=

\[<array-bound> [, ...< array-bound>] \]

<array-bound>::=

{ [*dim-lower*:] *dim-upper*} | { [*dim-lower*:] {DIM (*a*[, *n*]) | *}

<having-clause>::=

HAVING <having-option> [... <having-option>]

<having-option>::=

LABEL '*string*' | n'*string*'

| **FORMAT** *format*

| **INFORMAT** *format*

Arguments

PRIVATE

specifies variables that can be accessed only from within the package.

See "Attributes and Methods" in *SAS DS2 Programmer's Guide*

INT | BIGINT | SMALLINT | TINYINT

specifies an integer variable or array.

Alias INTEGER for INT

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

| DECIMAL[(*precision* [, *scale*])] | NUMERIC[(*precision* [, *scale*])]

specifies an exact numeric variable or array.

precision

specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point

Note Not all data sources can support a precision of 52 digits.

scale

specifies the maximum number of decimal digits that can be stored to the right of the decimal point

Range 0–*precision*

Note *scale* is less than or equal to *precision*.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

DOUBLE | DOUBLE PRECISION | FLOAT | REAL

specifies a floating-point variable or array.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

BINARY (*length*)

specifies a binary variable or array.

Requirement If you specify BINARY, you must also specify the *length* of the variable or array in bytes.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

VARBINARY (*length*)
specifies a varying-length binary variable or array.

Alias BINARY VARYING

Requirement If you specify VARBINARY, you must also specify the *length* of the binary variable or array in bytes.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

NCHAR | NVARCHAR | CHAR | VARCHAR
specifies a character variable or array.

Aliases NATIONAL CHARACTER, NATIONAL CHAR for NCHAR

NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING for NVARCHAR

CHARACTER for CHAR

CHARACTER VARYING for VARCHAR

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

character-length
specifies the maximum number of characters that the string can hold for NCHAR, NVARCHAR, CHAR, and VARCHAR data types.

Default 8

Tip The number of bytes that character variables declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character (for example, UTF-8), then CHAR and NCHAR are identical types and both use NCHAR.

CHARACTER SET *character-set-identifier*
specifies character set encoding information for CHAR and VARCHAR data types.

Default Default encoding depends on your operating system and locale.

Tip You can use a character string literal or a simple string for character set names. For example, you can specify `"ibm-866"` or `'ibm-866'`.

See For a complete list of character set encoding values, see "Encoding Values in SAS Language Elements" in the *SAS National Language Support (NLS): Reference Guide*.

TIME
specifies a time variable or array.

TIMESTAMP
specifies both a date and time variable or array.

precision
specifies the precision for a TIME or TIMESTAMP data type.

Default 6

Note If you are working with TIME and TIMESTAMP values in a data source other than SAS and you do not specify a precision, the default precision will always be the DS2 default precision of 6.

DATE
specifies a date variable or array.

variable
specifies the scalar variable or array name. You can specify one or more variables or arrays. However, *variable* can only be of the type specified in *data-type*. You can mix scalar and array variables of the same type.

dim-lower* and *dim-upper
specifies a positive or negative integer used to define the number and size of the array boundary.

Tip If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

See "Temporary Array Variable Declaration" on page 1229

DIM(*a*[, *n*])
specifies that the size of the upper bounds of the array is determined by the number of elements in a dimension of a previously declared array by using a DIM function call.

a
specifies the name of a previously declared array.

n
specifies the dimension, in a multidimensional array, for which you want to know the number of elements.

Tip If no *n* value is specified, the DIM function returns the number of elements in the first dimension of the array.

Restriction The DIM function is the only function that you can use to specify an upper array bounds. The DIM function cannot be used to specify the lower bound of a dimension.

See "DIM Function" on page 521

LABEL '*string*' | n'*string*'
assigns a descriptive label to the variable or array. The label can be a CHAR literal (*string*) or NCHAR literal (n*string*).

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

FORMAT *format*
Associates any valid DS2 format with the variable or array.

See "DS2 Formats" on page 57

INFORMAT *informat*
Associates any valid SAS informat with the variable or array.

See Chapter 8, "DS2 Informats," on page 1195

Details

Overview of the DECLARE Statement

The DECLARE statement can be used to specify scalar variables and temporary arrays. More than one variable and array can be specified in a DECLARE statement. For example, the following DECLARE statement specifies two scalar variables named **x** and **y** and two temporary arrays named **a** and **b**.

```
declare double a[10] x y b[20];
```

A DECLARE statement associates a data type with each variable in a variable list or an array. In the previous example, **x**, **y**, **a**, and **b** have a data type of DOUBLE.

In DS2, the DECLARE statement is also used for package and thread declarations. For more information about package and thread declaration, see the "DECLARE PACKAGE Statement" on page 1231 and the "DECLARE THREAD Statement" on page 1234.

By default, you receive a warning for any variable that is not declared. If you use a variable without declaring it, DS2 assigns the variable a data type (implicit declaration). The data type for an undeclared variable on the left side of an assignment statement is determined by the data type of the value on the right side of the assignment statement. However, you can use the DS2SCOND system option or the SCOND option on the DS2 procedure to control how DS2 handles an undeclared variable. You can use these options to require the declaration of all table columns and variables. If you specify DS2SCOND=ERROR or SCOND=ERROR, you must use a DECLARE statement for each column or variable. Declaration by assignment does not occur. For more information, see the "DS2SCOND= System Option" on page 1345, "DS2 Procedure" in *Base SAS Procedures Guide* , and "Variable Declaration" in *SAS DS2 Programmer's Guide*.

Note: Types must be an exact match between a data program and a thread program. For the decimal data type, both the precision and scale must match. For character strings, the column size, the character set, and whether the string is of varying length or fixed length, must all match.

Scalar Variable Declarations

Scalar declarations can be used for numeric, character, date, or time data types. You can specify the maximum number of characters a string can contain. Here is an example.

```
declare char(200) s;
```

DS2 imposes no limit on this number, but, in practice, there might be some restriction due to machine limitation. The default length or precision for a particular data type depends on the data source.

For fixed-length character variables, the maximum length is used as the initial (and only) allocation for the string memory. For varying character strings, memory is allocated on an as-needed basis up to the maximum length. There might be an execution-time advantage to using varying character variables because they do not require blank-padding after an operation as fixed-length character variables do.

The number of bytes that character variables declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character (for example, UTF-8), then CHAR and NCHAR are identical types and both use NCHAR.

An error occurs if a variable is declared more than once in the same scope, and the declarations are not identical.

If you use a variable without declaring it, it receives the type of the value assigned to it. If no value is assigned, DS2 assigns the variable as type DOUBLE and assigns the value as a missing or null value.

Temporary Array Variable Declaration

You use the DECLARE statement to create a temporary array. The elements of a temporary array are temporary in that they are not located in the PDV and therefore do not appear in the result table.

Array declarations are similar to scalar declarations. In addition to the data type and name you also specify the number and size of the array bounds.

Array bounds are given as a signed integer pair, [*l*:*h*], where *l* represents the lowest index for the given bound and *h* represents the highest index for the given bound. An error is returned if *h*<*l*. If you specify an array bound with only one integer, then that integer is interpreted as the highest index. The default lowest index is 1.

This example declares an array *a* of type double, with five elements indexed from 1 to 5.

```
declare double a[5];
```

Multiple bounds (or dimensions) are specified using comma separators. This example declares a two dimensional character array *b* with 5 elements in the first dimension and 10 elements in the second dimension for a total of 50 elements in the array.

```
declare char b[5,10];
```

This example declares an array *c* with two elements. The array is indexed with a lower bound of 0 and an upper bound of 1.

```
declare int c[0:1];
```

Temporary arrays exist only for the duration of the DS2 program. For more information, see “DS2 Arrays” in *SAS DS2 Programmer’s Guide* and “Temporary Arrays” in *SAS DS2 Programmer’s Guide*.

HAVING Clause

You can associate label, format, and informat attributes with one or more scalar variables or an array. The HAVING clause functions the same as the FORMAT, INFORMAT, and LABEL statements in Base SAS. However, in DS2, the attributes must be specified in the declaration statement of the variable or array.

For more information about how DS2 handles formats and informats, see "Using Formats in DS2" on page 60 and "How Informats Are Used in DS2" on page 1196.

For more information about arrays and the HAVING clause, see "Declaring Arrays with a HAVING Clause" in *SAS DS2 Programmer's Guide*.

Note: If variables are declared with a HAVING clause in a thread program and the variables are redeclared in a data program with a HAVING clause, the HAVING clause in the data program is used instead of the HAVING clause in the thread program. If there is no HAVING clause in the DECLARE statement in the data program, the HAVING clause in the thread program is not used.

Examples

Example 1: Declaring Variables

The following examples illustrate the DECLARE statement.

- `declare bigint b2 b3;`
- `declare double d;`
- `declare char(200) c1 c2;`
- `declare varchar vc;`
- `declare nchar(100) wc;`
- `declare time(4) tm;`
- `declare date dt;`
- `declare varbinary(10) b;`
- `dcl double x having label 'Amount' format ieee8.2;`
- `dcl char(10) y having label 'varchar' format $quote.;`
- `dcl private double x;`

Example 2: Declaring Temporary Arrays

The following examples illustrate the DECLARE statement for temporary arrays.

- `declare double darr[-5:4];`
- `declare char carr[1:2, 0:3];`
- `declare int iarr[10] having format octal7.;`

Example 3: Temporary Array Dimensions

The following table contains examples of statements that specify temporary arrays and the dimensions of those arrays.

Statement	Number of Dimensions	Range of Each Dimension	Number of Elements
`declare double a[100];`	1	1:100	100
`declare double a[10, 20, 30];`	3	1:10 1:20 1:30	10x20x30 = 6000
`declare double a[5:10];`	1	5:10	6
`declare double a[-3:3, 5, 7:9, 10];`	4	-3:3 1:5 7:9 1:10	7x5x3x10 = 1050
`declare double a[DIM(u)];`	1	1:DIM(u)	DIM(u)
`declare double a[DIM(u,1), 0:DIM(u,2)];`	2	1:DIM(u,1) 0:DIM(u,2)	DIM(u,1)x(DIM(u,2)+1)

See Also

- "Variable Declaration" in *SAS DS2 Programmer's Guide*
- "DS2 Arrays" in *SAS DS2 Programmer's Guide*
- "Temporary Arrays" in *SAS DS2 Programmer's Guide*
- "Declaring Arrays with a HAVING Clause" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement" on page 1231
- "DECLARE THREAD Statement" on page 1234
- "VARARRAY Statement" on page 1323

System Options:

- "DS2SCOND= System Option" on page 1345

DECLARE PACKAGE Statement

Creates a package variable and gives you the option to create an instance of the package.

Category: Local

See: The DECLARE PACKAGE statement for the predefined DS2 packages is documented in the reference section for each package.

Syntax

DECLARE PACKAGE *package* [(*table-options*)] *variable* [(*constructor-arguments*)] [...*variable* [(*constructor-arguments*)]];

Arguments

package
specifies the package name. *package* can be one of these forms.

- *catalog.schema.package*
- *schema.package*
- *catalog.package*
- *package*

catalog
is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema
is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

package
is the name of the package.

Requirements The package name must match the name of a package created in a PACKAGE statement of be a predefined DS2 package, or an error will occur.

Package naming conventions are based on the data source. For more information, see the documentation for your data source.

See “PACKAGE Statement” on page 1286

table-options
specifies optional arguments that the DS2 program applies when it creates a package. For more information about table options, see “DS2 Table Options” on page 1347.

variable
specifies a name that can reference an instance of the package.

constructor-arguments
specifies any constructor arguments that are passed to the constructor of the package instance.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

An instance of a package can be constructed either with the _NEW_ operator or with the DECLARE PACKAGE statement. The DECLARE PACKAGE statement with constructor arguments creates a package variable and constructs a package instance:

```
declare package complex c();
```

The above statement is equivalent to the following two statements:

```
declare package complex c; 1
 c = _new_ complex(); 2
```

1 Creates a package COMPLEX variable c that is a null package reference.

2 Constructs a package COMPLEX instance and sets package variable c to reference the constructed package instance.

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "User-Defined Packages" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator" on page 1199

Statements:

- "DECLARE PACKAGE Statement: FCMP Package" on page 1495
- "DECLARE PACKAGE Statement: Hash Package" on page 1514

- "DECLARE PACKAGE Statement: Hash Iterator Package" on page 1523
- "DECLARE PACKAGE Statement: Logger Package" on page 1665
- "DECLARE PACKAGE Statement: Matrix Package" on page 1703
- "DECLARE PACKAGE Statement: SQLSTMT Package" on page 1783
- "PACKAGE Statement" on page 1286

DECLARE THREAD Statement

Creates an instance of a thread.

Category: Local

Syntax

DECLARE THREAD *thread* [(*table-options*)] *instance*(*argument*) [... *instance* (*argument*)];

Arguments

thread
specifies the thread name. *thread* can be one of these forms.

- *catalog.schema.thread*
- *schema.thread*
- *catalog.thread*
- *thread*

catalog
is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema
is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

thread
is the name of the thread.

Requirements The thread name must match the name of a thread created in a THREAD statement, or an error occurs.

Thread naming conventions are based on the data source. For more information, see the documentation for your data source.

See "Overview of Threaded Processing" in *SAS DS2 Programmer's Guide*

table-options
specifies optional arguments that the DS2 program applies when it creates a thread. For more information about table options, see "DS2 Table Options" on page 1347.

instance
specifies a name that identifies an instance of the thread.

argument
specifies arguments used with *instance*.

Details

When a thread is declared, the variable representing the thread can be considered an instance of the thread. This means that two different thread variables represent two completely separate copies of a thread.

Thread variables can appear only in global scope, otherwise an error is returned. If the thread named in the DECLARE statement does not exist, an error is returned.

In this example, the thread `work.t` is instantiated and named `thread_name`:

```
declare thread work.t thread_name;
```

Once an instance of a thread has been created, you can use it in a SET FROM statement. For more information, see the "THREAD Statement" on page 1318. For more information about threads, see "Overview of Threaded Processing" in *SAS DS2 Programmer's Guide*.

Note: The SAS In-Database Code Accelerator enables you to publish a thread program to the database and execute that thread program in parallel inside the database. For more information about using the SAS In-Database Code Accelerator, see *SAS In-Database Products: User's Guide*. The SAS In-Database Code Accelerator is not supported in SAS Viya.

See Also

- "Threaded Processing" in *SAS DS2 Programmer's Guide*

Statements:

- "SET FROM Statement" on page 1310
- "THREAD Statement" on page 1318

DO Statement

Specifies a group of statements to be executed as a unit.

Category: Local

Syntax

DO [numeric-data-type][*index-variable* = <index-variable-clause>] [<conditional-clause>]
[, ...[<index-variable-clause>] [<conditional-clause>]] ;

...*statement-list*...

END [*end-label*] ;

< numeric-data-type >::=

<exact-numeric-type> | <approximate-numeric-type>

<exact-numeric-type> ::=

{**INT** | **BIGINT** | **SMALLINT** | **TINYINT**
| **DECIMAL** [(*precision* [,*scale*])] | **NUMERIC** [(*precision* [,*scale*])] }

<approximate-numeric-type> ::=

{ **DOUBLE** | **DOUBLE PRECISION** | **FLOAT** | **REAL** }

<index-variable-clause>::=

start [TO *stop* [BY *increment*]]

<conditional-clause>::=

WHILE (*expression*) | **UNTIL** (*expression*)

Without Arguments

The DO statement without the index variable argument and clauses is the simplest form of DO group processing. The statements between the DO and END statements are called a **DO group**. In a simple DO loop, statements in the DO group are executed one time only. You can nest DO statements within DO groups. A simple DO statement is often used within IF-THEN/ELSE statements to designate a group of statements to be executed depending on whether the IF condition is true or false.

Arguments

INT | BIGINT | SMALLINT | TINYINT
specifies an integer variable or array.

Alias INTEGER for INT

Restriction This argument is available only for SAS 9.4M6 and SAS Viya 3.4.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

DECIMAL[(*precision* [, *scale*])] | NUMERIC[(*precision* [, *scale*])]
specifies an exact numeric variable or array.

precision
specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point

Note Not all data sources can support a precision of 52 digits.

scale
specifies the maximum number of decimal digits that can be stored to the right of the decimal point

Range 0–*precision*

Note *scale* is less than or equal to *precision*.

Restriction This argument is available only for SAS 9.4M6 and SAS Viya 3.4.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

DOUBLE | DOUBLE PRECISION | FLOAT | REAL
specifies a floating-point variable or array.

Restriction This argument is available only for SAS 9.4M6 and SAS Viya 3.4.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

index-variable
names a variable that is used as an index counter for the loop.

CAUTION
Avoid changing the index variable within the DO group. If you modify the index variable within the iterative DO group, you might cause infinite looping.

Requirement The variable must resolve to a numeric value.

Tips If the variable is not declared as a local variable, it will end up in the table that is being created unless it is explicitly dropped. For more information about local variables, see "Scope of DS2 Identifiers " in *SAS DS2 Programmer's Guide*.

The index variable can be a THIS expression. For more information, see "THIS Expression" on page 1942.

statement-list
specifies any valid DS2 statements.

end-label

The END statement closes the DO loop. The optional *end-label* argument specifies an identifier. This label, created by using the Labels statement, must match the label immediately preceding the DO statement, or an error will occur. For more information, see the "Labeled Statement" on page 1260.

Clauses

< index-variable-clause >

specifies a numeric scalar expression or series of expressions that determines the number of times that the DO group will be executed.

start

specifies the initial value of the index variable. *start* can be any expression that resolves to a numeric value.

When it is used without TO *stop*, the value of *start* can be a series of items expressed in this form:

item-1 <, ...item-n>

The items can be a number or an expression that yields a number. The DO group is executed once for each value in the list. If a WHILE condition is added, it applies only to the item that it immediately follows.

Requirement When it is used with TO *stop*, *start* must be a number or an expression that yields a number.

Notes The DO group is executed first with *index-variable* equal to *start*. The value of *start* is evaluated before the first execution of the loop.

If *index-variable* is an integral type and *start* is floating-point type, the value of *start* is converted to INTEGER type with possible loss of precision.

TO *stop*

specifies the ending value of the index variable. *stop* can be any expression that resolves to a numeric value.

Notes Execution continues based on the value of *increment* until one of the following conditions is met: the value of *index-variable* passes the value of *stop*, until a WHILE or UNTIL clause that is specified in the DO statement is satisfied, or until a statement in the DO group directs execution out of the loop. The value of *stop* is evaluated before the first execution of the loop.

If *index-variable* is an integral type and *stop* is floating-point type, the value of *stop* is converted to INTEGER type with possible loss of precision. Any change to *stop* made within the DO group does not affect the number of iterations.

BY *increment*
specifies a positive or negative value that controls the incrementing or decrementing of *index-variable*. *increment* can be any expression that resolves to a numeric value.

Notes The value of *increment* is evaluated before the execution of the loop. When *increment* is positive, *start* must be the lower bound and *stop* must be the upper bound of the loop. If *increment* is negative, *start* must be the upper bound and *stop* must be the lower bound of the loop. If no increment is specified, the index variable is incremented by 1.

If *index-variable* is an integral type and *increment* is floating-point type, the value of *increment* is converted to INTEGER type with possible loss of precision. Any change to the increment made within the DO group does not affect the number of iterations.

<conditional-clause>
specifies a clause that returns true or false.

WHILE (*expression*)
causes DO group statements to execute repetitively while a condition is true.

Note A WHILE clause is evaluated before each execution of the loop, so that the statements inside the group are executed repetitively while the expression is true. If the expression is false the first time it is evaluated, the DO loop does not iterate even once.

See Appendix 2, “DS2 Expressions,” on page 1931

UNTIL (*expression*)
causes DO group statements to execute repetitively until a condition is true.

Note An UNTIL clause is evaluated after each execution of the loop, so that the statements inside the group are executed repetitively until the expression is true. The DO loop always iterates at least once.

See Appendix 2, “DS2 Expressions,” on page 1931

Details

The DO statement allows a group of statements to be executed as a unit. If iterative or conditional clauses are specified, this group of statements can be executed multiple times.

DO loop iteration with an integral index variable is performed using INTEGER arithmetic. Otherwise, DO loop iteration is performed using DOUBLE arithmetic. Because the representation of the DOUBLE type is a binary number, the accumulation of an imprecise number can introduce enough error to prevent execution of the DO loop the expected number of times. For example, this loop might not execute the expected number of times.

```
do i = 0.001 to 10 by 0.001;
```

For more information, see "Numerical Accuracy in SAS Software" in *SAS Language Reference: Concepts*.

A DO statement defines a scoping block so that any variables declared in the DO statement have scope local to the scope of the DO statement.

There are three forms of the DO statement:

- The DO statement without clauses is the simplest form of DO-group processing. In this form, a group of statements is executed as a unit, usually as a part of IF-THEN/ELSE statements.
- The iterative DO statement can execute statements between DO and END statements repetitively, based on the values of an index variable.
- The DO statement can execute statements in a DO loop repetitively while a conditional clause is true, checking the condition before each iteration of the DO loop. The DO UNTIL form evaluates the condition at the bottom of the loop; the DO WHILE form evaluates the condition at the top of the loop.

  The final value of the loop is, at most, the specified TO value. For example, `do i = 1 to 10` executes the loop 10 times. This is different from other languages (for example, C, where the last iteration is less than the TO value).

The CONTINUE statement can be used within the DO group to cause execution to immediately continue with the next iteration of the DO statement.

The LEAVE statement can be used within the DO statement to transfer execution to either the statement immediately following a specified target DO statement or the current DO statement.

In SAS 9.4M6 and SAS Viya 3.4, you can declare the loop counter within the DO statement. Here is an example.

```
do double i=0 to 10;
```

The scope of the variable declaration is local to the loop iteration.

Note: Only numeric data types are allowed.

Examples

Example 1: DO Statement with No Clauses

```
do;
   dcl int i j;
      i = 2;
      j = i + 5
end;
```

Example 2: DO with an Index Variable Clause

- `do j = 1 to 10 by 2;`

```
      i + j;
   end;
```

- ```
  do k = 11 to 0 by -3;
     i + k;
  end;
  ```

- ```
  x = -2;
  y = -1;
  do k = 11 to 0 by x + y;
     dcl int i;
     if k < 5 then i = k;
     else
     i = k - 1;
  end;
  ```

- ```
  dcl double i;
  do i = 0 to 5 by 0.5;
     put i=;
     /* the values output are */
     /* 0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5   */
  end;
  ```

Example 3: DO with an Index Variable Clause Containing a Series

```
dcl int i;
do i = 0 to 10 by 3, 6, 7, 8;
      put i=;
      /*  the values output are*/
      /*  0, 3, 6, 9, 6, 7, 8        */
end;
```

Example 4: DO Statement with a WHILE Clause

```
k = 1;
do while(k <= 5);
   k = k + 1;
end;
```

Example 5: DO Statement with Both an Index Variable and a WHILE Clause

```
j = -2;
do k = 1 to 10 by 2 while (j <= 0);
 j = j + 1;
end;
```

Example 6: DO Statement with an UNTIL Clause

```
k = 1;
do until(k > 6);
   k = k + 1;
end;
```
```

Example 7: DO Statement with Inline Declaration

This example declares the counter i as a DOUBLE.

```
data _null_;
    method init();
       do double i = 1.1 to 3.1;
          put i=;
       end;
    end;
enddata;
run;
```

The following lines are written to the SAS log.

```
1.1
2.1
3.1
```

Example 8: DO Statement with Inline Declaration Counting Backward

This example declares the counter i as a DOUBLE.

```
data _null_;
   method init();
      do double i = 1 to '-3' by -1;
         put i=;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
1
0
-1
-2
-3
```

Example 9: DO Statement with Inline Declaration and Non-Numeric Argument

Non-numeric values are assigned to the DOUBLE loop index in the following example. As a result, missing values are written to the log.

```
data _null_;
   method init();
      do double i = 'abc', 123, 'def', 'ghi';
         put i=;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
.
123
.
.
```

See Also

Statements:

- “CONTINUE Statement” on page 1219
- “Labeled Statement” on page 1260
- “LEAVE Statement” on page 1262

DROP Statement

Excludes columns from output tables.

Category: Global

Note: This statement cannot be used within a method.

Syntax

DROP *column-list* | *vararray*;

Arguments

column-list
specifies the name of one or more columns to omit from the output tables.

Restriction Numbered range lists in the format *col1–coln* and name prefix lists in the format *col:* are not supported.

vararray
specifies the name of a variable array.

See “VARARRAY Statement” on page 1323

Details

The DROP statement applies to all the tables that are created within the same DS2 program and can appear only in the global statements section of a data, thread, or package program. The columns in the DROP statement are available for processing

in the DS2 program. If no DROP or KEEP statement appears, all tables that are created in the DS2 program contain all columns. Do not use both DROP and KEEP statements within the same DS2 program.

Comparisons

- The DROP statement applies to all output tables that are named in the DATA statement. To exclude columns from some tables but not from others, use the DROP= table option in the DATA statement.
- The KEEP statement is a parallel statement that specifies a list of columns to write to output tables. Use the KEEP statement instead of the DROP statement if the number of columns to include is significantly smaller than the number to omit.
- The KEEP and DROP statements select columns to include in or exclude from output tables. The subsetting IF statement selects rows.

Example

- These examples show the correct syntax for listing columns with the DROP statement:
  - `drop time shift batchnum;`
  - `drop grade1 grade2 grade3 grade4;`
- In this example, the columns PURCHASE and REPAIR are used in processing but are not written to the output table INVENTRY:

```
data inventry;
   drop purchase repair;
   method run();
      set table-specification;
      totcost=sum(purchase,repair);
   end;
enddata;
```

- In this example, the first three variables in variable array x are dropped from the output.

```
/* drop x1, x2, x3 */
proc ds2;
data mytest;
   vararray double x[10];
   drop x1-x3;
   method init();
     do i = 1 to 10;
       x[i] = i;
     end;
   output;
   end;
enddata;
```

```
run;
quit;
```

See Also

Statements:

- “KEEP Statement” on page 1258

Table Options:

- “DROP= Table Option” on page 1413

DROP PACKAGE Statement

Deletes a DS2 package.

Category: Block Control

Restriction: This statement must be used outside of any other programming block. Programming blocks are delimited by the DATA...ENDDATA, THREAD...ENDTHREAD, or PACKAGE...ENDPACKAGE statements.

Syntax

DROP PACKAGE *package* [(*table-options*)];
RUN;

Arguments

package
specifies the name of the package to be deleted.

table-options
specifies optional arguments that the DS2 program applies when it deletes a package. For more information about table options, see “DS2 Table Options” on page 1347.

Details

The DROP PACKAGE statement drops, or deletes, the table that contains the code for the specified DS2 package. The table must have been previously created with a PACKAGE statement.

Note: The RUN statement is required after the DROP PACKAGE statement.

Example

These examples show the syntax for dropping a DS2 package:

```
drop package mypkg;
run;
drop package mypkg (pw='n1234');
run;
```

See Also

Statements:

- "PACKAGE Statement" on page 1286

DROP THREAD Statement

Deletes a DS2 thread program.

Category: Block Control

Restriction: This statement must be used outside of any other programming block. Programming blocks are delimited by the DATA...ENDDATA, THREAD...ENDTHREAD, or PACKAGE...ENDPACKAGE statements.

Syntax

DROP THREAD *thread* [(*table-options*)];

Arguments

thread
specifies the name of the thread to be deleted.

table-options
specifies optional arguments that the DS2 program applies when it deletes a thread. For more information about table options, see "DS2 Table Options" on page 1347.

Details

The DROP THREAD statement drops, or deletes, the table that contains the code for the specified DS2 thread. The table must have been previously created with a THREAD statement.

Example

These examples show the syntax for dropping a DS2 program thread:

```
drop thread mythread;
drop thread mythread (pw='n1234');
```

See Also

Statements:

- "THREAD Statement" on page 1318

DS2_OPTIONS Statement

Specifies or changes the default behavior of a DS2 program.

Category: Behavior

Requirement: The DS2_OPTIONS statement must appear at the top level of the DS2 program and applies only to the next DATA, PACKAGE, or THREAD statement.

Syntax

DS2_OPTIONS *options*;

Arguments

options
specifies one or more DS2 options. *options* can be one of the following values:

DIVBYZERO=ERROR | IGNORE
specifies how DS2 processes a division by zero operation.

ERROR
halts row processing and writes an error to the SAS log.

IGNORE
writes a missing or null value to the result set. No message is written to the SAS log.

Default ERROR

MISSING_NOTE
writes a note to the SAS log when an invalid function argument generates a missing value.

Default An error message is written to the SAS log when an invalid function argument generates a missing value.

SAS
specifies that nonexistent values are processed as SAS missing values. This option overrides the ANSIMODE system option or DS2 procedure option.

Default By default, DS2 processes nonexistent values as SAS missing values.

Notes You can also specify ANSIMODE in the PROC DS2 statement. For more information, see "PROC DS2 Statement" in *Base SAS Procedures Guide*.

For more information, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.

SCOND
specifies the level of messages that is displayed in the SAS log for the DS2 variable declaration strict mode, which requires that every variable must be declared in the DS2 program. For more information about the DS2 variable declaration strict mode, see "Variable Declaration" in *SAS DS2 Programmer's Guide*.

WARNING
writes warning messages to the SAS log.

NONE
no messages are written to the SAS log.

NOTE
writes notes to the SAS log.

ERROR
writes error messages to the SAS log.

Default The default is determined by the DS2SCOND= system option. The default for DS2SCOND= is WARNING. For more information, see "DS2SCOND= System Option" on page 1345.

Note You can also specify SCOND in the PROC DS2 statement. For more information, see "PROC DS2 Statement" in *Base SAS Procedures Guide*.

REPORTLINE=YES | NO

specifies whether line numbers are reported in log messages that are generated for DS2 programs that run in CAS.

The log line numbers are included in ERROR, WARNING, and NOTE messages that are generated during the execution of the DS2 program and indicate the location of the DS2 statement or user-defined package that was being processed when the message was generated.

YES

causes line numbers for execution-time messages to be displayed in the SAS log.

NO

suppresses line numbers for most execution-time messages, resulting in reduced CPU time for compiling and executing the DS2 program.

Default YES

Restriction The log line numbers are reported in CAS messages only.

Notes This option is available in August 2021 as a SAS Viya 3.5 hotfix. This hotfix also adds log line numbers to compilation messages for DS2 programs that run in CAS and don't already report a line number. However, the line numbers in the program compilation messages cannot be suppressed.

Line numbers that are reported for stored packages and threads are the line number that are offset from the start of the package or thread block rather than SAS log line numbers.

The REPORTLINE= option controls messages that are generated by DS2 services. It has no effect on messages that are generated during statement preprocessing and postprocessing, or by SAS/ACCESS services.

TRACE

provides a trace of what statements are executed.

Restrictions This option is not supported on the CAS server.

This option is valid only when using SAS In-Database Code Accelerator.

Requirement You must have Write permission to the current directory.

Note Because this option creates a trace of every statement that is executed, there could be a significant performance impact.

TRACEVARIABLES

enables tracing of variable value changes during DS2 program execution.

Restriction This option is valid only in SAS Viya 3.5 when using the CAS server or the SAS Micro Analytic Service server.

Requirement: The DS2 App.TableServices.DS2.Runtime.TraceVariables logger must also be configured and set to TRACE or DEBUG. For more information, see "App.TableServices.DS2.Runtime.TraceVariables" in *SAS DS2 Programmer's Guide* for a list of the statements and functions for which variable changes are traced.

Interaction: This option enables tracing for the program block directly below the DS2_OPTIONS statement. To enable variable tracing of more than one program block, you must specify a DS2_OPTIONS statement before each program block that you want to trace.

Example: "Example: Logging Trace Variables" in *SAS DS2 Programmer's Guide*

TYPEWARN
prints a warning to the SAS log when an implicit type conversion occurs.

Example

Here are some examples:

```
ds2_options typewarn trace;
ds2_options scond=error;
ds2_options divbyzero=ignore;
ds2_options reportline=no
```

ENDDATA Statement

Marks the end of a DATA statement.

Category: Block

Alias: ENDTABLE

Syntax

ENDDATA;

Details

A DS2 program can have multiple package subprograms followed by an optional data program. The following restrictions apply:

- There can be only one data program and the data program must be the last subprogram.
- The ENDPACKAGE, ENDTHREAD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

See Also

Statements:

- “DATA Statement” on page 1220

ENDPACKAGE Statement

Marks the end of a PACKAGE statement.

Category: Block

Syntax

ENDPACKAGE;

Details

A DS2 program can have multiple subprograms followed by an optional data program. The following restrictions apply:

- There can be only one data program and the data program must be the last subprogram.
- The ENDPACKAGE, ENDTHREAD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

See Also

Statements:

- “PACKAGE Statement” on page 1286

ENDTHREAD Statement

Marks the end of a THREAD statement.

Category: Block

Syntax

ENDTHREAD;

Details

A DS2 program can have multiple thread subprograms followed by an optional data program. The following restrictions apply:

- There can be only one data program and the data program must be the last subprogram.
- The ENDPACKAGE, ENDTHREAD, or ENDDATA statements are optional for the last subprogram of the DS2 program. These statements are required for all other subprograms.

See Also

Statements:

- “THREAD Statement” on page 1318

FORWARD Statement

Indicates that the method definition follows the method expression.

Category: Global

Syntax

FORWARD *method* [...*method*];

Arguments

method
specifies the name of the method to be defined.

Details

When a method definition appears after any method expression that refers to it, a FORWARD statement for the method must be declared before the method expression. Otherwise, the DS2 compiler cannot determine whether the method expression refers to a method.

Example

- In this example, the D method is called inside the RUN method. Because the D method is defined after it is called, a FORWARD statement must be specified before the D method is called.

```
forward d;
method run();
   d = d();
   d = d(100);
end;
method d() returns double;
   return 99;
end;
method d(int y) returns int;
   return 100 + y;
end;
```

- This example creates a user-defined method, SIN, that masks the system function SIN. The user method calls the system function SIN.

```
forward sin;
method run()
   d = sin(3.14159/2);
   put 'd= ' d;
end;
method sin(double x) returns double;
   return system.sin(x);
end;
```

GOTO Statement

Transfers execution immediately to a labeled statement.

Category: Local

Syntax

GOTO *label*;

Arguments

label
specifies a statement label that identifies the GOTO destination.

Details

The destination label for the GOTO statement must be within the same DS2 method. You must specify the *label* argument or an error occurs. Statement labels are defined by using the Labels statement.

Comparisons

GOTO statements can often be replaced by DO-END and IF-THEN/ELSE programming logic.

Example

In this example, when x = 2, program execution transfers to the DONE label.

```
method run();
   x = 1;
   do;
      if x=2 then goto done;
      put x;
      x+1;
   end;
   done:
   put x;
end;
```

See Also

Statements:

- “DO Statement” on page 1236
- “Labeled Statement” on page 1260
- “LEAVE Statement” on page 1262

IF Statement: Subsetting

Continues processing only those rows that meet the condition.

Category: Local

Syntax

IF *expression*;

Arguments

expression
is any valid expression that evaluates to true or false.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

Details

The expression in a subsetting IF statement is evaluated to produce a result that is either a nonzero value or zero. A nonzero value causes the expression to be true; a result of zero causes the expression to be false.

The subsetting IF statement is equivalent to this IF-THEN statement:

```
if not (expression)
   then return;
```

If *expression* is true, DS2 continues to execute statements in the program.

Note: In logical operations, any expression, such as a subsetting IF statement, a missing value evaluates to False in SAS mode; a null value evaluates to neither true nor false in ANSI mode.

Note: A nonretained variable from the input table is reset to missing or null if it is used before the SET or MERGE statement in the RUN method.

Comparisons

Use the IF-THEN/ELSE statement to process statements when both true and false conditions are present or when more processing is required before values are generated.

See Also

Statements:

- "IF-THEN/ELSE Statement" on page 1256

IF-THEN/ELSE Statement

Executes a statement for rows that meet specific conditions.

Category: Local

Syntax

IF *expression* **THEN** *statement*;
[ELSE *statement* ;]

Arguments

expression
is any valid expression that evaluates to true or false and is a required argument.

See "DS2 Expressions" in *SAS DS2 Programmer's Guide*

statement
can be any executable statement or DO group.

Details

The expression in an IF-THEN statement is evaluated to produce a result that is either a nonzero value or zero. A nonzero value causes the expression to be true; a result of zero causes the expression to be false.

Note: In logical operations, including the IF-THEN/ELSE statement, a SAS missing value and a null value evaluate to zero (or false). To check for a null value in an IF-THEN-ELSE statement, you must use the NULL function as shown in this example.

```
method init ();
   x=null;
   if (null(x)) then
      put 'null';
   else
      put 'not null';
end;
```

If the conditions that are specified in the IF clause are met, the IF-THEN statement executes a statement. An optional ELSE statement gives an alternative action if the THEN clause is not executed. The ELSE statement, if used, must immediately follow the IF-THEN statement.

Using IF-THEN statements *without* the ELSE statement causes all IF-THEN statements to be evaluated. If the IF clause is true, the statement after THEN is executed, otherwise the statement after ELSE is executed.

Note: For greater efficiency, construct your IF-THEN/ELSE statement with conditions of decreasing probability.

Note: You can use an IF expression to select between two values based on whether a conditional evaluates to true or false. In addition, IF expressions can be nested to select between many values for a multi-way decision. For more information, see "IF Expression" on page 1943.

Comparisons

- Use a SELECT expression rather than a series of IF-THEN statements when you have a long series of mutually exclusive conditions. A large number of nested IF-THEN/ELSE statements could cause an internal error. By contrast, the SELECT expression is evaluated only once, which could improve performance.
- Use subsetting IF statements, without a THEN clause, to continue processing only those expressions that evaluate to nonzero values when the condition indicates that no more processing is required and no output is to be produced.

Example

These examples illustrate the IF-THEN/ELSE statement.

- ```
  if a = b then
     d = e;
  else
  ```
  ```

```
   d = f;
```

- ```
  if status='OK' and type=3 then count+1;
  ```
- ```
  if x=0 then
     if y ne 0 then put 'X ZERO, Y NONZERO';
     else put 'X ZERO, Y ZERO';
  else put 'X NONZERO';
  ```
- ```
  if answer=9 then
     do;
        answer=.;
        put 'INVALID ANSWER FOR ' id=;
     end;
  else
     do;
        answer=answer10;
        valid+1;
     end;
  ```
- ```
  xx = if
     if x then y
     else z then b
  else c;
  ```

See Also

- "IF Expression" on page 1943

Statements:

- "IF Statement: Subsetting" on page 1255

KEEP Statement

Includes columns in output tables.

Category: Global

Note: This statement cannot be used within a method.

Syntax

KEEP *column-list* | *vararray*;

Arguments

column-list
specifies the names of one or more columns to write to the output table.

Restriction Numbered range lists in the format *col1–coln* are not supported.

vararray
specifies the name of a variable array.

See "VARARRAY Statement" on page 1323

Details

The KEEP statement specifies that all columns in the column list should be included in the creation of output rows. When the KEEP statement is specified, all columns that are not included in the KEEP statement are dropped from the output rows. When no DROP or KEEP statement appears, all tables that are created in the DS2 program contain all columns. Do not use both DROP and KEEP statements within the same DS2 program.

Comparisons

- The KEEP statement applies to all output tables that are named in the DATA statement. To write different columns to different tables, you must use the KEEP= table option.
- The DROP statement is a parallel statement that specifies columns to omit from the output table.
- The KEEP and DROP statements select columns to include in or exclude from output tables. The subsetting IF statement selects rows.
- Do not confuse the KEEP statement with the RETAIN statement. The RETAIN statement holds a row value in a column from one iteration of the DS2 RUN method to the next iteration. The KEEP statement does not affect the row values, but specifies only which columns to include in any output tables.

Examples

Example 1

This example uses the KEEP statement to include only the columns NAME and AVG in the output table.

```
data;
   keep name avg;
   method run();
      set table-specification;
   end;
enddata;
```

Example 2

In this example, the first three variables in variable array x are kept in the output.

```
/* keep x1, x2, x3 */
proc ds2;
data mytest;
   vararray double x[10];
   keep x1-x3;
   method init();
     do i = 1 to 10;
       x[i] = i;
     end;
     output;
   end;
enddata;
run;
quit;
```

See Also

Statements:

- “DROP Statement” on page 1243

Table Options:

- “KEEP= Table Option” on page 1434

Labeled Statement

Identifies a statement that is referred to by another statement.

Category: Local

Syntax

label: *statement*; [... *statement*];

Arguments

label
specifies any identifier, which is followed by a colon (:). You must specify the label argument.

Range 1–255 characters

Restriction If the label contains non-Latin characters, you must enclose it in double quotation marks.

statement
specifies any executable statement, including a null statement (;). You must specify the statement argument.

Details

A label associates an identifier with a given statement so that the statement can be referred to by other statements, such as GOTO and LEAVE. You can have multiple labels for a statement.

Comparisons

The LABEL attribute of the DECLARE statement's HAVING clause assigns a descriptive label to a column. A statement label identifies a statement or group of statements that are referred to in the same DS2 program by another statement, such as GOTO or LEAVE.

Example

- ```
  restock:
     if x > 1 then
        y = 3;
     else
        y = 5;
  ```
- ```
  label1:
     label2:
        do i = 1 to 3;
           j = j * i;
        end;
  ```

See Also

Statements:

- "GOTO Statement" on page 1253
- "LEAVE Statement" on page 1262

LEAVE Statement

Stops processing the current DO loop and transfers execution to either the statement following the current DO statement, or a labeled DO statement that encloses the current DO statement.

Category: Local

Syntax

LEAVE [*identifier*] ;

Without Arguments

The LEAVE statement stops the processing of the current DO statement and resumes processing with the next statement following the current DO statement.

Arguments

identifier
label associated with the target DO statement.

Details

You can use the LEAVE statement to exit a DO loop prematurely. You can use the LEAVE statement either on its own or use it based on a condition (for example, in conjunction with an IF statement). If the LEAVE statement is not followed by an identifier, then the target is the DO statement immediately enclosing the LEAVE statement. If the LEAVE statement is followed by an identifier, then the target is the DO statement with the label specified by the identifier. The target DO statement must enclose the current DO statement. An error occurs if the identifier specifies a statement other than a DO statement, or a DO statement that does not enclose the current DO statement. An error also occurs if the specified label does not exist.

Comparisons

- The LEAVE statement stops the processing of the current DO statement and resumes program execution outside of the current DO statement.
- The CONTINUE statement stops the processing of the current iteration of a DO statement and resumes program execution with the next iteration of the current DO statement.

Examples

Example 1: LEAVE Statement without an Identifier

This example illustrates the LEAVE statement without an identifier.

```
data _null_;
   dcl int i;
   method init();
      do i = 1 to 10;
         if i > 4 then leave;
         put i;
      end;
   end;
enddata;
```

The following lines are written to the SAS log.

```
1
2
3
4
```

Example 2: LEAVE Statement with an Identifier

This example illustrates the LEAVE statement with an identifier.

```
data _null_;
   dcl int sum i j k ;
   method init();
      lab1: do i = 1 to 5;
         sum + 1;
      lab2: do j = 1 to 3;
         sum + 1;
         do k = 1 to 3;
            put sum i j k;
            sum + 1;
            if j = 2 then
               leave lab2;
            if i = 2 then
               leave lab1;
            if k = 2 then
               leave ;
            end;
         end;
         end;
   end;
enddata;
```

The following lines are written to the SAS log.

```
2 1 1 1
3 1 1 2
5 1 2 1
8 2 1 1
```

See Also

Statements:

- “CONTINUE Statement” on page 1219
- “DO Statement” on page 1236

MERGE Statement

Joins rows from two or more tables into a single row.

Category: Local

Restriction: Tables that are in SPD Engine or HDMD format do not support the MERGE statement.

Requirement: The MERGE statement must be followed by a BY statement.

Interactions: The MERGE statement can be executed inside the database when you are using the SAS In-Database Code Accelerator. If you are using the SAS In-Database Code Accelerator for Hadoop, Hive .13 or later is required. The SAS In-Database Code Accelerator is not supported in SAS Viya.

Setting the PROC DS2 BYPARTITION=NO option does not affect a MERGE statement when you are using in-database processing. In-database processing is not supported on the CAS server.

Note: The variables that are read using the MERGE statement are set to either a missing or null value.

Tip: The width of the resulting column is determined by the largest width across all the tables in a single SET statement. Trailing blanks are irrelevant to the MERGE statement.

Syntax

MERGE <table-reference> <table-reference> [... <table-reference>] [/RETAIN];

<table-reference>::=

{ *table* [(*table-options*)] }

Arguments

table

specifies the name of the table. *table* can be one of these forms.

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*

- *table-name*
- *caslib.table-name*

catalog is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

table-name is the name of the table.

caslib.table-name is the name of the table on the CAS server.

Notes If the table name has a dot in it and you are accessing a CAS table, you must enclose the table name in double quotation marks. Here is an example:

```
data mycaslib "tdlibref.foo";
```

If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in `data mytable;`, the table name is stored as "MYTABLE" and in `data "MyTable";`, the table name is stored as "MyTable". This is important if table and schema names in your data source are case-sensitive.

CAUTION **Using the PRESERVE_TAB_NAMES=no option in your LIBNAME statement can cause unexpected results.**

(*table-options*)
specifies optional arguments that the DS2 program applies when it writes rows to the output table.

Note For more information about table options, see "DS2 Table Options" on page 1347.

/RETAIN
specifies that the final row of a data set in a particular BY group be used repeatedly until there are no more rows in any of the contributing data sets.

Restriction This argument is available only in SAS 9.4M6 and SAS Viya 3.4.

Tip Use this argument to achieve merge behavior like that of a DATA step merge.

Details

The MERGE statement is flexible and has a variety of uses in DS2 programming. This section describes basic uses of MERGE. Other applications include using more than one BY variable, merging more than two tables, and merging a few rows with all rows in another table.

Match-merging combines rows from two or more tables into a single row in a new table according to the values of a common variable. The number of rows in the new table is the sum of the largest number of rows in each BY group in all tables. To perform a match-merge, use a BY statement immediately after the MERGE statement. The variables in the BY statement must be common to all tables. Only one BY statement can accompany each MERGE statement in a data program.

For more information, see "Match-Merging" in *SAS DS2 Programmer's Guide*.

CAUTION
BY variables in a DS2 merge that have a DECIMAL or NUMERIC data type are converted to a DOUBLE data type. If matching DECIMAL columns are not BY variables, the DECIMAL columns remain as a DECIMAL data type.

CAUTION
If there is a type, scale, or precision mismatch between columns with a DECIMAL or NUMERIC data type between tables, the column is converted to a DOUBLE data type.

Note: The order of the data sets in the MERGE statement can affect the matching.

Note the difference in merging behavior between the DATA step and DS2. The MERGE statement does not produce a Cartesian product on a many-to-many match-merge. Instead, it performs a sparse one-to-one merge while there are rows in the BY group in at least one table. In comparison to DATA step merging, the result of the DS2 MERGE statement is a subset of the Cartesian product. By contrast, the result of the DATA step merge is not a subset because it uses a lazy retain strategy to fill in the PDV.

Here is the code.

```
merge T1 (in=inT1) T2 (in=inT2); by K;
```

The following example shows the DATA step retain strategy during the merging of the tables T1 and T2 by a common variable K. When match-merging of the second row of table T1 occurs, the value of column C is retained from the previous match of the BY variable. The variable inT2 is set to 1 indicating that table T2 contributed to the final table results.

Figure 10.1 *DATA Step Merge*

T1

K	A	B
101	Cat	Apple
101	Pig	Banana
102	Duck	Fig

T2

K	A	C
101	*Cow*	*Red*
102	.	*Yellow*
103	.	*Blue*

K	inT1	inT2	A	B	C
101	1	1	*Cow*	Apple	*Red*
101	1	1	Pig	Banana	*Red*
102	1	1	.	Fig	*Yellow*
103	0	1	.	.	*Blue*

In contrast to the DATA step merge, the DS2 merge clears the PDV between BY-group processing. Because the second row in table T1 does not have a corresponding match in table T2, column C remains empty and the variable inT2 is set to `0` indicating that table T2 did not contribute to the final table results.

Figure 10.2 *DS2 Merge*

T1

K	A	B
101	Cat	Apple
101	Pig	Banana
102	Duck	Fig

T2

K	A	C
101	*Cow*	*Red*
102	.	*Yellow*
103	.	*Blue*

K	inT1	inT2	A	B	C
101	1	1	*Cow*	Apple	*Red*
101	1	0	Pig	Banana	.
102	1	1	.	Fig	*Yellow*
103	0	1	.	.	*Blue*

In SAS 9.4M6 and SAS Viya 3.4, however, you can use the RETAIN argument in the MERGE statement to produce a many-to-many match-merge that is similar to a DATA step merge. To see the same DS2 program shown above produce similar results as the DATA step program, see "Example 2: Merging Tables with Retained Rows" on page 1269.

Match-merging tables that do not contain a one-to-one row mapping between the table rows can produce unexpected results. Within a given BY group, observations in a BY group are not necessarily selected in the order in which they appear in the data set during match-merging.

Note: The MongoDB data source has special requirements for creating tables. Depending on your data, you might want to create these tables: a root table, a parent table, and a child table. You can create only root tables with DS2. To create

parent and child tables, you must use FedSQL. See the information about MongoDB in *SAS/ACCESS for Relational Databases: Reference*.

Comparisons

If you specify a SET statement, SAS stops processing before all rows are read from all tables if the number of rows are not equal. In contrast, SAS continues processing all rows in all tables that are named in the MERGE statement.

Examples

Example 1: Merging Two Tables Using One Column

The following example creates two tables and uses a MERGE statement to combine the tables using the **common** column.

```
data mrg01a(overwrite=yes);
   dcl varchar(10) common animal;
   method init();
      common='a'; animal='Ant'; output;
      common='b'; animal='Bird'; output;
      common='c'; animal='Cat'; output;
      common='d'; animal='Dog';  output;
      common='e'; animal='Eagle'; output;
      common='f'; animal='Frog';  output;
   end;
enddata;
run;


data mrg01b(overwrite=yes);
   dcl varchar(10) common plant;
   method init();
      common='a'; plant='Apple'; output;
      common='b'; plant='Banana'; output;
      common='c'; plant='Coconut'; output;
      common='d'; plant='Dewberry';  output;
      common='e'; plant='Eggplant'; output;
      common='f'; plant='Fig';  output;
      common='g'; plant='Grapefruit'; output;
   end;
enddata;
run;


/*  match merge */
data;
   method run();
      merge mrg01a mrg01b; by common;
   end;
```

```
 enddata;
run;
```

The following table is generated.

common	animal	plant
a	Ant	Apple
b	Bird	Banana
c	Cat	Coconut
d	Dog	Dewberry
e	Eagle	Eggplant
f	Frog	Fig
g		Grapefruit

Example 2: Merging Tables with Retained Rows

The following example re-creates the data sets shown in the Details sections and merges them using the RETAIN argument to produce a result that is similar to the Cartesian product that the DATA step produced.

```
/********** create data sets ***********/
proc ds2;
data t1 (overwrite=yes);
   dcl double k;
   dcl char a b;
   method init();
      k=101; a='Cat'; b='Apple'; output;
      k=101; a='Pig'; b='Banana'; output;
      k=102; a='Duck'; b='Fig'; output;
   end;
enddata;
run;
quit;

proc ds2;
data t2 (overwrite=yes);
   dcl double k;
   dcl char a c;
   method init();
      k=101; a='Cow'; c='Red'; output;
      k=102; a='' ; c='Yellow'; output;
      k=103; a='' ; c='Blue';  output;
   end;
enddata;
run;
quit;

/*************Merge with Retain argument **********/
proc ds2;
data ds2merge (overwrite=yes);
```

```
   dcl double k;
   dcl char a b c;
   method run();
      merge t1 t2 /retain;
      by k;
   end;
enddata;
run;
quit;

proc print data=ds2merge;
run;
quit;
```

The following table is created. It is identical to the table produced by the DATA step merge code shown in "Details" on page 1266.

Obs	k	a	b	c
1	101	Cow	Apple	Red
2	101	Cow	Banana	Red
3	102		Fig	Yellow
4	103			Blue

See Also

- "Match-Merging" in *SAS DS2 Programmer's Guide*

Statements:

- "BY Statement" on page 1214
- "SET Statement" on page 1305

METHOD Statement

Defines a block of code that can be called and executed multiple times.

Category: Block

Syntax

[PRIVATE] **METHOD** *method* ([[IN_OUT] <parameter> [, ... [IN_OUT]<parameter>]])

[RETURNS *data-type*];

...*method-body* ...
END;

<parameter>::=

<data-type> *variable*

< data-type >::=

<exact-numeric-type> | <approximate-numeric-type> | <binary-string-type> | <string-type>
| <date-type>

<exact-numeric-type> ::=

{**INT** | **BIGINT** | **SMALLINT** | **TINYINT**
| **DECIMAL** [(*precision* [,*scale*])] | **NUMERIC** [(*precision* [,*scale*])] }

<approximate-numeric-type> ::=

{ **DOUBLE** | **DOUBLE PRECISION** | **FLOAT** | **REAL** }

<binary-string-type>::=

BINARY(*length*) | **VARBINARY**(*length*)

<string-type>::=

NCHAR [(*character-length*)]

| **NVARCHAR** [(*character-length*)]

| **CHAR** [(*character-length*)] [CHARACTER SET *character-set-identifier*]

| **VARCHAR** [(*character-length*)] [CHARACTER SET *character-set-identifier*]

<date-type>::=

{ **TIME** | **TIMESTAMP** } [(*precision*)] | **DATE**

Arguments

PRIVATE
specifies a method that can be accessed only from within the package.

See "Attributes and Methods" in *SAS DS2 Programmer's Guide*

method
specifies a name for the method. Method names have global scope.

IN_OUT
specifies that the argument is to be manipulated by reference, not by value. The IN_OUT parameter manipulates the argument rather than a copy of the argument.

Requirements The argument that is passed to the IN_OUT parameter must be a modifiable value, such as an identifier, not an expression.

The data type of the argument must be the same data type as the IN_OUT parameter.

Note If the method declaration specifies a length for an IN_OUT parameter, a warning is issued and the length is ignored. The length of the supplied argument is always used.

Tip DS2 methods can support up to 1000 arguments. A DS2 method that has more than 1000 arguments can generate a compilation error.

See "Avoiding Issues with Precedence in Expressions" on page 1276

"Passing Character Values to Methods" on page 1275

parameter
specifies a parameter that is passed to the method. The type can be any valid character, numeric, or date type, or package. Parameters have scope that is local to the method and, by default, parameters are passed by value. A parameter is initialized to be a copy of the value of the argument that is specified for the parameter, unless it is specified with the IN_OUT parameter or is a package or an array. Package type variables and arrays are always passed by reference, even if the IN_OUT keyword is not specified.

Interaction If the IN_OUT parameter or a package parameter is specified, the value of the argument that is passed into the parameter might have changed when the method returns.

Tip DS2 methods can support up to 1000 arguments. A DS2 method that has more than 1000 arguments can generate a compilation error.

INT | BIGINT | SMALLINT | TINYINT
specifies an integer variable or array.

Alias INTEGER for INT

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

DECIMAL[(*precision* [, *scale*])] | NUMERIC[(*precision* [, *scale*])]
specifies an exact numeric variable or array.

precision
specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point

Note Not all data sources can support a precision of 52 digits.

scale
specifies the maximum number of decimal digits that can be stored to the right of the decimal point

Range 0–*precision*

Note *scale* is less than or equal to *precision*.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

DOUBLE | DOUBLE PRECISION | FLOAT | REAL
specifies a floating-point variable or array.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

BINARY (*length*)
specifies a binary variable or array.

Requirement If you specify BINARY, you must also specify the *length* of the variable or array in bytes.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

VARBINARY (*length*)
specifies a varying-length binary variable or array.

Alias BINARY VARYING

Requirement If you specify VARBINARY, you must also specify the *length* of the binary variable or array in bytes.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

NCHAR | NVARCHAR | CHAR | VARCHAR
specifies a character variable or array.

Aliases NATIONAL CHARACTER, NATIONAL CHAR for NCHAR

NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING for NVARCHAR

CHARACTER for CHAR

CHARACTER VARYING for VARCHAR

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

character-length
specifies the maximum number of characters that the string can hold for NCHAR, NVARCHAR, CHAR, and VARCHAR data types.

Default 8

Tip The number of bytes that character variables that are declared using CHAR use for storage depends on the session encoding. Those declared using any of the NCHAR variants have wider storage and can be used to represent character sets for which single-byte character storage is insufficient (for example, Unicode). If a session encoding requires multiple bytes per character (for example, UTF-8), then CHAR and NCHAR are identical types and both use NCHAR.

CHARACTER SET *character-set-identifier*
specifies character set encoding information for CHAR and VARCHAR data types.

Default Default encoding depends on your operating system and locale.

Tip You can use a character string literal or a simple string for character set names. For example, you can specify `"ibm-866"` or `'ibm-866'`

See For a complete list of character set encoding values, see "Encoding Values in SAS Language Elements" in the *SAS National Language Support (NLS): Reference Guide*.

TIME
specifies a time variable or array.

TIMESTAMP
specifies both a date and time variable or array.

precision
specifies the precision for a TIME or TIMESTAMP data type.

Default 6

Note If you are working with TIME and TIMESTAMP values in a data source other than SAS and you do not specify a precision, the default precision will always be the DS2 default precision of 6.

DATE
specifies a date variable or array.

variable
specifies the scalar variable or array name. You can specify one or more variables or arrays. However, *variable* can only be of the type specified in *data-type*. You can mix scalar and array variables of the same type.

RETURNS *data-type*
specifies the data type of the value that the method returns. The type can be any valid character, numeric, or date type.

method-body
comprises the variable declarations and executable DS2 code that runs when the method is called. All variables that are declared in the method body are local to the method.

END
marks the end of the method.

Details

Method Basics

There are two types of methods in DS2: *system* methods, and *user-defined* methods. The METHOD statement enables you to create your own user-defined methods. For information about system methods, see "DS2 System Methods" on page 1333.

If a method returns a value, each RETURN statement that appears in the method must have an associated return expression. For more information, see the "RETURN Statement" on page 1300.

When a method definition appears after any method expression that refers to it, a FORWARD statement for the method must appear for the method before the method expression. User-defined methods that are not defined before the DS2 INIT, RUN, or TERM methods must be declared in a FORWARD statement. For more information, see the "FORWARD Statement" on page 1252.

Note: TINYINT and SMALLINT method parameters are automatically promoted to INTEGER, and REAL method parameters are automatically promoted to DOUBLE. A warning message appears. For more information, see "DS2 Type Conversions for Expression Operands" in *SAS DS2 Programmer's Guide*.

Passing Character Values to Methods

DS2's default parameter-passing semantics are pass-by-value. This means that a copy of the parameter is made and the copy is passed to the method, not the original parameter. Any changes to the parameter by the method are made to the copy and not to the original parameter. This ensures that the value of the original parameter is maintained for the caller of the method.

Given the potential size of character variables, the cost of making the copy, in terms of both memory and time, can be great. To avoid this cost, consider using the IN_OUT parameter, which causes the parameter to be passed-by-reference. This means that the location or address of the parameter is passed to the method, and a second copy of the parameter is not made. Changes that are made to the value of the parameter by the method, either intentional or unintentional, are reflected in any reference to the parameter by the caller after the method returns.

This syntax shows the two types of parameter passing:

```
/* pass-by-value */
method copy_made(char(256) x);
  ...
end;

/* pass-by-reference */
method no_copy(in_out char x);
  ...
end;
```

Returning Character Values from Methods

When a string value is returned by a method, DS2 creates a temporary string to hold the return value. There is a cost to copy the returned value from the temporary string to the target location. To avoid the cost of the copy, provide the target location of the return value with an IN_OUT parameter.

```
method copy_made() returns char(1024);
  return 'very long string';
end;
```

```
method no_copy(in_out char return_val);
  return_val = 'very long string';
end;
```

One benefit of passing the target location with an IN_OUT parameter is that the method does not need to specify the size of the returned string value at compilation.

Avoiding Issues with Precedence in Expressions

The order of precedence for method variables is not defined in DS2. Therefore, using a variable more than once in the same expression where it is also used as an IN_OUT variable can produce unexpected results. Here is an example.

```
METHOD addValue( in_out integer i) returns integer;
    i = i + 100;
    return i;
    end;
X = 0;
Y = addValue(X) + X;
```

The order of precedence is not defined for evaluating X in the expression `addValue (X) + X`. If the expression is evaluated from left-to-right, then the addValue method is called before the rightmost X is captured, resulting in Y=200. If the expression is evaluated right-to-left, then the rightmost X is captured first (as zero), and the result is Y=100.

Examples

Example 1: User-defined Methods

In these three examples, M, CONCAT, and ADD are user-defined methods.

- ```
  method m(int x, int y) returns int;
     return x + y;
  end;
  ```
- ```
  method concat(char(100) x, char(100) y) returns char(200);
     return trim(x) || y;
  end;
  ```
- ```
  method add(double x, double y);
     this.x = this.x + x;
     this.y = this.y + y;
  end;
  ```

Example 2: Overloaded Methods

In the following examples, the D method and the CONCAT methods are overloaded. If any two method definitions have the same name, but different type signatures (that is, if the methods have different types), the method is overloaded.

```
method d(double x, double y) returns double;
   dcl double temp;
```
```

```
   temp = x * 99;
   return x + y + temp;
end;
method d(double x, int y) returns double;
   return x + y
end;
method d(int x);
   put x;
end;
method concat(char(100) x, char(100) y) returns char(200);
   return "pre" || trim(x) || y;
end;
method concat(char(100) x, char(100) y, char(100) z) returns char(200);
   return trim(x) || trim(y) || z;
end;
```

Example 3: Using the IN_OUT Argument

In the following example, the method *swapper* exchanges argument values. The IN_OUT argument enables the values to be changed where the method is called.

```
package xyzzy;
   method swapper(in_out double a, in_out double b);
      declare double x;
      x=a; a=b; b=x;
   end;
endpackage;
run;

data _null_;
   method init();
      dcl package xyzzy x();
      a=10; b=42;
      put 'before: ' a= b=;
      x.swapper(a,b);
      put 'after:  ' a= b=;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
before:  a=10 b=42
after:  a=42 b=10
```

Example 4: Passing in a Package as a Method Argument

In the following example, the *person_list* package contains the two methods *addP1* and *addP2*. The *addP1* and *addP2* methods each take the *person* package as an input parameter.

```
proc ds2;
  package work.person / overwrite=yes;
    declare varchar(32) lastname;
    declare varchar(32) firstname;
    method setNames(varchar(32) lastname_p, varchar(32) firstname_p);
```

```
      lastname = lastname_p;
      firstname = firstname_p;
    end;
    method getFullname() returns varchar(66);
      return (catx(', ', lastname, firstname));
    end;
  endpackage;
  package work.person_list / overwrite=yes;
    declare package work.person pl1;
    declare package work.person pl2;
    method addP1( package work.person pers_p);
      pl1 = pers_p;
    end;
    method addP2( package work.person pers_p);
      pl2 = pers_p;
    end;
    method getPersonList() returns varchar(256);
      return ( catx(':', pl1.getFullname(), pl2.getFullname()) );
    end;
  endpackage;
run;

  data new (overwrite=yes);
    declare char(66) myname;
    declare char(256) allNames;
      method init();
      declare package work.person pers1 ();
      declare package work.person pers2 ();
      declare package work.person_list pl ();
      pers1.setNames('Miller', 'Brad');
      pers2.setNames('Smith', 'Colin');
      myname = pers1.getFullname();
      pl.addP1(pers1);
      pl.addP2(pers2);
      allNames = pl.getPersonList();
      put myname=;
      put allNames=;
    end;
  enddata;
 run;
quit;
```

The following lines are written to the SAS log.

```
myname=Miller, Brad
allNames=Miller, Brad:Smith, Colin
```

See Also

- "Methods" in *SAS DS2 Programmer's Guide*

Statements:

- "FORWARD Statement" on page 1252

- “RETURN Statement” on page 1300

Methods:

- “INIT Method” on page 1335
- “REGISTEROUTPARAMETER Method” on page 1823
- “RUN Method” on page 1336
- “TERM Method” on page 1340

Null Statement

Creates an empty statement.

Category: Local

Syntax

;

Details

The Null statement consists solely of a semicolon. It creates an empty statement.

Example

This example shows how the Null statement can be used to not execute any statements.

```
method init();
   x = 1;
   y = 0;
   z = 1;
   if x & y | not z then
      ;
   else
      put 'else';
end;
```

OUTPUT Statement

Writes the current row to a table.

Category: Local

Syntax

OUTPUT [{ *table* [... *table*] } | _ROWSET_ | _NULL_];

Without Arguments

Using OUTPUT without arguments causes the current row to be written to all tables that are named in the DATA statement or to the thread program. If no tables are specified in the DATA statement or to the thread program, then the row is written to the client application.

Arguments

table

specifies the name of the table to which to write rows. *table* can be one of these forms.

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*
- *table-name*
- *caslib.table-name*

catalog is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

table-name is the name of the table.

caslib.table-name is the name of the table on the CAS server.

Restriction All names specified in the OUTPUT statement must also appear in the DATA statement.

Notes If the table name has a dot in it and you are accessing a CAS table, you must enclose the table name in double quotation marks. Here is an example:

```
data mycaslib "tdlibref.foo";
```

If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in `data mytable;`, the table name is stored as "MYTABLE" and in `data "MyTable";`, the table name is stores as "MyTable". This is important if table and schema names in your data source are case-sensitive.

Tip You can specify up to as many tables in the OUTPUT statement as you specified in the DATA statement for that DS2 program.

CAUTION **Using the PRESERVE_TAB_NAMES=no option on your LIBNAME statement can cause unexpected results.**

ROWSET
specifies that the OUTPUT statement should not write rows to a table, but it should instead return table rows to the client application.

NULL
specifies that the OUTPUT statement should not write rows to either a table or the client application.

Details

When and Where the OUTPUT Statement Writes Rows

The OUTPUT statement creates an output row, using values for the row that are contained in the global variables when output statement executes. The OUTPUT statement writes the current row to a table immediately, not at the end of the DS2 program. If no table name is specified in the OUTPUT statement, the row is written to the table or tables that are listed in the DATA statement.

DS2 keeps track of the values in the order in which the compiler encounters them within a DS2 program, whether they are read from existing tables or created in the program.

Note: The MongoDB data source has special requirements for creating tables. Depending on your data, you might want to create these tables: a root table, a parent table, and a child table. You can create only root tables with DS2. To create

parent and child tables, you must use FedSQL. See the information about MongoDB in *SAS/ACCESS for Relational Databases: Reference*.

Implicit versus Explicit Output

If you do not supply an OUTPUT statement, DS2 adds one implicitly at the end of the RUN method that writes rows to the table or tables that are being created.

Placing an explicit OUTPUT statement in a DS2 program overrides the automatic output, and adds a row to a table only when an explicit OUTPUT statement is executed. Once you use an OUTPUT statement to write a row to any one table, however, there is no implicit OUTPUT statement at the end of the RUN method. In this situation, a DS2 program writes a row to a table only when an explicit OUTPUT statement executes. You can use the OUTPUT statement alone or as part of an IF-THEN/ELSE or SELECT statement or in DO loop processing.

Using the OUTPUT Statement in DS2 Program Threads

OUTPUT statements in thread programs cannot contain any table names. Each output row is returned to the thread program that started the thread.

Comparisons

- The OUTPUT statement writes rows to a table or to the client application; the PUT statement writes variable values or text strings to the SAS log.
- To control whenever a row is written to a table, use the OUTPUT statement. To control which columns are written to a table, use the KEEP= or DROP= table option in the DATA statement or use the KEEP or DROP statement.

Examples

Example 1: Sample Uses of OUTPUT

- This line of code writes the current row to a table.

```
output;
```

- This line of code writes the current row to a table when a specified condition is true.

```
if deptcode gt 2000 then output;
```

- This line of code writes a row to the MARKUP table when the PHONE value is missing.

```
if phone=. then output markup;
```

Example 2: Creating Multiple Rows from Each Row of Input

You can create two or more rows from each row of input data. This DS2 program creates three rows in the RESPONSE table for each row in the SULFA table:

```
data response(drop= (time1 time2 time3));
   method run();
      set sulfa;
      time=time1;
      output;
      time=time2;
      output;
      time=time3;
      output;
    end;
enddata;
```

Example 3: Creating Multiple Tables from a Single Input Table

You can create more than one table from one input table. In this example, OUTPUT writes rows to two tables, EASTERN and WESTERN:

```
data eastern western;
   method run();
      set cities;
      if location = 'east' then output eastern;
      else output western;
   end;
enddata;
```

Example 4: Creating One Row from Several Rows of Input

You can combine several input rows into one row. In this example, OUTPUT creates one row that totals the values of DEFECTS in the first ten rows of the input table:

```
data discards;
 drop defects;
   method run();
      set gadgets;
           reps+1;
      if reps=1 then total=0;
      total+defects;
      if reps=10 then do;
         output;
         stop;
       end;
   end;
enddata;
```

Example 5: Output Using Threads

The following example generates a result set of 20 random numbers. The data program starts 4 threads. Each thread generates and writes 5 random numbers with variable **x**. The data program reads the output of each thread with the SET statement and then writes the input random numbers to the data set **random_data**.

```
/* Thread generates and outputs 5 random numbers */
```

```
thread thread_pgm;
   declare double x;
   method init();
      declare int i;
      streaminit(_threadid_);
      do i = 1 to 5;
         x = rand('uniform', 1);
         output;               /* output variable x */
      end;
   end;
endthread;

 data random_data;
    dcl thread thread_pgm t;
    method run();
       /* Start 4 threads and read the output of each thread. */
       set from t threads=4;
    end;
enddata;
run;
```

The following output is generated.

The SAS System

Obs	x
1	0.58602
2	0.78108
3	0.03955
4	0.31368
5	0.73902
6	0.45233
7	0.98123
8	0.06654
9	0.10132
10	0.62904
11	0.51530
12	0.88424
13	0.35661
14	0.11297
15	0.16502
16	0.79072
17	0.90079
18	0.79053
19	0.26467
20	0.22305

See Also

Statements:

- "PUT Statement" on page 1291
- "DATA Statement" on page 1220

PACKAGE Statement

Creates a DS2 package.

Category: Block

Syntax

Form 1:

PACKAGE *package* [/ENCRYPT=SAS | AES] [/*table-options*];
... package-body ...
ENDPACKAGE;

Form 2:

PACKAGE *fcmp-package-name* [/ENCRYPT=SAS | AES] [/*table-options*]
LANGUAGE=[']FCMP['] TABLE='*library-name*';
... package-body ...
ENDPACKAGE;

Arguments

package
specifies the package name. *package* can be one of these forms.

- *catalog.schema.package*
- *schema.package*
- *catalog.package*
- *package*

catalog
is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema
is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

package
is the name of the package.

Requirement Package naming conventions are based on the data source. For more information, see the documentation for your data source.

/ENCRYPT=SAS | AES
specifies the encryption algorithm. SAS specifies the SAS Proprietary algorithm. AES specifies the Advanced Encryption Standard algorithm.

Default SAS

Interaction The ENCRYPT option for the PACKAGE statement is different from and has different values than the ENCRYPT= table option. The ENCRYPT= table option affects only SAS output data sets. For more information, see "ENCRYPT= Table Option" on page 1415.

/table-options
specifies optional arguments that the DS2 program applies when it creates a package. For more information about table options, see "DS2 Table Options" on page 1347.

Note Options that are not recognized by DS2 are passed without error to the underlying data source.

package-body
contains the declarations and methods in the package.

fcmp-package-name
specifies the name of the FCMP package.

Restriction This argument is not supported on the CAS server.

See "Using the FCMP Package" in *SAS DS2 Programmer's Guide*

'library-name'
specifies the name of the library where the FCMP function resides.

Restriction This argument is not supported on the CAS server.

Requirement This location must be the library.dataset portion of the location that was specified in the OUTLIB option in the PROC FCMP statement.

See The FCMP procedure in *Base SAS Procedures Guide*

Details

Package Basics

A package is similar to a DS2 program. The package body consists of a set of global declarations and a list of methods. The main syntactical differences are the

PACKAGE and ENDPACKAGE statements. These statements define a block with global scope. For more information about scope, see "Scope of DS2 Identifiers " in *SAS DS2 Programmer's Guide*.

The package's library of methods can be called from any other DS2 program including another package. Consequently, the INIT, RUN, and TERM methods have no special meaning in a package.

After successful compilation of a package, a copy of the package's source code is stored in the catalog entry that is identified by the package name.

The PACKAGE statement is required for all user-defined packages and for the FCMP package that is supplied by SAS. The hash, hash iterator, HTTP, JSON, logger, matrix, PCRXFIND, PCRXREPLACE, and SQLSTMT packages do not require a PACKAGE statement. These packages are also supplied by SAS. For more information, see "Introduction to DS2 Packages" in *SAS DS2 Programmer's Guide*.

Package variables are declared using the DECLARE PACKAGE statement. Declaring a package variable does not automatically construct an instance of a package. A package instance is constructed either by providing constructor arguments when a package variable is declared or by calling the _NEW_ operator.

By default, DS2 packages that are saved to a table for reuse are encrypted with SAS encryption. You can override this default and specify AES encryption by using the ENCRYPT= table option in the PACKAGE statement. SAS Proprietary is a fixed encoding algorithm that is included with Base SAS software. It requires no additional SAS product licenses. For more information, see *Encryption in SAS*.

Table options can be specified in the PACKAGE statement. They are specified after the package name and preceded by a slash.

FCMP Packages

SAS provides an FCMP package that supports calls to FCMP functions and subroutines from within the DS2 language. For more information, see "Using the FCMP Package" in *SAS DS2 Programmer's Guide*.

Comparisons

For a comparison of packages, DS2 programs, and threads, see "Block Statements" on page 1205.

Examples

Example 1: Creating a Complex Number Package

This example creates a very simple complex number package. Two global variables, `X` and `Y`, represent the ordered pair of real numbers that constitute a complex number. This set of methods performs various operations on complex numbers, such as add and multiply.

```
package complex;
   dcl double x y;
  method init(double x, double y);
     this.x = x;
     this.y = y;
  end;
  method add(double x, double y);
     this.x = this.x + x;
     this.y = this.y + y;
  end;
  method mult(double x, double y);
     this.x = this.x * x - this.y * y;
     this.y = this.x * y + x * this.y;
  end;
  method norm() returns double;
     return sqrt(x ** 2 + y ** 2);
  end;
  method print();
     put 'x = ' x ' y= ' y;
  end;
endpackage;
run;
```

The following DS2 program instantiates and calls the methods defined in the previous PACKAGE statement.

```
data _null_;
   method init();
      dcl package complex c();
      dcl package complex c2();
      c.x=3;
      c.y=4;
      d = c.norm();
      put 'd= ' d;
      c.add(5, 6);
      c.print();
      c2.x=7;
      c2.y=24;
      d = c2.norm();
      put 'd= ' d;
      c2.print();
   end;
enddata;
run;
```

The following lines are written to the SAS log:

```
d=  5
x =  8  y=  10
d=  25
x =  7  y=  24
```

Example 2: Creating an FCMP Package

This example creates a square routine in FCMP and uses that routine in a DS2 program. The current directory is used as the "library" of FCMP packages.

```
libname base '.';
```

```
* fcmp defines a function, square;
proc fcmp outlib = base.fcmpsubs.package1;
   function square(a);
      return (a*a);
   endsub;
run;

* define the ds2 package thru which the fcmp functions will be called;
proc ds2;
package pkg /
   overwrite=yes
   language='fcmp'
   table='base.fcmpsubs';
run;

* call fcmp thru the ds2 wrapper package;
data;
   dcl package pkg p();
   dcl double a b;
   method init();
      do a = 10 to 20;
         b=p.square(a);
         put a= b=;
      end;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
a=10 b=100
a=11 b=121
a=12 b=144
a=13 b=169
a=14 b=196
a=15 b=225
a=16 b=256
a=17 b=289
a=18 b=324
a=19 b=361
a=20 b=400
```

See Also

- "User-Defined Packages" in *SAS DS2 Programmer's Guide*
- "Using the FCMP Package" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement" on page 1231
- "DECLARE PACKAGE Statement: FCMP Package" on page 1495

PUT Statement

Prints the values of program variables, arrays, and constants to the SAS log.

Category: Local

Syntax

PUT < put-list > [... <put-list>] ;

<put-list>::=

ALL

| '*character-string*'

| ['*character-string*']<eq-expression> [=] [[:] *format* [-L | -C | -R]]

<eq-expression>::=

identifier

| *array-reference*

| *this-expression*

Without Arguments

PUT without arguments prints a blank line to the SAS log.

Arguments

ALL
prints the values of all variables, which includes predefined variables, to the SAS log.

'character-string'
specifies a string of text that is written to the SAS log.

identifier
names a variable whose value is written to the SAS log.

array-reference
specifies an array element. The subscript can be any SAS expression that resolves to an integer value when the PUT statement executes. Use the array subscript asterisk (*) to write all elements of the array.

this-expression
specifies a THIS expression.

See "THIS Expression" on page 1942

=
If an equal sign is added after a variable or array element, then the output is preceded by the variable or array element name and an equal sign.

:
enables you to specify a format that the PUT statement uses to write the variable value. All leading and trailing blanks are deleted, and each value is followed by a single blank.

Restriction If you use ":", you must specify a format.

format
specifies a format to use when the data value is written to the SAS log. If you use a colon modifier (:) with the format name, all leading and trailing blanks are deleted and each value is followed by a single blank. To override the default alignment, you can add an alignment specification to a format:

-L left aligns the value.

-C centers the value.

-R right aligns the value.

Tip Ensure that the format width provides enough space to write the value and any commas, dollar signs, decimal points, or other special characters that the format includes.

Details

How to Use the PUT Statement

The PUT statement consists of the keyword PUT followed by a list of variables and constants. You list the names of the variables whose values you want written, or you specify a character string in quotation marks. If you do not specify a format, the PUT statement writes a variable value with the default format, inserts a single blank, and then writes the next value. If you specify a format, the output is written using the format width. Character values are left-aligned in the field; leading and trailing blanks are removed. Numeric values are right-aligned in the field.

How nonexistent data (SAS missing values or null values) are written depends on whether you are in ANSI mode or SAS mode. A period is generated for DOUBLE missing dot and null values when the default format, BEST32., is associated with the DOUBLE. For special missing values (.a-.z and ._), the character after the period is generated when using BEST32.. For example, if a variable held the value .a, A would be generated. INTEGER and other non-DOUBLE numeric types cannot be missing. However, they can be null in which case nothing is generated by the PUT statement when the value is null. For more information, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.

How List Output Is Spaced

List output uses different spacing methods when it writes variable values and character strings. When a variable is written with list output, SAS automatically inserts a blank space. The output pointer stops at the second column that follows the variable value.

```
dcl int a b;
dcl varchar c d;
area = 9924;
city = 1001;
ctry1='Peru';
ctry2='Bolivia';
    put area city ctry1 ctry2;
```

These lines are written to the SAS log.

```
----+----1----+----2
9924 1001 Peru Bolivia
```

However, when a character string is written, SAS does not automatically insert a blank space. The output pointer stops at the column that immediately follows the last character in the string.

To avoid spacing problems when both character strings and variable values are written, you might want to use a blank space as the last character in a character string.

If you use a colon modifier (:) with the format name, SAS writes the value with the specified format, inserts a blank space, and moves the pointer to the next column.

```
dcl int a b;
dcl varchar c d;
area = 9924;
POP = 1000;
ctry1='Peru';
ctry2='Bolivia';
put area : octal10. pop : comma8.2 ctry1 ctry2;
```

These lines are written to the SAS log.

```
----+----1----+----2----+----3----+
0000023304 1,000.00 Peru Bolivia
```

Formatted Output

You can use a format to control how SAS prints the variable values. The PUT statement uses the format that follows the variable name to write each value. With formatted output, SAS does not automatically add blanks between values. Formatted output moves the pointer the length of the format, even if the value does not fill that length. The pointer moves to the next column; an intervening blank is not inserted. If the value uses fewer columns than specified, character values are left-aligned and numeric values are right-aligned in the field that is specified by the format width.

```
dcl int a b;
dcl varchar c d;
pop = 1000
area = 9924;
```

```
ctry1='Peru';
ctry2='Bolivia';
put area octal10. pop comma8.2 ctry1 ctry2;
```

These lines are written to the SAS log.

```
----+----1----+----2----+----3----+
000000233041,000.00Peru Bolivia
```

If no format is specified, the variable's default format is used. You can associate a format with a column by using a HAVING clause in the DECLARE statement. For more information, see "DECLARE Statement" on page 1224.

Comparisons

The PUT statement and the PUT function have similar behavior. However, the PUT statement directs its results to the SAS log whereas the PUT function returns an NCHAR value containing the result of formatting its argument.

Examples

Example 1: PUT Statements

This example contains several PUT statements.

```
x = 1;
y = 2;
z = 3;
s = 'abc';
a[4] = 99;
put 'x = ' x;
put 'y = ' y;
put z s a[4];
```

Note: If an equal sign is added after a variable or array element, then the output is preceded by the variable or array element name and an equal sign. For example, these two code lines are equivalent.

```
put x= y= a[2]=;
put 'x=' x ' y=' y ' a[2]=' a[2];
```

Example 2: Using PUT to Write Arrays

This example writes the contents of both temporary and variable arrays.

```
data _null_;
   declare  double a[6] having format 4.2;
   vararray double b[2, 3];
   declare  double c[0:1, 2:4, 5:5];
```

```
   method init();
      a := (3 6 9 12 15 18);
      b := (3 6 9 12 15 18);
      c := (3 6 9 12 15 18);
      put a[*]=;
      put b[*]=;
      put c[*]=;
      put 'array a with default format:' a[*];
      put 'array a with best3.  format:' a[*] best3.;
   end;
enddata;
```

The following lines are written to the SAS log.

```
a[1]=3.00 a[2]=6.00 a[3]=9.00 a[4]=12.0 a[5]=15.0 a[6]=18.0
b[1,1]=3 b[1,2]=6 b[1,3]=9 b[2,1]=12 b[2,2]=15 b[2,3]=18
c[0,2,5]=3 c[0,3,5]=6 c[0,4,5]=9 c[1,2,5]=12 c[1,3,5]=15 c[1,4,5]=18
array a with default format: 3.00 6.00 9.00 12.0 15.0 18.0
array a with best3.  format:   3  6  9 12 15 18
```

Example 3: Using the _ALL_ Argument

This example uses the _ALL_ argument in the PUT statement to print the values of all variables, including the predefined _N_ variable, to the SAS log.

```
data;
  dcl double a b c;
  method init();
    a = 116;
    b = 220;
    c = 37;
    put _all_;
  end;
enddata;
run;
```

The following lines are written to the SAS log.

```
a=116 b=220 c=37 _n_=1
```

See Also

- "How to Write Array Content" in *SAS DS2 Programmer's Guide*

Functions:

- "PUT Function" on page 992

RENAME Statement

Specifies new names for columns in output tables.

Category: Global

Syntax

RENAME *old-name* {= | AS} *new-name* [... *old-name* {= | AS} *new-name*];

Arguments

old-name
specifies the name of a column as it appears in the input table, or in the current DS2 program for newly created columns.

new-name
specifies the name to use in the output table.

Details

The RENAME statement enables you to change the names of one or more columns. The new column names are written to the output table only. Use the old column names in programming statements for the current DS2 program. RENAME applies to all output tables. In addition to changing the name of a column, the RENAME statement also changes the label for the column.

Comparisons

- The RENAME= table option enables you to specify the columns that you want to rename for each input or output table. Use it in input tables to rename columns before processing.
- If you use the RENAME= table option in an output table, you must continue to use the old column names in programming statements for the current DS2 program. The RENAME= table option affects only that output table. The RENAME statement affects all output tables.
- If you use both the RENAME statement and RENAME= output table option, the RENAME statement has precedence. If X is renamed to Y with a RENAME statement and X is renamed to Z with a RENAME= table option, the RENAME statement takes precedence and X is renamed to Y.

- The RENAME= table option in the SET statement renames columns in the input table. You must use the new names in programming statements for the current DS2 program.

Example

The following examples illustrate the RENAME statement.

- `rename prod=ProductName;`
- `rename street as Address cit as City st as State;`
- `rename oldsalary=newsalary;`

See Also

Table Option

- "RENAME= Table Option" on page 1452

RETAIN Statement

Specifies that all columns or all columns in the column list have their values retained between executions of the RUN method.

Category: Global

Syntax

Form 1: **RETAIN**;

Form 2: **RETAIN** *column-list*;

Form 3: **RETAIN** *column-list* < constant-value >;

Form 4: **RETAIN** *column-list* (< constant-value > ... < constant-value >);

< constant-value >::=

bit_constant

| *hex_constant*

| *floating_constant*

| *decimal_constant*

| *sas_missing_value*

| *integer_constant*

| *string_constant*

| *null*

| **DATE** *character_constant*

| **TIME** *character_constant*

| **TIMESTAMP** *character_constant*

Form 5: **RETAIN** *vararray*;

Without Arguments

(Form 1) If you do not specify any arguments, the RETAIN statement causes the values of all columns to be retained from one execution of the RUN method to the next.

Arguments

column-list
specifies column names whose values you want retained.

vararray
specifies the name of a variable array.

See "VARARRAY Statement" on page 1323

Details

Column Behavior (Form 2)

The RETAIN statement specifies that all columns in the column list should have their values retained during each execution of the RUN method. Normally, columns in the PDV are set to either a missing or null value before the RUN method executes.

Assigning Initial Values (Forms 3 and 4)

Use a RETAIN statement to specify initial values for individual columns, a list of columns, or members of an array. If a value appears in a RETAIN statement, columns that appear before it in the list are set to that value initially.

(Form 3) You can assign one value to all columns. For example, the following statement assigns a value of 100 to columns A, B, and C.

```
retain a b c 100;
```

(Form 4) You can assign different values to each column. For example, the following statement assigns the values 'Vancouver', 'BC', and 'Canada' to columns CITY, PROVINCE, and COUNTRY.

```
retain city province country ('Vancouver', 'BC', 'Canada');
```

Note that you can also use an iterator to assign one value to all columns. For example, the following statement assigns a value of 100 to columns A, B, and C.

```
retain a b c (3* 100);
```

(Form 5) You can assign initial values to the array variables. The RETAIN initializer works across array boundaries. In addition, if the initial list is short, the remaining values are set to missing values. In this example, the value of ns[3] is a missing value.

```
proc ds2;
data y /overwrite=yes;
   vararray double c[5];
   vararray double sums[dim(c)];
   vararray double ns[dim(c)];

   retain sums ns (1 2 3 4 5 6 7);
   method init();
      sums[4] = 99;
   end;

   method run();
      put sums[1]=;
      put sums[2]=;
      put sums[3]=;
      put sums[4]=;
      put sums[5]=;
      put ns[1]=;
      put ns[2]=;
      put ns[3]=;
   end;
enddata;
run;
quit;
```

Redundancy

It is redundant to name any of these items in a RETAIN statement, because their values are automatically retained from one iteration of the DS2 program to the next:

- columns that are read with a SET statement

  Note: It might not be redundant if multiple tables are associated with the SET statement. Assume that table **A** defines variable **X** but table **B** does not. Without a RETAIN statement, variable **X** is set to missing when records are read from table **B**. With a RETAIN statement, variable **X** has whatever value was last assigned to it by table **A** or by DS2 program logic.

- a column whose value is assigned in a sum statement
- data elements that are specified in an array

Comparisons

The RETAIN statement specifies columns whose values are preserved. The KEEP statement specifies columns that are to be included in any table that is being created.

See Also

Statements:

- "KEEP Statement" on page 1258

RETURN Statement

Returns execution from a method to the method caller.

Category: Local

Syntax

RETURN [*expression*];

Without Arguments

When a RETURN statement does not have an *expression*, control is transferred back to the caller of the method in which the RETURN statement is located. No value is returned to the caller of the method.

Arguments

expression
specifies any valid expression that returns a single value. The expression's type is evaluated, and if necessary, converted to the type specified in the METHOD statement's RETURNS clause. The value of *expression* is then passed back to the caller of the method.

Details

When the RETURN statement is executed in the implicit loop, the next iteration of the implicit loop executes. The RETURN statement transfers control and, if *expression* is present, returns a value back to the caller of the method. Any method that returns a value (in other words, that has a RETURNS clause in the METHOD statement) must have RETURN *expression* statement as the last statement in the METHOD body. Otherwise, an error occurs. A warning occurs if a method has a RETURN *expression* statement, but does not have a RETURNS clause.

You can use the STOP statement to terminate the RUN method.

Example

In this example, the CONCAT method returns a concatenated string. The RETURN statement's type is converted to the type of the RETURNS clause. In this example, the return type is CHAR.

```
method concat(char(100) x, char(100) y) returns char(200);
   return trim(x) || y;
end;
```

See Also

Statements:

- "METHOD Statement" on page 1270

SELECT Statement

Executes one of several statements or groups of statements.

Category: Local

Syntax

SELECT [(*select-expression*)];

[< when-list > [...< when-list>]] ;

[OTHERWISE *statement-list*] ;

END [*end-label*];

<when-list>::=

WHEN (*when-expression*) [*statement-list*]

Arguments

select-expression
specifies an expression that evaluates to a single value of any type other than VARBINARY.

end-label
The END statement closes the SELECT statement. The optional *end-label* argument specifies an identifier. This label, created by using the Labels statement, must match the label immediately preceding the SELECT statement, or an error will occur.

when-expression
specifies any expression.

Requirement You must specify at least one *when-expression*.

statement-list
can be any executable statement or statements.

Details

Using WHEN Statements in a SELECT Group

The SELECT statement begins a SELECT group. SELECT groups contain WHEN statements that identify DS2 statements that are executed when a particular condition is true. Use at least one WHEN statement in a SELECT group. An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. An END statement ends a SELECT group.

Null statements that are used in WHEN statements cause no further action to be taken when the condition is true.

Note: SELECT statements can be nested.

Note: You can use a SELECT expression to select between multiple expressions based on the values of other expressions. For more information, see "SELECT Expression" on page 1946.

Note: In logical operations, any expression, including the WHEN statement, a missing value evaluates to False in SAS mode; a null value evaluates to neither true nor false in ANSI mode.

Evaluating the *when-expression* When a *select-expression* Is Included

If the *select-expression* is present, both the *select-expression* and *when-expression* are evaluated. The two are compared for equality and a value of true or false is returned. If the comparison is true, the *statement-list* is executed and processing exits the SELECT statement. No other conditions are tested.

If the comparison is false, execution proceeds to the next WHEN statement. If no WHEN statements remain, execution proceeds to the OTHERWISE statement, if one is present. If the result of all SELECT-WHEN comparisons is false and no OTHERWISE statement is present, no error is given and the DS2 program continues to execute.

Evaluating the *when-expression* When a *select-expression* Is Not Included

If no *select-expression* is present, the *when-expression* is evaluated to produce a result of true or false. If the result is true, the *statement-list* is executed and processing exits the SELECT statement. No other conditions are tested.

If the result is false, execution proceeds to the next WHEN statement, or to the OTHERWISE statement if one is present. (That is, the action that is indicated in the first true WHEN statement is performed.) If the result of all *when-expressions* is false and no OTHERWISE statement is present, an error message is issued. If more than one WHEN statement has a true *when-expression*, only the first WHEN statement is used. Once a *when-expression* is true, no other *when-expressions* are evaluated.

Evaluating the *when-expression* When a *statement-list* Is Not Included

If a *when-expression* appears without a *statement-list*, it uses the *statement-list* of the next *when-expression*. The following example produces an output of 10.

```
a = 10;
    select(a);
    when(10)
       when(20) put a=;
```

However, a *when-expression* without a *statement-list* breaks this behavior. This example produces no output.

```
a = 10;
    select(a);
    when(10) ;
    when(20) put a=;
```

Note: This is not true for a *when-expression* that precedes OTHERWISE. In this case, a *when-expression* without a statement is treated as if it has an empty statement (;).

Comparisons

Use IF-THEN/ELSE statements for programs with few statements. Use subsetting IF statements without a THEN clause to continue processing only those rows that meet the condition that is specified in the IF clause.

Examples

Example 1: SELECT with a select-expression

This example illustrates how to use the SELECT statement with a *select-expression*.

```
select (a);
   when (1) x=x*10;
   when (2);
   when (3) x=x*100;
   when (4) x=x*100;
   when (5) x=x*100;
   otherwise;
end;
```

Example 2: SELECT without a select-expression

This is an example of a SELECT statement without a *select-expression*.

```
select;
   when (mon in ('JUN', 'JUL', 'AUG')
   and temp>70) put 'SUMMER ' mon=;
   when (mon in ('MAR', 'APR', 'MAY'))
   put 'SPRING ' mon=;
   otherwise put 'FALL OR WINTER ' mon=;
end;
```

Example 3: SELECT without an IF Expression

This example uses a SELECT statement. You could also use IF expressions. IF expressions allow a more compact representation of calculations.

```
temp3=.;
select(age);
   when(12) temp3=0;
   when(13) temp3=-14.2769145744513;
   when(14) temp3=-27.3577925153955;
   when(15) temp3=-21.9485551871151;
   when(16) temp3=-13.1764092846992;
   when(17) temp3=-0.63898626243485;
end;
temp4=.;
select(sex);
   when('F') temp4=-1.47186167693037;
   when('M') temp4=1.47186167693037;
end;
temp5=.;
select(age);
   when(12) DO;
      select(sex);
         when('F') temp5=0;
         when('M') temp5=0;
      end;
   end;
   when(13) DO;
```

```
      select(sex);
         when('F') temp5=7.47012474340756;
         when('M') temp5=-7.47012474340756;
      end;
   end;
   when(14) DO;
      select(sex);
         when('F') temp5=4.35656087162482;
         when('M') temp5=-4.35656087162482;
      end;
   end;
   when(15) DO;
      select(sex);
         when('F') temp5=2.40720037896732;
         when('M') temp5=-2.40720037896732;
      end;
   end;
   when(16) DO;
      select(sex);
         when('F') temp5=7.56020843202274;
         when('M') temp5=-7.56020843202274;
      end;
   end;
   when(17) DO;
      select(sex);
         when('F') temp5=-0.0520606347702515;
         when('M') temp5=0.0520606347702515;
      end;
   end;
end;

predictedWeight = -182.556181904311 + temp3 + temp4 +
   4.85030791094268*height + temp5;
```

See Also

- “SELECT Expression” on page 1946

Statements:

- “IF Statement: Subsetting” on page 1255
- “IF-THEN/ELSE Statement” on page 1256

SET Statement

Reads rows from one or more tables.

Category: Local

Interactions: Only one SET statement is allowed when using the SAS In-Database Code Accelerator. If more than one SET statement is used in the thread program, the thread program is not run inside the database. Instead, the thread program runs on the client.

In SAS 9.4M3, multi-table SET statements and a SET statement with embedded SQL code can be executed inside the database when you are using the SAS In-Database Code Accelerator. If you are using the SAS In-Database Code Accelerator for Hadoop, Hive .13 or later is required. The SAS In-Database Code Accelerator is not supported on SAS Viya.

Note: Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. *sql-text* must be enclosed in braces ({ }).

Tips: A SET statement in a thread program shares a single reader for that SET statement. Each time a SET statement is executed, one row is read from the named input table. However, all threads share a single reader. Each row in the input table is sent to exactly one thread. If you use a SET statement in the INIT method, the first thread reads all the rows. The other threads reach end-of file, and processing is terminated without their advancing to their associated RUN methods. Consequently, it is recommended that you not use the SET statement in the INIT or TERM method of a thread program.

The width of the resulting column is determined by the largest width across all the tables on a single SET statement. Trailing blanks are irrelevant to the SET statement.

Syntax

SET < table-reference > [... < table-reference >] [INDSNAME=*variable*] ;

[BY [DESCENDING] *column* [... [DESCENDING] *column*]] ;

< table-reference>::=

{ *table* [(*table-options*)] }

| \{ *sql-text* \}

Arguments

INDSNAME=*variable*

creates and names a variable that stores the name of the table from which the current row is read. The stored name can be a table name or a physical name. The physical name is the name by which the operating environment recognizes the file.

Data type: NCHAR

Tips: Although the INDSNAME variable is automatically declared as NCHAR, you can explicitly declare it as CHAR.

Unless previously defined, the length of the variable is set to 41 characters. If the variable is declared as CHAR with a specific length,

that length is not changed. If the value placed into the INDSNAME variable is longer than that length, then the value is truncated.

column
names each column by which the table is sorted.

Tip The table can be sorted by more than one column.

table
specifies the name of the input table. *table* can be one of these forms.

- *catalog.schema.table-name*
- *schema.table-name*
- *catalog.table-name*
- *table-name*
- *caslib.table-name*

catalog	is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.
schema	is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.
table-name	is the name of the table.
caslib.table-name	is the name of the table on the CAS server.

Notes If the table name has a dot in it and you are accessing a CAS table, you must enclose the table name in double quotation marks. Here is an example:

```
data mycaslib "tdlibref.foo";
```

If you do not use quotation marks around the table and schema names, DS2 stores them as uppercase and includes double quotation marks. Table and schema names that are enclosed in quotation marks are used as is. That is, they remain quoted and with the original casing in the quotation marks. For example, in `data mytable;`, the table name is stored as "MYTABLE" and in `data "MyTable";`, the table name is stored as "MyTable". This is important if table and schema names in your data source are case-sensitive.

CAUTION **Using the PRESERVE_TAB_NAMES=no option in your LIBNAME statement can cause unexpected results.**

table-options
specifies optional arguments that the DS2 program applies when it writes rows to the output table. For more information about table options, see "DS2 Table Options" on page 1347.

{sql-text}
is any valid FedSQL code that resolves to a set of table rows.

Restriction: The SQL in a SET statement can reference columns only if they are included in the data source table.

Requirement: The FedSQL query must be enclosed in braces ({ }).

Tips: You can use *sql-text* to merge rows from one or more tables.

The SQL in a SET statement is evaluated statically at compile time.

See: The SELECT statement in *SAS FedSQL Language Reference*

DESCENDING
specifies that the tables are sorted in descending order by the column that is specified. DESCENDING means largest to smallest for numeric columns, or reverse alphabetical for character columns.

Details

What SET Does

The SET statement is flexible and has a variety of uses in DS2 programming. These uses are determined by the options and statements that you use with the SET statement:

- reading rows and columns from existing tables for further processing in a DS2 program
- concatenating and interleaving tables, and performing one-to-one reading of tables

Each time the SET statement executes, one row is read into the program data vector. SET reads all columns one row at a time from the input tables unless you specify otherwise. A SET statement can contain multiple tables; a DS2 program can contain multiple SET statements.

Note: A SET statement in a thread program shares a single reader for that SET statement. Each row in the input table is sent to exactly one thread.

Note: The SET statement is best used in the RUN method to take advantage of the RUN method's implicit looping capability.

Examples

Example 1: Reading a Table

In this example, each row in the table NC.MEMBERS is read into the program data vector. Only those rows whose value of CITY is `Raleigh` are written to the new table RALEIGH.MEMBERS:

```
data raleigh.members;
   method run();
      set nc.members;
      if city='Raleigh';
   end;
enddata;
run;
```

Example 2: Concatenating Tables

If more than one table name appears in the SET statement, the resulting output table is a concatenation of all the tables that are listed. SAS reads all rows from the first table, and then all from the second table, and so on, until all rows from all the tables have been read. This example concatenates the three tables into one output table named FITNESS:

```
data fitness;
   method run();
      set health exercise well;
   end;
enddata;
run;
```

Example 3: Interleaving Tables

To interleave two or more tables, use a BY statement after the SET statement:

```
data april;
   method run();
      set payable recvable;
      by account;
   end;
enddata;
run;
```

Example 4: Combining a Single Row with All Rows in a Table

A row to be combined into an existing table can be one that is created by another DS2 program. In this example, the table AVGSALES has only one row:

```
data national;
   method init ();
      set avgsales;
   end;
   method run();
      set totsales;
   end;
```

```
enddata;
run;
```

Example 5: Reading from the Same Table More Than Once

In this example, SAS treats each SET statement independently. That is, it reads from one table as if it were reading from two separate tables. The LOCKTABLE=share option is used so that the same data set (in this case trial5) can be opened at the same time:

```
data drugxyz;
   method run();
      set trial5(locktable=share keep=(sample));
      if sample>2;
         set trial5;
   end;
enddata;
run;
```

For each iteration of the DS2 program, the first SET statement reads one row. The next time the first SET statement is executed, it reads the next row. Each SET statement can read different rows with the same iteration of the DS2 program.

See Also

- "BY-Group Processing When Running Thread Programs inside the Database" in *SAS In-Database Products: User's Guide*
- "Reading Data Using the SET Statement" in *SAS DS2 Programmer's Guide*
- "Combining DS2 Tables: Methods" in *SAS DS2 Programmer's Guide*

Statements:

- "BY Statement" on page 1214
- "DATA Statement" on page 1220
- "DECLARE PACKAGE Statement: Matrix Package" on page 1703
- "MERGE Statement" on page 1264

Table Option:

- "IN= Table Option" on page 1429

SET FROM Statement

Runs a DS2 program as one or more threads.

Category: Local

Restriction: Multiple SET FROM statements are not allowed in a data program. Otherwise, an error occurs.

Syntax

SET FROM *thread* [THREADS = *threads*];

Arguments

thread

specifies the thread name that is executed by the SET statement. *thread* can be one of these forms.

- *catalog.schema.thread*
- *schema.thread*
- *thread*

catalog

is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema

is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

thread

is the name of the thread.

Requirements: The thread name must match the name of a thread created in a THREAD statement and the thread must be created before the SET FROM statement is executed, or an error will occur.

Thread naming conventions are based on the data source. For more information, see the documentation for your data source.

See: "Threaded Processing" in *SAS DS2 Programmer's Guide*

THREADS= *threads*

specifies the number of threads that are run for *thread*.

Restrictions: This argument has no effect if the SAS In-Database Code Accelerator is used to access a database table. The SAS In-Database Code Accelerator always uses either the number of reducers that are specified in the config file with any maximum limitations that are set by the administrator or it uses one reducer

when processing the data program that is forced down to one reducer. The SAS In-Database Code Accelerator is not supported on SAS Viya.

If you are using an FCMP package, the maximum value of *threads* is 100. Otherwise, there is no restriction on the value for *threads*.

Requirement *threads* must be an integer value.

Tip If *threads* is not present, the thread runs as a single thread.

See "THREADS= Argument and the SAS In-Database Code Accelerator" on page 1312

Details

The Basics

The SET FROM statement enables a DS2 program to run as a single thread or as multiple threads. The thread name specified in a SET FROM statement references a DS2 program thread that has been created by a THREAD statement.

Note: The SET FROM statement is best used in the RUN method to take advantage of the RUN method's implicit looping capability.

Note: The MongoDB data source has special requirements for creating tables. Depending on your data, you might want to create these tables: a root table, a parent table, and a child table. You can create only root tables with DS2. To create parent and child tables, you must use FedSQL. See the information about MongoDB in *SAS/ACCESS for Relational Databases: Reference*.

THREADS= Argument and the SAS In-Database Code Accelerator

If the thread program is run inside the database, the number of threads is set by the SAS In-Database Code Accelerator. The THREADS= argument has no effect if the SAS In-Database Code Accelerator for Greenplum, Hadoop, or Teradata is used to access a database table.

For more information about using the SAS In-Database Code Accelerator, see *SAS In-Database Products: User's Guide*.

Note: The SAS In-Database Code Accelerator is not supported in SAS Viya.

THREADS= Argument and DS2 Threaded Programs in CAS

The basic computational model for analytics in SAS Cloud Analytic Services (CAS) combines distributed computing and multi-threaded computing. CAS actions that pass through the data typically do so by involving multiple threads: each thread receives a portion of the data, and each record in the input table is consumed by only one thread. These threads are called *worker threads* to emphasize the concurrent nature of their execution. The maximum number of concurrent worker threads that you are allowed to use is determined by your software license.

If your DS2 program specifies more threads than the maximum that you are allowed, DS2 reduces the number of threads to that maximum number. If your DS2 program either specifies 0 threads or does not specify the number of threads, DS2 uses the maximum number of threads allowed, as determined by your software license.

Comparisons

After the thread specified in SET FROM begins execution, the SET FROM statement executes similarly to the SET statement.

Similarities to note are:

- The PDV information for the thread is read by the DS2 program in which the SET FROM statement appears, so that all the output variables from the thread are declared automatically with correct types in the DS2 program.
- SET FROM and SET both loop through input until there are no more rows to read.

Here are differences to notice:

- Instead of reading from tables, the SET FROM statement reads the output from each of the threads.
- In general, the SET FROM statement's input consists of the stream of output produced by all the running threads, via the thread's OUTPUT statement. Because the execution order of threads is unpredictable, the input is not read sequentially like the SET statement reads tables. If the thread contains a SET statement that reads rows from tables, the rows are asynchronously divided among the threads. If a thread is not using a SET statement to read data, then the SET FROM statement's input is similar to reading one or more copies of a table, but with no given order on the incoming rows.

Examples

Example 1: Running a Single Thread

In this example, threads X and Y are automatically declared in the DS2 program with the appropriate types. When the SET FROM statement executes, T is started as a single thread, its rows are read, and a simple calculation is done for each.

```
thread work.t;
   dcl int x;
   dcl double y;
   method init();
      dcl int i;
      do i = 1 to 5;
         x = i;
         y = i * 2.5;
         output;
      end;
   end;
endthread;
data;
   dcl thread work.t t;
   method run();
      set from t;
      sum = x + y;
      put ' x= ' x ' y= ' y ' sum= ' sum;
   end;
enddata;
```

The following lines are written to the SAS log.

```
x=  1  y=  2.5   sum=  3.5
x=  2  y=  5     sum=  7
x=  3  y=  7.5   sum=  10.5
x=  4  y=  10    sum=  14
x=  5  y=  12.5  sum=  17.5
```

Example 2: Running Multiple Threads

This example modifies a thread, T, to run multiple threads by adding the THREADS option to the SET FROM statement.

```
thread work.t;
   dcl int x;
   dcl double y;
   method init();
      dcl int i;
      do i = 1 to 5;
         x = i;
         y = i * 2.5;
         output;
      end;
   end;
endthread;
data;
   dcl thread work.t t;
   method run();
      set from t threads=2;
      sum = x + y;
      put ' x= ' x ' y= ' y ' sum= ' sum;
   end;
enddata;
```

This runs two threads for T. The following lines are written to the SAS log.

```
x=  1  y=  2.5   sum=  3.5
x=  2  y=  5     sum=  7
x=  3  y=  7.5   sum=  10.5
x=  4  y=  10    sum=  14
x=  5  y=  12.5  sum=  17.5
x=  1  y=  2.5   sum=  3.5
x=  2  y=  5     sum=  7
x=  3  y=  7.5   sum=  10.5
x=  4  y=  10    sum=  14
x=  5  y=  12.5  sum=  17.5
```

In this case, the output is sequential, although there is no guarantee that will happen consistently.

Example 3: Accumulating Thread Values

In this example, the thread T generates a value of 1. In the DS2 program, four threads are started, and all output values are summed and printed in the TERM method.

```
thread t;
   dcl int x;
   method init();
      x = 1;
      output;
   end;
endthread;
data;
   dcl thread t t;
   dcl int sum;
   method init();
      sum = 0;
   end;
   method run();
      set from t threads=4;
      sum + x;
   end;
   method term();
      put 'sum= ' sum;
   end;
enddata;
```

The following line is written to the SAS log.

```
sum=  4
```

See Also

- "Threaded Processing" in *SAS DS2 Programmer's Guide*

Statements:

- "SET Statement" on page 1305
- "THREAD Statement" on page 1318

STOP Statement

Stops execution of the current DS2 program.

Category: Local

Syntax

STOP;

Without Arguments

The STOP statement causes processing of the current DS2 program to stop immediately and resume processing statements after the end of the current DS2 program.

Details

If DS2 generates a table, the row being processed when STOP executes is not added. The STOP statement can be used alone or in an IF-THEN/ELSE statement or SELECT group.

The TERM method always executes regardless of the method in which the STOP statement is executed. If you use the STOP statement in the TERM method, the TERM method will stop at the point where the STOP statement is executed.

If the STOP statement is executed in the INIT method or any method that is called from the INIT method, the RUN method does not execute.

Sum Statement

Adds or subtracts the result of an expression to an accumulator variable.

Category: Local

Note: The Sum statement can be used only with global variables. If you use the Sum statement with local variables, the values are not retained.

Syntax

variable + *expression*;

variable – *expression*;

Arguments

variable
specifies the name of the accumulator variable, which contains a numeric value.

Restrictions If you use an undeclared array in the Sum statement, an error occurs and a message is written to the SAS log.

The variable name cannot be “x”.

Tips The variable is automatically set to 0 before DS2 reads the first row. The variable's value is retained from one iteration to the next, as if it had appeared in a RETAIN statement.

To initialize a sum variable to a value other than 0, include it in a RETAIN statement with an initial value.

expression
is any valid DS2 expression.

Tip DS2 treats an expression that produces a missing or null value as zero.

See “DS2 Expressions” in *SAS DS2 Programmer's Guide*

Details

expression is evaluated and the result added to the accumulator variable. When the plus sign (+) is used, the result is added to the accumulator variable. When a minus sign (–) is used, the negative result is added to, in essence subtracted from, the accumulator variable.

Comparisons

The Sum statement is equivalent to using the SUM function and the RETAIN statement, as shown in this example.

```
retain variable 0;
variable=sum(variable,expression);
```

Examples

Example 1:

Here are examples of the Sum statement.

```
i + 2;
```

```
balance - debit;
numvalid + (not missing(x));
```

Example 2: Using the Sum Statement with Global and Local Variables

The following example uses both global and local variables in a Sum statement. Note that the value of the local variable, **x**, does not change.

```
data _null_;
  dcl int y;
   method m();
     dcl int x;
     x = 1;
     x + 2;
     y + 4;
     put x= y=;
   end;

   method init();
    do i = 1 to 3;
      m();
    end;
   end;
enddata;
run;
```

The following lines are written to the SAS log:

```
x=3 y=4
x=3 y=8
x=3 y=12
```

THREAD Statement

Creates a DS2 program thread.

Category: Block

Syntax

THREAD *thread* [(*data-type variable* [, ... *data-type variable*])] [/ENCRYPT=SAS | AES] [/*table-options*];

... *thread-body* ...

ENDTHREAD;

Arguments

thread

specifies the thread name. *thread* can be one of these forms.

- *catalog.schema.thread*
- *schema.thread*
- *catalog.thread*
- *thread*

catalog

is an implementation of the ANSI SQL standard for an SQL catalog, which is a data container object that groups logically related schemas. The catalog is the first-level (top) grouping mechanism in a data organization hierarchy that is used along with a schema to provide a means of qualifying names. A catalog is a metadata object in a SAS Metadata Repository.

schema

is an implementation of the ANSI SQL standard for an SQL schema, which is a data container object that groups files such as tables and views and other objects supported by a data source such as stored procedures. The schema provides a grouping object that is used along with a catalog to provide a means of qualifying names.

thread

is the name of the thread.

Requirement Thread naming conventions are based on the data source. For more information, see the documentation for your data source.

data-type

is an optional data type declaration. For more information, see "DS2 Data Types" in *SAS DS2 Programmer's Guide*.

variable

names an optional variable that identifies the parameter.

Restriction Thread parameters in the THREAD statement are not supported on the CAS server.

/ENCRYPT=SAS | AES

specifies the encryption algorithm. SAS specifies the SAS Proprietary algorithm. AES specifies the Advanced Encryption Standard algorithm.

Default SAS

Interaction The ENCRYPT option for the THREAD statement is different from and has different values from the ENCRYPT= table option. The ENCRYPT= table option affects only SAS output data sets. For more information, see "ENCRYPT= Table Option" on page 1415.

/table-options
specifies optional arguments that the DS2 program applies when it creates a thread. For more information about table options, see "DS2 Table Options" on page 1347.

thread-body
contains the declarations and methods in the thread.

Details

A DS2 thread begins with the THREAD statement and ends with the ENDTHREAD statement. These statements define a block with global scope. For more information about global scope, see "Scope of DS2 Identifiers " in *SAS DS2 Programmer's Guide*.

The thread body consists of a set of global declarations and a list of methods. You can specify the number of threads used by a thread program by using the SET FROM statement.

A DS2 program processes input data and produces output data. A DS2 program can run in two different ways: as a program and as a thread. When a DS2 program runs as a program, here are the results:

- Input data can include both rows from database tables and rows from DS2 program threads.
- Output data can be either database tables or rows that are returned to the client application.

When a DS2 program runs as a thread, here are the results:

- Input data can include only rows from database tables, not other threads.
- Output data includes the rows that are returned to the DS2 program that started the thread.

For more information about threads, see "Overview of Threaded Processing" in *SAS DS2 Programmer's Guide*.

A DS2 thread must be given a name. This name identifies a catalog entry in which the thread's source code is stored after it successfully compiles. Other DS2 programs and threads can then read and execute the thread by using the SET FROM statement.

When a thread is declared, a table is created with the name of the thread. A note is written to the SAS log that indicates that a table was created and where it was created, typically to the Work library. In most situations, a single-level named table does not persist after a SAS session ends. However, some single-level named tables do persist. Tables with multi-level names always persist after a SAS session. If a thread persists, it can be executed multiple times without having to redeclare the thread. You can add a DROP THREAD statement to your program to clean up unwanted tables.

Threads are declared for use in a DS2 program by using the DECLARE THREAD statement. When you declare a thread, the variable representing the thread is considered an instance of the thread. Thread variables can appear only in global

scope. Otherwise, an error occurs. For more information about instantiating a thread, see the DECLARE THREAD statement.

Note: If variables are declared with a HAVING clause in a thread program and the variables are redeclared in a data program with a HAVING clause, the HAVING clause in the data program is used instead of the HAVING clause in the thread program. If there is no HAVING clause in the DECLARE statement in the data program, the HAVING clause in the thread program is not used.

Threads can have parameters, as in this example:

```
thread work.t (double d, char (100) sp);
```

When you are using parameterized threads, the parameter names and their types are specified in the THREAD statement. The DS2 program that calls the thread must initialize the thread's parameters by calling the SETPARMS method. In this example, the parameter D is initialized with a value of *99* and the parameter SP is initialized with a value of *'ijk'*.

```
t.setparms(99, 'ijk');
```

TIP Package types are not supported as parameters in the THREAD statement. A DS2 thread can instantiate a local instance of a package within the thread program. The package instance is not accessible from any other DS2 thread, and data is not shared across threads. If the DS2 program spawns 10 DS2 threads, then there are 10 package instances in memory.

Note: Thread parameters in the THREAD statement and the SETPARMS method are not supported on the CAS server.

By default, DS2 threads are encrypted with SAS encryption. You can override this default and specify AES encryption by using the ENCRYPT table option in the THREAD statement. SAS Proprietary is a fixed encoding algorithm that is included with Base SAS software. It requires no additional SAS product licenses. For more information, see *Encryption in SAS*.

Table options can be specified in the THREAD statement. They are specified after the package name and preceded by a slash.

Note: The SAS In-Database Code Accelerator enables you to publish a thread program to the database and execute that thread program in parallel inside the database. For more information about using the SAS In-Database Code Accelerator, see *SAS In-Database Products: User's Guide*. The SAS In-Database Code Accelerator is not supported on SAS Viya.

Comparisons

For a comparison between packages, DS2 programs, and threads, see "Block Statements" on page 1205.

Examples

Example 1: Simple Thread

In this example, a single thread is created by using the THREAD statement.

```
thread t;
   dcl int x;
   method init();
      x = 99;
      output;
   end;
endthread;
```

Example 2: Running Multiple Threads

This example modifies a thread, T, to run multiple threads by adding the THREADS option to the SET FROM statement.

```
thread work.t;
   dcl int x;
   dcl double y;
   method init();
      dcl int i;
      do i = 1 to 5;
         x = i;
         y = i * 2.5;
         output;
      end;
   end;
endthread;
data;
   dcl thread work.t t;
   method run();
      set from t threads=2;
      sum = x + y;
      put ' x= ' x ' y= ' y ' sum= ' sum;
   end;
enddata;
```

This runs two threads for T. The following lines are written to the SAS log.

```
x=  1  y=  2.5   sum=  3.5
x=  2  y=  5     sum=  7
x=  3  y=  7.5   sum=  10.5
x=  4  y=  10    sum=  14
x=  5  y=  12.5  sum=  17.5
x=  1  y=  2.5   sum=  3.5
x=  2  y=  5     sum=  7
x=  3  y=  7.5   sum=  10.5
x=  4  y=  10    sum=  14
x=  5  y=  12.5  sum=  17.5
```

In this case, the output is sequential, although there is no guarantee that will happen consistently.

See Also

- "Threaded Processing" in *SAS DS2 Programmer's Guide*

Methods:

- "SETPARMS Method" on page 1338

Statements:

- "DECLARE THREAD Statement" on page 1234
- "SET FROM Statement" on page 1310

VARARRAY Statement

Declares one or more DS2 variable arrays.

Note: Square brackets in the syntax convention indicate optional arguments. The escape character (\ ) before a square bracket indicates that the square bracket is required in the syntax. Array bounds must be contained by square brackets ([ ]).

Syntax

VARARRAY <data-type> *array-name* <array-declaration> [<variable-list>] [<having-clause>];

< data-type >::=

<exact-numeric-type> | <approximate-numeric-type> | <binary-string-type> | <string-type>
| <date-type>

<exact-numeric-type> ::=

{**INT** | **BIGINT** | **SMALLINT** | **TINYINT**

| **DECIMAL** [(*precision* [,*scale*])] | **NUMERIC** [(*precision* [,*scale*])] }

<approximate-numeric-type> ::=

{ **DOUBLE** | **DOUBLE PRECISION** | **FLOAT** | **REAL** }

<binary-string-type>::=

BINARY(*length*) | **VARBINARY**(*length*)

<string-type>::=

NCHAR [(*character-length*)]

| **NVARCHAR** [(*character-length*)]

| **CHAR** [(*character-length*)] [**CHARACTER SET** *character-set-identifier*]

| **VARCHAR** [(*character-length*)] [**CHARACTER SET** *character-set-identifier*]

<date-type>::=

{ **TIME** | **TIMESTAMP** } [(*precision*)] | **DATE**

<array-declaration::=\[array-bound>[, ...<array-bound>]\

<array-bound>::= {[*dim-lower*:]*dim-upper*} | {[*dim-lower*:] {DIM(*a*[, *n*]) | *} }

<variable-list>::=

name-varlist |
| *numbered-range-varlist*
| *name-range-list*
| *name-prefix-list*
| *type-varlist*
| *special-name-list*

<having-clause>::=

HAVING <having-option> [... <having-option>]

<having-option>::=

LABEL '*string*' | n'*string*'

| **FORMAT** *format*

| **INFORMAT** *format*

Arguments

INT | BIGINT | SMALLINT | TINYINT

specifies an integer array.

Alias INTEGER for INT

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

| DECIMAL[(*precision* [, *scale*])] | NUMERIC[(*precision* [, *scale*])]

specifies an exact numeric variable or array.

precision

specifies the maximum total number of decimal digits that can be stored, both to the left and to the right of the decimal point

Note Not all data sources can support a precision of 52 digits.

scale
specifies the maximum number of decimal digits that can be stored to the right of the decimal point

Range 0–*precision*

Note *scale* is less than or equal to *precision*.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

DOUBLE | DOUBLE PRECISION | FLOAT | REAL
specifies a floating-point array.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

BINARY (*length*)
specifies a binary variable or array.

Requirement If you specify BINARY, you must also specify the *length* of the variable or array in bytes.

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

VARBINARY (*length*) | BINARY (*length*)
specifies a fixed-length or varying-length binary array.

Alias BINARY VARYING

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

NCHAR | NVARCHAR | CHAR | VARCHAR
specifies a character array.

Aliases NATIONAL CHARACTER, NATIONAL CHAR for NCHAR

NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING for NVARCHAR

CHARACTER for CHAR

CHARACTER VARYING for VARCHAR

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

character-length
specifies the maximum number of characters that the string can hold for NCHAR, NVARCHAR, CHAR, and VARCHAR data types.

Default 8

Requirement All character variables in a character variable array must have the same length.

CHARACTER SET *character-set-identifier*
specifies character set encoding information for CHAR and VARCHAR data types.

Default Default encoding depends on your operating system and locale.

Requirement All character variables in a character variable array must have the same encoding.

Tip You can use a character string literal or a simple string for character set names. For example, you can specify `"ibm-866"` or `'ibm-866'`

See For a complete list of character set encoding values, see "Encoding Values in SAS Language Elements" in the *SAS National Language Support (NLS): Reference Guide*.

TIME
specifies a time array.

TIMESTAMP
specifies both a date and time array.

precision
specifies the precision for a TIME or TIMESTAMP data type.

Default 6

Note If you are working with TIME and TIMESTAMP values in a data source other than SAS and you do not specify a precision, the default precision will always be the DS2 default precision of 6.

DATE
specifies a date array.

dim-lower* and *dim-upper
specifies a positive or negative integer used to define the number and size of the array boundary.

Tip If the lower bound of a dimension is not specified, then the lower bound defaults to 1.

See "Variable Array Declaration" in *SAS DS2 Programmer's Guide*

DIM(*a*[, *n*])
specifies that the size of the upper bounds of the array is determined by the number of elements in a dimension of a previously declared array by using a DIM function call.

a
specifies the name of a previously declared array.

n
specifies the dimension, in a multidimensional array, for which you want to know the number of elements.

Tip If no *n* value is specified, the DIM function returns the number of elements in the first dimension of the array.

Restriction The DIM function is the only function that you can use to specify an upper array bounds. The DIM function cannot be used to specify the lower bound of a dimension.

See "DIM Function" on page 521

*****

specifies a one-dimensional array in which the lower bound is 1 and the upper bound is the number of variables in the variable list.

Requirement You must specify at least one *variable* in the variable list.

<variable-list>

specifies the name of the variable(s) that is to be referenced by the elements of the array.

Requirement *variable* must be the same type specified in *data-type*.

Tip You can specify one or more variables.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

LABEL | '*string*' | n'*string*'

assigns a descriptive label to the variable array. The label can be a CHAR literal (*string*) or NCHAR literal (n*string*).

See "DS2 Data Types" in *SAS DS2 Programmer's Guide*

FORMAT *format*

Associates any valid DS2 format with the variable or array.

See "DS2 Formats" on page 57

INFORMAT *informat*

Associates any valid SAS informat with the variable or array.

See Chapter 8, "DS2 Informats," on page 1195

Details

You use the VARARRAY statement to create a variable array. A variable array is a temporary grouping of global variables. Only one array can be specified in a VARARRAY statement.

Variable arrays exist only for the duration of the DS2 program.

The different forms of variable lists can be mixed within a single variable list specification. For example, `vararray double a[*] u x1-x3 u:;` is a valid statement.

The above variable list would expand to `u x1 x2 x3 u u1 u2`. Therefore, a seven-element variable array would be constructed. Note that a single variable can be referenced by multiple elements of a variable array.

For more information, see "DS2 Arrays" in *SAS DS2 Programmer's Guide* and "Variable Arrays" in *SAS DS2 Programmer's Guide*.

For information about how to create a temporary array, see "DECLARE Statement" on page 1224.

Example

The following table contains examples of statements that specify variable arrays and the dimensions of those arrays.

Statement	Number of Dimensions	Range of Each Dimension	Number of Elements	Referenced Variables
`vararray double a[100];`	1	1:100	100	a1...a100
`vararray double a[2, 2];`	2	1:2 1:2	4	a1 a2 a3 a4
`vararray double a[-3:3, 5, 7:9, 10];`	4	-3:3 1:5 7:9 1:10	7x5x3x10 = 1050	a1 ... a1050
`vararray double a[3] x y z;`	1	1:3	3	x y z
`vararray double a[3] c3-c1;`	1	1:3	3	c3 c2 c1
`vararray double a[2, 2] t u v w;`	2	1:2 1:2	4	t u v w
`vararray double a[2, 2, 2] u v2-v4 w1-w3 x;`	3	1:2 1:2 1:2	8	u v2 v3 v4 w1 w2 w3 x
`vararray double a[*] x y z;`	1	1:3	13	x y z
`vararray double a[*] a1-a10;`	1	1:10	10	a1...a10

See Also

- "DS2 Arrays" in *SAS DS2 Programmer's Guide*
- "Variable Arrays" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE Statement" on page 1224

VARLIST Statement

Creates a named variable list.

Note: Square brackets in the syntax convention indicate optional arguments. The escape character (\ ) before a square bracket indicates that the square bracket is required in the syntax. Array bounds must be contained by square brackets ([ ]).

Syntax

VARLIST *list-name* \[*variable-list*\];

Arguments

list-name
specifies the name of the variable list.

[*variable-list*]
specifies the variables that are to be referenced by the list.

Requirement The *variable-list* must be enclosed in brackets ([]).

Details

Note that the VARLIST statement is limited to the global scope of the DS2 package or program. The VARLIST statement cannot be used to create a local variable list.

Examples

Example 1: Variable List That Contains All Variables in the PDV

In this example, the VARLIST statement creates a variable list named **allvars**, which contains all the PDV variables in the DS2 program.

```
varlist allvars [_all_];
```

Example 2: Initializing the Data Values in a Variable List

In this example, the variable list, c3, is declared and then initialized in the INIT method.

```
data;
   declare int cost1 cost2 cost3 cost4 cost5;
   varlist c3 [cost1-cost5];
```

```
   method init();
      cost1 = 1;
      cost2 = 2;
      cost3 = 3;
      cost4 = 4;
      cost5 = 5;
   end;
enddata;
run;
```

Example 3: Using a VARLIST to Create a Local Variable List Causes an Error

In this example, the VARLIST statement causes the following error to appear: `Invalid variable list. vars is not a global scalar variable.` The VARLIST statement cannot be used to create a local variable list.

```
data _null_;
   declare double a;
   varlist vars [a];
   vararray double y[1] vars;  /* Error: varlist variable */
enddata;
run;
```

Example 4: Passing a Variable List to a Function That Accepts a Variable List Argument

The following example creates a method, printNames, that contains a variable list, v. The variable list, v, is passed into the VNAME and VTYPE functions.

```
data _null_;
   vararray double x[5];
   dcl bigint xyz;
   dcl date xanadu;

   method printNames(varlist v);
      dcl varchar(100) name type;
      dcl bigint i;
      do i = 1 to dim(v);
         name = vname(v[i]);
         type = vtype(v[i]);
         put name= type=;
      end;
   end;

   method init();
      printNames([x:]);
   end;
enddata;
run
```

The following lines are written to the SAS log:

```
name=x1 type=double
name=x2 type=double
name=x3 type=double
name=x4 type=double
name=x5 type=double
name=xyz type=bigint
name=xanadu type=date
```

See Also

"Creating Named Variable Lists" in *SAS DS2 Programmer's Guide*

11

DS2 System Methods

Overview of System Methods 1333

Dictionary 1335
INIT Method 1335
RUN Method 1336
SETPARMS Method 1338
TERM Method 1340

Overview of System Methods

Methods are basic program execution units. A method defines a scoping block, so any parameters and any variable declarations in the method body are local to the method. In DS2, all program code must reside in some method.

System methods have a preset meaning in DS2. There are three system methods: INIT, RUN, and TERM. These methods cannot be overloaded. There is one optional system method that is used only with threads: SETPARMS.

The following table lists and summarizes the purpose of each DS2 system method.

System Method	Execution Details	Purpose
INIT( )	Automatically executes one time, as the first method of a program.	As the name implies, INIT( ) is a good place to initialize global program variables. Most global variables are not initialized by the system. However, the system does initialize predefined variables, such as _N and _N_, and variables that are used in Sum and RETAIN statements. If your program does not require the capabilities of the other system methods, you can code your entire program in the INIT( ) method. Just add DS2 statements, including but not limited to the following:

System Method	Execution Details	Purpose
		■ DECLARE statements to create method-scope local variables ■ Calls to one or more user-defined methods ■ DS2 statements that perform variable assignments, call DS2 functions, execute loops or other logic, and so on For more information, see "INIT Method" on page 1335.
RUN( )	Automatically executes after INIT( ) completes.	The RUN( ) method is the functional equivalent of the DATA step. That is, if your RUN( ) method contains a SET statement, the method runs as an implicit loop. You can also use RUN( ) to read rows from a thread program using the SET FROM statement. **Note:** You are not required to include code that leverages the implicit loop capabilities. If appropriate, you can code your entire program in the RUN( ) method, as described for INIT( ). For more information, see "RUN Method" on page 1336.
TERM( )	Automatically executes one time, as the last method of a program.	As the name implies, TERM( ) is where final processing takes place, before the program exits. TERM( ) automatically resets global variables to uninitialized values, with the following exceptions: ■ predefined variables, such as _N and _N_ ■ accumulator variables that were used in Sum statements ■ variables that were used in a RETAIN statement ■ package variables If appropriate, you can code your entire program in the TERM( ) method, as described for INIT( ). For more information, see "TERM Method" on page 1340.
SETPARMS( )	Executes one time, when called from a data program, to initialize the values of a parameterized thread.	SETPARMS( ) initializes the values of a parameterized thread. Because only parameterized thread programs require this, SETPARMS( ) is the only system method that must be called. **Note:** Do not write a SETPARMS( ) method in your thread program. The system supplies the method for you. **Note:** Do not call SETPARMS( ) more than once. The initialization works only the first time. For more information, see "SETPARMS Method" on page 1338.

For complete information about how methods work in DS2, see "Methods" in *SAS DS2 Programmer's Guide*.

Dictionary

INIT Method

Calls a DS2 system method where program initializations can take place.

Syntax

METHOD INIT( );
END;

Without Arguments

The METHOD INIT statement has no arguments. If you try to pass arguments, an error occurs.

Details

Typically, the INIT method contains any initialization code such as variable initialization or opening of tables. Code in the INIT method runs once at the beginning of the DS2 program.

Every DS2 program contains, either implicitly or explicitly, the INIT, RUN, and TERM methods. If you do not specify a METHOD INIT statement, DS2 automatically provides one.

For more information about the INIT method and how DS2 programs work, see "Methods" in *SAS DS2 Programmer's Guide*.

Example

```
method init();
   dcl int i;
   dcl double d;
   d = 99;
   do i = 1 to 3;
      d = d + i;
      output d;
   end;
```

```
end;
```

See Also

- "Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "RUN Method" on page 1336
- "TERM Method" on page 1340

RUN Method

Calls a DS2 system method where DS2 program code can run in an implicit loop.

Syntax

METHOD RUN( );
END;

Without Arguments

The METHOD RUN statement has no arguments. If you try to pass arguments, an error occurs.

Details

Typically, the RUN method contains the main DS2 program code. The RUN method has the same feature of automatic, implicit looping as the SAS DATA step. After the RUN method has been executed one time, the RUN method either runs again or control is passed to the TERM method.

Every DS2 program contains, either implicity or explicitly, the INIT, RUN, and TERM methods. If you do not specify a METHOD INIT or METHOD TERM statement, DS2 automatically provides one.

After the INIT method runs and before the RUN method is executed, variables in the program data vector, which have not been retained (by using the RETAIN statement), are set to either SAS missing values or null values depending on whether you are in SAS mode or ANSI mode.

Local variables in the RUN method completely cease to exist between invocations of RUN in the implicit loop. For each invocation of the RUN method, all local

variables are constructed at the start of execution of the method and destroyed at end of execution of the method. All global variables, except column variables read by the SET statement, are set to missing or null between each iteration (RUN method invocation) of the implicit loop. To retain state data through the implicit loop, you must create a global variable AND also specify that the variable's value be retained across executions of the RUN method with the RETAIN statement. The RUN method is executed *x*+1 times for a table with *x* rows. If a SET statement is executed and finds no more rows, then the implicit looping of the RUN method ceases.

For more information about SAS and ANSI mode, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.

For more information about the RUN method and how DS2 programs work, see "Methods" in *SAS DS2 Programmer's Guide*.

Comparisons

In DATA step programming, the entire DATA step represents the implicit loop. In the DS2 language, the implicit loop is represented by the RUN method.

Example

DS2's flow of execution is to call the INIT method once, then the RUN method until the input tables are completely read, then the TERM method. The RUN method is where the implicit loop exists. The following program demonstrates this flow of control by finding the minimum of values in a table. The INIT method initializes the columns used to find the current minimum, the RUN method compares input values with the current minimum, and the TERM method writes the minimums to an output table.

```
data xy_data;
   dcl double x y;
   method init();
      do x = 1 to 5;
         y = 2*x;
         output;
      end;
   end;
enddata;
run;
/* Find the minimum value for x and y */
data xy_mins;
   dcl double min_x min_y;
   retain min_x min_y;
   keep min_x min_y;
   method init();
      min_x = 999999;
      min_y = 999999;
   end;
   method run();
```

```
      set xy_data;
      if x < min_x then min_x = x;
      if y < min_y then min_y = y;
   end;
   method term();
      output;
   end;
enddata;
run;
/* Send result table of minimums to output */
data;
    method run();
       set xy_mins;
    end;
enddata;
run;
```

See Also

- "Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "INIT Method" on page 1335
- "TERM Method" on page 1340

SETPARMS Method

Initializes parameters for an instance of a DS2 thread.

Restrictions: The SETPARMS method cannot be used with the SAS In-Database Code Accelerator.

This method is not supported on the CAS server.

Syntax

thread.**SETPARMS**(*parameter-value*[, ...*parameter-value*]);

Required Arguments

thread
specifies an instance of the thread.

parameter-value
specifies the initial value of the thread parameter.

Details

When using parameterized threads, the parameter names and their types are specified in the THREAD statement. The DS2 program that invokes the thread must initialize the thread's parameters by calling the SETPARMS method. In this example, assume you have an instance of the thread, T, that takes two parameters, INV and PROD.

```
thread work.t (double inv, char (30) prod);
```

Using the SETPARMS method, the parameter INV is initialized with a value of *38824* and the parameter PROD is initialized with a value of *'rice'*.

```
t.setparms(38824, 'rice');
```

Each argument is passed by value to the corresponding thread parameter. All arguments are converted, if necessary, to the data type of the corresponding parameter. If the SETPARMS method is called for a thread, which has no parameters, an error occurs.

The SETPARMS method must be called to initialize parameters for a thread before the thread's SET FROM statement executes, or the parameters initialize with SAS missing values or null values, depending on whether you are in SAS mode or ANSI mode. For more information, see "How DS2 Processes Nulls and SAS Missing Values" in *SAS DS2 Programmer's Guide*.

Example

This example illustrates how to use threads with parameters.

```
thread work.t (double d, char (100) sp);
   dcl int x;
   dcl double y;
   dcl nchar(20) s;
   dcl char(30) c;
   method init();
      dcl int i;
      s = 'abc' || sp;
      c = 'uvwxyz' || sp;
      do i = 1 to 100;
         x = i;
         y = i * 2.5 + d;
         output;
      end;
    end;
endthread;
run;
data;
   dcl thread work.t t;
   method init();
      t.setparms(99, 'ijk');
   end;
   method run();
      set from t;
```

```
      answer = x + y;
      put 's= ' s ' x= ' x ' y= ' y ' c= ' c ' answer= ' answer;
   end;
enddata;
run;
```

This is a partial listing of lines that are written to the SAS log:

```
s=  abcijk     x=  1   y=  101.5   c=  uvwxyzijk   answer=  102.5
s=  abcijk     x=  2   y=  104     c=  uvwxyzijk   answer=  106
s=  abcijk     x=  3   y=  106.5   c=  uvwxyzijk   answer=  109.5
s=  abcijk     x=  4   y=  109     c=  uvwxyzijk   answer=  113
s=  abcijk     x=  5   y=  111.5   c=  uvwxyzijk   answer=  116.5
```

See Also

Statements:

- “SET FROM Statement” on page 1310
- “THREAD Statement” on page 1318

TERM Method

Calls a DS2 system method where program finalizations can take place.

Syntax

METHOD TERM ( );

END;

Without Arguments

The METHOD TERM statement has no arguments. If you try to pass arguments, an error occurs.

Details

Typically, the TERM method contains any finalization code such as writing data to the SAS log. Code in the TERM method runs once at the end of the DS2 program.

Every DS2 program contains, either implicitly or explicitly, the INIT, RUN, and TERM methods. If you do not specify a METHOD TERM statement, DS2 automatically provides one.

For more information about the TERM method and how DS2 programs work, see "Methods" in *SAS DS2 Programmer's Guide*.

See Also

- "Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "INIT Method" on page 1335
- "RUN Method" on page 1336

12

DS2 System Options

Overview of System Options 1343

Dictionary 1344
DS2ACCEL= System Option 1344
DS2SCOND= System Option 1345

Overview of System Options

System options are instructions that affect the processing of an entire SAS program or interactive SAS session from the time the option is specified until it is changed.

Here is the syntax for specifying system options in an OPTIONS statement:

OPTIONS *options(s)*;

Here is an explanation of the syntax:

option
specifies one or more SAS system options that you want to change.

The following example shows how to use the system option DS2SCOND in an OPTIONS statement.

```
options ds2scond=none;
```

Note: DS2 does not support Base SAS system options.

Dictionary

DS2ACCEL= System Option

Specifies whether DS2 code is enabled for parallel processing in supported environments using the SAS In-Database Code Accelerator.

Valid in:	Configuration file, SAS invocation, OPTIONS statement, SAS System Options window, Greenplum, Hadoop, and Teradata
PROC OPTIONS GROUP=	PERFORMANCE
Default:	NONE
Restriction:	This system option is not supported on the CAS server.
Note:	This option can be restricted by a site administrator. For more information, see "Restricted Options" in *SAS System Options: Reference*.

Syntax

DS2ACCEL=ANY | NONE

Required Arguments

ANY
enables DS2 code to execute in supported parallel environments.

NONE
disables DS2 code from executing in supported parallel environments.

Details

The SAS In-Database Code Accelerator enables you to publish a DS2 thread program to the database and execute the thread program in parallel inside the database. If you are using the SAS In-Database Code Accelerator for Teradata or Hadoop, the DS2 data program is also published and executed inside the database.

The DS2ACCEL= system option controls whether DS2 code is executed inside the database.

You can override the DS2ACCEL= system option by specifying the DS2ACCEL= option in the PROC DS2 statement.

See Also

- "Using the DS2ACCEL Option to Control In-Database Processing" in *SAS DS2 Programmer's Guide*

Procedures:

- "DS2 Procedure" in *Base SAS Procedures Guide*

System Options:

- "DSACCEL= System Option" in *SAS System Options: Reference*

DS2SCOND= System Option

Specifies the level of messages that PROC DS2 displays in the SAS log for the DS2 variable declaration strict mode, which requires that every variable must be declared in the DS2 program.

Valid in:	Configuration file, SAS invocation, OPTIONS statement, SAS System Options window
PROC OPTIONS GROUP=	ErrorHandling
Note:	This option can be restricted by a site administrator. For more information, see "Restricted Options" in *SAS System Options: Reference*.

Syntax

DS2SCOND=ERROR | NONE | NOTE | WARNING

Required Arguments

ERROR
writes error messages to the SAS log.

Alias ERR

NONE
no messages are written to the SAS log.

NOTE
writes notes to the SAS log.

WARNING
writes warning messages to the SAS log. This is the default.

Alias WARN

Details

You can override the DS2SCOND system option by specifying the SCOND= option in the PROC DS2 statement.

See Also

- "Variable Declaration" in *SAS DS2 Programmer's Guide*

Procedures:

- "DS2 Procedure" in *Base SAS Procedures Guide*

13

DS2 Table Options

Overview of Table Options ... 1349

Using Table Options in DS2 ... 1350

How to Specify Table Options in DS2 ... 1351

DS2 Table Option Examples ... 1351

DS2 Statement Table Options by Data Source ... 1352

Dictionary ... 1364

- ALTER= Table Option ... 1364
- ASYNCINDEX= Table Option ... 1365
- BL_ACCOUNTNAME= Table Option ... 1366
- BL_ALLOW_READ_ACCESS= Table Option ... 1367
- BL_APPLICATIONID= Table Option ... 1368
- BL_AZURE_SAS= Table Option ... 1369
- BL_AZURE_TENANTID= Table Option ... 1370
- BL_COMPRESS= Table Option ... 1371
- BL_COPY_LOCATION= Table Option ... 1372
- BL_CPU_PARALLELISM= Table Option ... 1373
- BL_DATA_BUFFER_SIZE= Table Option ... 1374
- BL_DEFAULT_DIR= Table Option ... 1375
- BL_DELETE_DATAFILE= Table Option ... 1376
- BL_DELIMITER= Table Option ... 1377
- BL_DISK_PARALLELISM= Table Option ... 1378
- BL_ERRORS= Table Option ... 1379
- BL_EXCEPTION= Table Option ... 1379
- BL_FILESYSTEM= Table Option ... 1380
- BL_FOLDER= Table Option ... 1381
- BL_INDEXING_MODE= Table Option ... 1382
- BL_INTERNAL_STAGE= Table Option ... 1383
- BL_LOAD= Table Option ... 1384
- BL_LOAD_REPLACE= Table Option ... 1385
- BL_LOG= Table Option ... 1385
- BL_LOGFILE= Table Option ... 1386
- BL_NUM_DATAFILES= Table Option ... 1387
- BL_OPTIONS= Table Option ... 1388
- BL_PARALLEL Table Option ... 1389
- BL_PORT_MAX= Table Option ... 1389

BL_PORT_MIN= Table Option ... 1390
BL_RECOVERABLE= Table Option ... 1391
BL_REMOTE_FILE= Table Option ... 1392
BL_SKIP= Table Option ... 1393
BL_SKIP_INDEX_MAINTENANCE= Table Option ... 1393
BL_SKIP_UNUSABLE_INDEXES= Table Option ... 1394
BL_TIMEOUT= Table Option ... 1395
BL_USE_ESCAPE= Table Option ... 1396
BL_WARNING_COUNT= Table Option ... 1397
BUFNO= Table Option ... 1398
BUFSIZE= Table Option ... 1399
BULKLOAD= Table Option ... 1400
BULKOPTS= Table Option ... 1403
COMPRESS= Table Option ... 1406
CONFIG= Table Option ... 1408
DATAFILE= Table Option ... 1409
DBCREATE_INDEX_OPTS= Table Option ... 1410
DBCREATE_TABLE_OPTS= Table Option ... 1411
DROP= Table Option ... 1413
ENCRYPT= Table Option ... 1415
ENCRYPTKEY= Table Option ... 1418
ENDOBS= Table Option ... 1421
EXTENDOBSCOUNTER= Table Option ... 1422
FETCH_NUMERIC_TYPE= Table Option ... 1424
GP_DISTRIBUTED_BY= Table Option ... 1424
IDXNAME= Table Option ... 1425
IDXWHERE= Table Option ... 1427
IN= Table Option ... 1429
INLINE Table Option ... 1430
IOBLOCKSIZE= Table Option ... 1433
KEEP= Table Option ... 1434
LABEL= Table Option ... 1436
LOCKTABLE= Table Option ... 1437
ORHINTS= Table Option ... 1437
ORNUMERIC= Table Option ... 1438
OVERWRITE= Table Option ... 1439
PADCOMPRESS= Table Option ... 1441
PARTITION_KEY= Table Option ... 1442
PARTSIZE= Table Option ... 1443
PASSWORD= Table Option ... 1444
PICKLIST= Table Option ... 1445
POINTOBS= Table Option ... 1445
POST_TABLE_OPTS= Table Option ... 1447
PRE_TABLE_OPTS= Table Option ... 1448
PW= Table Option ... 1449
READ= Table Option ... 1450
READMODE= Table Option ... 1451
RENAME= Table Option ... 1452
REUSE= Table Option ... 1454
RS_AWS_CONFIG= Table Option ... 1455
RS_BUCKET= Table Option ... 1456
RS_CREDENTIALS_FILE= Table Option ... 1456
RS_DEFAULT_DIR= Table Option ... 1457
RS_DELETE_DATAFILE= Table Option ... 1458

RS_KEY= Table Option ... 1458
RS_REGION= Table Option ... 1459
RS_SECRET= Table Option ... 1460
RS_TOKEN= Table Option ... 1460
STARTOBS= Table Option ... 1461
TABLE_TYPE= Table Option ... 1462
TD_BUFFER_MODE= Table Option ... 1463
TD_CHECKPOINT= Table Option ... 1464
TD_DATA_ENCRYPTION= Table Option ... 1464
TD_DROP_ERROR_TABLE= Table Option ... 1465
TD_DROP_LOG_TABLE= Table Option ... 1466
TD_DROP_WORK_TABLE= Table Option ... 1467
TD_ERROR_LIMIT= Table Option ... 1467
TD_ERROR_TABLE_1= Table Option ... 1468
TD_ERROR_TABLE_2= Table Option ... 1469
TD_LOG_MECH_DATA= Table Option ... 1470
TD_LOG_MECH_TYPE= Table Option ... 1471
TD_LOG_TABLE= Table Option ... 1472
TD_LOGDB= Table Option ... 1473
TD_MAX_SESSIONS= Table Option ... 1474
TD_MIN_SESSIONS= Table Option ... 1474
TD_NOTIFY_LEVEL= Table Option ... 1475
TD_NOTIFY_METHOD= Table Option ... 1476
TD_NOTIFY_STRING= Table Option ... 1477
TD_PACK= Table Option ... 1478
TD_PACK_MAXIMUM= Table Option ... 1479
TD_PAUSE_ACQ= Table Option ... 1479
TD_SESSION_QUERY_BAND= Table Option ... 1480
TD_TENACITY_HOURS= Table Option ... 1481
TD_TENACITY_SLEEP= Table Option ... 1481
TD_TPT_OPER= Table Option ... 1482
TD_TRACE_LEVEL=, TD_TRACE_LEVEL_INF= Table Options ... 1483
TD_TRACE_OUTPUT= Table Option ... 1485
TD_UNICODE_PASSTHRU= Table Option ... 1485
TD_WORK_TABLE= Table Option ... 1486
TD_WORKING_DB= Table Option ... 1487
THREADNUM= Table Option ... 1488
TYPE= Table Option ... 1489
UNIQUESAVE= Table Option ... 1490
USER= Table Option ... 1491
WHERENOINDEX= Table Option ... 1492
WRITE= Table Option ... 1493

Overview of Table Options

Table options are analogous to data set options in DATA step programming.

Table options specify actions that apply only to the tables with which they appear. These are some of the operations that table options enable you to perform:

- rename variables
- specify passwords
- specify options for bulk loading data
- drop variables from processing or from the result table

Note: Some table options apply to all data sources. Others are data source specific. Table options that are not recognized by DS2 are passed without error to the underlying table driver.

Using Table Options in DS2

Table options can be used on these DS2 statements:

- DECLARE PACKAGE
- DECLARE THREAD
- DROP PACKAGE
- DROP THREAD
- PACKAGE
- SET
- DATA
- THREAD

Some table options can apply to packages and threads.

Most table options can apply to either input or output tables. If a table option is associated with an input table, the action applies to the table that is being read. If the option appears in the DATA statement, SAS applies the action to the output table. In DS2, table options for output tables must appear in the DATA statement, not in any OUTPUT statements that might be present.

Some table options, such as COMPRESS=, are meaningful only when you create a SAS data set because they set attributes that exist for the duration of the data set. To change or cancel most table options, you must re-create the table.

When table options appear in both input and output tables in the same DS2 program, first SAS applies table options to input tables. Then SAS evaluates programming statements or applies table options to output tables. Likewise, table options that are specified for the table being created are applied after programming statements are processed. For example, when using the RENAME= table option, the new names are not associated with the columns until the DS2 program is compiled.

In some instances, table options conflict when they are used in the same statement. For example, you cannot specify both the DROP= and KEEP= table options for the same variable in the same statement. Timing can also be an issue in

some cases. For example, if you are using KEEP= and RENAME= in a table that is specified in the SET statement, KEEP= must use the original column names. SAS processes KEEP= before the table is read. The new names that are specified in RENAME= apply to the programming statements that follow the SET statement.

Table options are applicable whenever you are reading or writing a table that contains data. Therefore, table options work with DS2 packages and threads because packages and threads are stored in tables.

How to Specify Table Options in DS2

Table options are either enclosed in parentheses or preceded by a forward slash (/), depending on which statement they are used in. Table options should be placed at the end of the statement when they are preceded by the forward slash.

Table options are enclosed in parentheses when used in these statements:

- DECLARE PACKAGE
- DECLARE THREAD
- DROP PACKAGE
- DROP THREAD
- SET
- DATA

If the table option is enclosed in parentheses and the option value can be several items separated by spaces, the option values are also enclosed in parentheses. For examples, see “DS2 Table Option Examples” on page 1351.

Table options are preceded by a forward slash (/) when used in these statements:

- PACKAGE
- THREAD

DS2 Table Option Examples

Here are some examples of table options that are enclosed in parentheses:

```
data a (bufno=10);

data prod (drop=(price sales));

declare package sales (pw=lk34890f);
```

```
drop thread complex (write=24klj);
```

Here are some examples of table options that are preceded by a forward slash (/):

```
package invent /overwrite=yes;

thread work.t (double d, char (100) sp) /read=44kl7;
```

DS2 Statement Table Options by Data Source

The following table lists the table options that are supported in DS2 programs by the data source that supports them. The options are listed alphabetically for each data source.

Table 13.1 *List of Supported Table Options by Data Source*

Data Source	Language Element	Category	Description
All	"DROP= Table Option" on page 1413	Table control	For an input table, excludes the specified columns from processing; for an output table, excludes the specified columns from being written to the table.
	"IN= Table Option" on page 1429	Table control	Creates an integer variable that indicates whether the table contributed data to the current row.
	"INLINE Table Option" on page 1430	Table control	Specifies that the package or thread source code is not saved to a table for reuse and is validated and compiled only when loaded by a data program.
	"KEEP= Table Option" on page 1434	Table control	For an input table, specifies the columns to process; for an output table, specifies the columns to write to the table.
	"OVERWRITE= Table Option" on page 1439	Table control	For a table, drops the output table before the replacement output table is populated with rows; for packages and threads, drops the existing package or thread if a

Data Source	Language Element	Category	Description
			package or thread by the same name exists.
	"RENAME= Table Option" on page 1452	Table control	Changes the name of a column.
Amazon Redshift	"BULKLOAD= Table Option"	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
Aster	"PARTITION_KEY= Table Option" on page 1442	Table control	Specifies the column name to use as the partition key for creating fact tables.
DB2 under UNIX and PC	"BL_ALLOW_READ_ACCESS= Table Option" on page 1367	Bulk loading	Specifies that the original table data is still visible to readers during bulk load.
	"BL_COPY_LOCATION= Table Option" on page 1372	Bulk loading	Specifies the directory to which DB2 saves a copy of the loaded data.
	"BL_CPU_PARALLELISM= Table Option" on page 1373	Bulk loading	Specifies the number of processes or threads to use when building table objects.
	"BL_DATA_BUFFER_SIZE= Table Option" on page 1374	Bulk loading	Specifies the total amount of memory to allocate for the bulk load utility to use as a buffer for transferring data.
	"BL_DISK_PARALLELISM= Table Option" on page 1378	Bulk loading	Specifies the number of processes or threads to use when writing data to disk.
	"BL_EXCEPTION= Table Option" on page 1379	Bulk loading	Specifies the exception table into which rows in error are copied.
	"BL_INDEXING_MODE= Table Option" on page 1382	Bulk loading	Specifies which scheme the DB2 load utility should use for index maintenance.

Data Source	Language Element	Category	Description
	"BL_LOAD_REPLACE= Table Option" on page 1385	Bulk loading	Specifies whether DB2 appends or replaces rows during bulk loading.
	"BL_LOG= Table Option" on page 1385	Bulk loading	Identifies a log file that contains information such as statistics and error information for a bulk load.
	"BL_OPTIONS= Table Option" on page 1388	Bulk loading	Passes options to the DBMS bulk-load facility, affecting how it loads and processes data.
	"BL_PORT_MAX= Table Option" on page 1389	Bulk loading	Sets the highest available port number for concurrent uploads.
	"BL_PORT_MIN= Table Option" on page 1390	Bulk loading	Sets the lowest available port number for concurrent uploads.
	"BL_RECOVERABLE= Table Option" on page 1391	Bulk loading	Specifies whether the LOAD process is recoverable.
	"BL_REMOTE_FILE= Table Option" on page 1392	Bulk loading	Specifies the base filename and location of DB2 LOAD temporary files.
	"BL_WARNING_COUNT= Table Option" on page 1397	Bulk loading	Specifies the maximum number of row warnings to allow before the load fails.
	"BULKLOAD= Table Option" on page 1400	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
Google BigQuery	"BULKLOAD= Table Option" on page 1400	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific options to be added to the DATA statement.

Data Source	Language Element	Category	Description
	"FETCH_NUMERIC_TYPE= Table Option" on page 1424	Table control	Specifies how to handle NUMERIC data from Google BigQuery.
Greenplum	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
	"GP_DISTRIBUTED_BY= Table Option" on page 1424	Table control	Specifies the distribution key for the table being created.
HAWQ	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
	"GP_DISTRIBUTED_BY= Table Option" on page 1424	Table control	Specifies the distribution key for the table being created.
Hive	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows database-specific options to be placed after the DATA statement.
	"POST_TABLE_OPTS= Table Option" on page 1447	Table control	Allows database-specific options to be placed after the table name in a DATA statement.
	"PRE_TABLE_OPTS= Table Option" on page 1448	Table control	Allows database-specific options to be placed before the table name in a DATA statement.
Impala	"BULKLOAD= Table Option" on page 1400	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"CONFIG= Table Option" on page 1408	Bulk loading	Specifies a file or path name for Hadoop configuration path resolution.
	"DATAFILE= Table Option" on page 1409	Bulk loading	Specifies an alternate name and location for the temporary HDFS file.
	"PASSWORD= Table Option" on page 1444	Bulk loading	Specifies the password for the HDFS user.
	"PICKLIST= Table Option" on page 1445	Bulk loading	Specifies the picklist to use for the bulk-loading operation.

Data Source	Language Element	Category	Description
	"USER= Table Option" on page 1491	Bulk loading	Specifies the HDFS user name.
MDS	"BULKLOAD= Table Option" on page 1400	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
Microsoft SQL Server	"BULKLOAD= Table Option" on page 1400	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"DBCREATE_INDEX_OPTS = Table Option" on page 1410	Table control	Allows DBMS-specific syntax to be added to the CREATE INDEX statement.
	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
MySQL	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
Netezza	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows additional DBMS-specific syntax to be added to the DATA statement.
ODBC	"DBCREATE_INDEX_OPTS = Table Option" on page 1410	Index control	Allows DBMS-specific syntax to be added to the CREATE INDEX statement.
	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
Oracle	"BL_DEFAULT_DIR= Table Option" on page 1375	Bulk loading	Specifies where bulk load creates all intermediate files.
	"BL_ERRORS= Table Option" on page 1379	Bulk loading	Specifies that, after the indicated number of errors is received, the load should stop.
	"BL_LOAD= Table Option" on page 1384	Bulk loading	Specifies that, after the indicated number of rows is loaded, the load should stop.
	"BL_LOGFILE= Table Option" on page 1386	Bulk loading	Specifies the filename for the bulk load log file.

Data Source	Language Element	Category	Description
	"BL_PARALLEL Table Option" on page 1389	Bulk loading	Specifies whether to perform a parallel bulk load.
	"BL_RECOVERABLE= Table Option" on page 1391	Bulk loading	Determines whether the LOAD process is recoverable.
	"BL_SKIP= Table Option" on page 1393	Bulk loading	Specifies to skip the indicated number of rows before starting the bulk load.
	"BL_SKIP_INDEX_MAINTENANCE= Table Option" on page 1393	Bulk loading	Specifies whether to perform index maintenance on the bulk load.
	"BL_SKIP_UNUSABLE_INDEXES= Table Option" on page 1394	Bulk loading	Specifies whether to skip index entries that are in an unusable state and continue with the bulk load.
	"BULKLOAD= Table Option" on page 1400	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"DBCREATE_TABLE_OPTS= Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
	"ORHINTS= Table Option" on page 1437	Data control	Specifies Oracle hints to pass to Oracle from FedSQL.
	"ORNUMERIC= Table Option" on page 1438	Table control	Specifies how numbers read from or inserted into the Oracle NUMBER column are treated.
SAP HANA	"DBCREATE_TABLE_OPTS= Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
	"TABLE_TYPE= Table Option" on page 1462	Table access	Specifies the type of table storage FedSQL uses when creating tables in SAP HANA.
SAS data set	"ALTER= Table Option" on page 1364	Table control	Assigns an ALTER password to a SAS data set that prevents users from replacing or deleting the file, and enables access to a read- or write-protected file.

Data Source	Language Element	Category	Description
	"BUFNO= Table Option" on page 1398	Table control	Specifies the number of buffers to be allocated for processing a SAS data set.
	"BUFSIZE= Table Option" on page 1399	Table control	Specifies the size of a permanent buffer page for an output SAS data set.
	"COMPRESS= Table Option" on page 1406	Table control	Specifies how rows are compressed in a new output data set.
	"ENCRYPT= Table Option" on page 1415	Table control	Specifies whether to encrypt an output SAS data set.
	"ENCRYPTKEY= Table Option" on page 1418	Table control	Specifies a key value for AES encryption.
	"EXTENDOBSCOUNTER= Table Option" on page 1422	Table control	Specifies whether to extend the maximum observation count in a new output SAS data file.
	"IDXNAME= Table Option" on page 1425	User control of index usage	Directs SAS to use a specific index to match the conditions of a WHERE clause.
	"IDXWHERE= Table Option" on page 1427	User control of index usage	Specifies whether SAS uses an index search or a sequential search to match the conditions of a WHERE clause.
	"LABEL= Table Option" on page 1436	Observation control	Specifies a label for a SAS data set.
	"LOCKTABLE= Table Option" on page 1437	Table control	Places shared or exclusive locks on tables.
	"POINTOBS= Table Option" on page 1445	Table control	Specifies whether SAS creates compressed data sets whose observations can be randomly accessed or sequentially accessed.
	"PW= Table Option" on page 1449	Table control	Assigns a READ, WRITE, and ALTER password to a SAS file, and enables access to a password-protected file.
	"READ= Table Option" on page 1450	Table control	Assigns a READ password to a SAS file that prevents users from reading

Data Source	Language Element	Category	Description
			the file, unless they enter the password.
	"REUSE= Table Option" on page 1454	Table control	Specifies whether new rows can be written to freed space in a compressed SAS data set.
	"TYPE= Table Option" on page 1489	Table control	Specifies the data set type for a specially structured SAS data set.
	"WRITE= Table Option" on page 1493	Table control	Assigns a WRITE password to a SAS file that prevents users from writing to the file or that enables access to a write-protected file.
Snowflake	"BULKLOAD= Table Option"	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
SPD Engine data set	"ALTER= Table Option" on page 1364	Table control	Assigns an ALTER password to an SPD Engine data set that prevents users from replacing or deleting the file, and enables access to a read- or write-protected file.
	"ASYNCINDEX= Table Option" on page 1365	User control of index usage	Specifies to create indexes in parallel when creating multiple indexes on an SPD Engine data set.
	"COMPRESS= Table Option" on page 1406	Table control	Specifies to compress SPD Engine data sets on disk as they are being created.
	"ENCRYPT= Table Option" on page 1415	Table control	Specifies whether to encrypt an output SPD Engine data set.
	"ENCRYPTKEY= Table Option" on page 1418	Table control	Specifies a key value for AES encryption.
	"ENDOBS= Table Option" on page 1421	Observation control	Specifies the end observation number in a user-defined range of observations to be processed.
	"IDXWHERE= Table Option" on page 1427	User control of index usage	Specifies to use indexes when processing WHERE expressions in the SPD Engine.

Data Source	Language Element	Category	Description
	"IOBLOCKSIZE= Table Option" on page 1433	Table control	Specifies the size in bytes of a block of observations to be used in an I/O operation.
	"LABEL= Table Option" on page 1436	Observation control	Specifies a label for an SPD Engine data set.
	"PADCOMPRESS= Table Option" on page 1441	Table control	Specifies the number of bytes to add to compressed blocks in a data set opened for OUTPUT or UPDATE.
	"PARTSIZE= Table Option" on page 1443	Table control	Specifies the size of the data component partitions in an SPD Engine data set.
	"PW= Table Option" on page 1449	Table control	Assigns a READ, WRITE, and ALTER password to a SAS file, and enables access to a password-protected file.
	"READ= Table Option" on page 1450	Table control	Assigns a READ password to a SAS file that prevents users from reading the file, unless they enter the password.
	"STARTOBS= Table Option" on page 1461	Observation control	Specifies the starting observation number in a user-defined range of observations to be processed.
	"THREADNUM= Table Option" on page 1488	Table control	Specifies the maximum number of I/O threads the SPD Engine can spawn for processing an SPD Engine data set.
	"TYPE= Table Option" on page 1489	Table control	Specifies the data set type for a specially structured SPD Engine data set.
	"UNIQUESAVE= Table Option" on page 1490	User control of index usage	Specifies to save observations with nonunique key values (the rejected observations) to a separate data set when appending or inserting observations to data sets with unique indexes.
	"WHERENOINDEX= Table Option" on page 1492	User control of index usage	Specifies a list of indexes to exclude when making WHERE expression evaluations.

Data Source	Language Element	Category	Description
	"WRITE= Table Option" on page 1493	Table control	Assigns a WRITE password to a SAS file that prevents users from writing to the file or that enables access to a write-protected file.
SPD Server table	"COMPRESS= Table Option" on page 1406	Table control	Specifies to compress SPD Server tables on disk as they are being created.
	"ENCRYPT= Table Option" on page 1415	Table access	Specifies whether to encrypt an output SPD Server table.
	"ENCRYPTKEY= Table Option" on page 1418	Table access	Specifies a key for AES encryption.
	"ENDOBS= Table Option" on page 1421	Observation control	Specifies the end observation number in a user-defined range of observations to be processed.
	"IOBLOCKSIZE= Table Option" on page 1433	Table control	Specifies the size in bytes of a block of rows to be used in an I/O operation.
	"PARTSIZE= Table Option" on page 1443	Table control	Specifies the size of the data component partitions in an SPD Server table.
	"PW= Table Option" on page 1449	Table access	Enables you to access an SPD Server table that is protected by SAS Proprietary encryption.
	"STARTOBS= Table Option" on page 1461	Observation control	Specifies the starting observation number in a user-defined range of observations to be processed.
Spark	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows database-specific options to be placed after the DATA statement.
	"POST_TABLE_OPTS= Table Option" on page 1447	Table control	Allows database-specific options to be placed after the table name in a DATA statement.
	"PRE_TABLE_OPTS= Table Option" on page 1448	Table control	Allows database-specific options to be placed before the table name in a DATA statement.

Data Source	Language Element	Category	Description
Teradata	"BULKLOAD= Table Option" on page 1400	Bulk loading	Determines whether SAS uses a DBMS facility to insert data into a DBMS table.
	"DBCREATE_TABLE_OPTS = Table Option" on page 1411	Table control	Allows DBMS-specific syntax to be added to the DATA statement.
	"TD_BUFFER_MODE= Table Option" on page 1463	Bulk loading	Specifies whether the LOAD method is used.
	"TD_CHECKPOINT= Table Option" on page 1464	Bulk loading	Specifies when the TPT operation issues a checkpoint or savepoint to the database.
	"TD_DATA_ENCRYPTION= Table Option" on page 1464	Bulk loading	Activates data encryption.
	"TD_DROP_LOG_TABLE= Table Option" on page 1466	Bulk loading	Drops the log table at the end of the job, whether the job completed successfully or not.
	"TD_DROP_ERROR_TABLE = Table Option" on page 1465	Bulk loading	Drops the error tables at the end of the job, whether the job completed successfully or not.
	"TD_DROP_WORK_TABLE = Table Option" on page 1467	Bulk loading	Drops the work table at the end of the job, whether the job completed successfully or not.
	"TD_ERROR_LIMIT= Table Option" on page 1467	Bulk loading	Specifies the maximum number of records that can be stored in an error table.
	"TD_ERROR_TABLE_1= Table Option" on page 1468	Bulk loading	Specifies a name for the first error table.
	"TD_ERROR_TABLE_2= Table Option" on page 1469	Bulk loading	Specifies a name for the second error table.
	"TD_LOG_TABLE= Table Option" on page 1472	Bulk loading	Specifies the name of the restart log table.

Data Source	Language Element	Category	Description
	"TD_LOG_MECH_TYPE= Table Option" on page 1471	Bulk loading	Specifies the logon mechanism for a bulk load.
	"TD_LOG_MECH_DATA= Table Option" on page 1470	Bulk loading	Specifies additional data for the logon mechanism.
	"TD_LOGDB= Table Option" on page 1473	Bulk loading	Specifies the database where the TPT utility tables are created.
	"TD_MAX_SESSIONS= Table Option" on page 1474	Bulk loading	Specifies the maximum number of logon sessions that TPT can acquire for a job.
	"TD_MIN_SESSIONS= Table Option" on page 1474	Bulk loading	Specifies the minimum number of sessions for TPT to acquire before a job starts.
	"TD_NOTIFY_LEVEL= Table Option" on page 1475	Bulk loading	Specifies the level at which log events are recorded.
	"TD_NOTIFY_METHOD= Table Option" on page 1476	Bulk loading	Specifies the method for reporting events.
	"TD_NOTIFY_STRING= Table Option" on page 1477	Bulk loading	Defines a string that precedes all messages sent to the system log.
	"TD_PACK= Table Option" on page 1478	Bulk loading	Specifies the number of statements to pack into a multistatement request.
	"TD_PACK_MAXIMUM= Table Option" on page 1479	Bulk loading	Enables the Stream operator to determine the maximum possible pack factor for the current Stream job.
	"TD_PAUSE_ACQ= Table Option" on page 1479	Bulk loading	Forces a pause between the acquisition phase and the application phase of a load job.
	"TD_SESSION_QUERY_BAND= Table Option" on page 1480	Bulk loading	Passes a string of user-specified name=value pairs for use by the TPT session.

Data Source	Language Element	Category	Description
	"TD_TENACITY_HOURS= Table Option" on page 1481	Bulk loading	Specifies the amount of time the TPT operator continues trying to log on to the Teradata database.
	"TD_TENACITY_SLEEP= Table Option" on page 1481	Bulk loading	Specifies the amount of time the TPT operator pauses, before retrying to log on to the Teradata database.
	"TD_TPT_OPER= Table Option" on page 1482	Bulk loading	Specifies the load operator used by the Teradata Parallel Transporter.
	"TD_TRACE_LEVEL=, TD_TRACE_LEVEL_INF= Table Options" on page 1483	Bulk loading	Specify the trace levels for driver tracing. TD_TRACE_LEVEL sets the primary trace level. TD_TRACE_LEVEL_INF sets the secondary trace level.
	"TD_TRACE_OUTPUT= Table Option" on page 1485.	Bulk loading	Specifies the name of the external file used for trace messages.
	"TD_UNICODE_PASSTHRU= Table Option" on page 1485	Bulk loading	Allows pass-through characters (PTCs) to be imported into and exported from Teradata.
	"TD_WORKING_DB= Table Option" on page 1487	Bulk loading	Specifies the database where the table is to be created.
	"TD_WORK_TABLE= Table Option" on page 1486	Bulk loading	Specifies a name for the TPT work table.

Dictionary

ALTER= Table Option

Assigns an ALTER password to a data set that prevents users from replacing or deleting the file, and enables access to a Read- and Write-protected file.

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts*.

Syntax

ALTER==*alter-password*

Required Argument

alter-password
must be a valid SAS name.

Details

The ALTER= option applies only to a SAS data set. You can use this option to assign a password or to access a read-protected, write-protected, or alter-protected file. When you replace a data set that is protected with an ALTER password, the new data set inherits the ALTER password.

The password is blotted out when the code is written in the SAS log. Here is an example:

```
set a(alter=XXXXXXX);
```

Note: A SAS password does not control access to a SAS file beyond SAS. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

ASYNCINDEX= Table Option

Specifies to create indexes in parallel when creating multiple indexes on an SPD Engine data set.

Valid in: CREATE INDEX Statement

Category: User Control of Index Usage

Restriction: This table option is not supported on the CAS server.

Data source: SPD Engine data set

Syntax

ASYNCINDEX= YES | NO

Arguments

YES
creates the indexes in parallel (asynchronously).

NO
creates one index at a time (synchronously). This is the default value.

Details

The SPD Engine can create multiple indexes for a data set at the same time. The SPD Engine spawns a single thread for each index created, and then processes the threads simultaneously. Although creating indexes in parallel is much faster than creating one index at a time, the default for this option is NO. Parallel creation requires additional utility work space and additional memory, which might not be available. If the index creation fails due to insufficient resources, you can do one of the following:

- set the SAS system option to MEMSIZE=0
- increase the size of the utility file space using the SPDEUTILLOC= system option

You increase the memory space that is used for index sorting using the SPDEINDEXSORTSIZE= system option. If you specify to create indexes in parallel, specify a large-enough space using the SPDEUTILLOC= system option.

BL_ACCOUNTNAME= Table Option

Specifies the name of the account to use on the external storage system.

Category:	Bulk Loading
Default:	none
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1403.
Data source:	Snowflake
Note:	Support for this option was added in SAS 9.4M9.

Syntax

BL_ACCOUNTNAME= *account-name*

Arguments

account-name
specifies the name of the account to use on the external storage system.

Details

This option is used to bulk load data to Snowflake in an Azure Data Lake Storage (ADLS) location.

The external data location is generated using this option, the BL_FILESYSTEM= table option, and optionally the BL_FOLDER= table option. These values are combined to define the URL to an external storage location in an Azure Data Lake Storage Gen2 location:

```
https://<account-name>.<network-storage-host-name>/<file-system-name>/<file-path-folder>
```

In the HTTPS path, *account-name* is the stage account.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_ALLOW_READ_ACCESS= Table Option

Specifies that the original table data is still visible to readers during bulk load.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	DB2 under UNIX and PC

Syntax

BL_ALLOW_READ_ACCESS= YES | NO

Arguments

YES
specifies that the original (unchanged) data in the table is still visible to readers while bulk load is in progress.

NO
specifies that readers cannot view the original data in the table while bulk load is in progress. This is the default value.

Details

For more information about using this option, see the SQLU_ALLOW_READ_ACCESS parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_APPLICATIONID= Table Option

Specifies the application ID (a GUID string) for accessing the external storage system.

Category:	Bulk Loading
Default:	none
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirements:	Must be specified within the "BULKOPTS= Table Option" on page 1403. The user ID that is used to authenticate must be assigned Contributor and Storage Blob Data Contributor roles in the ADLS account configuration.
Data source:	Snowflake
Note:	Support for this option was added in SAS 9.4M9.
Example:	bl_applicationid="b1fc955d5c-e0e2-45b3-a3cc-a1cf54120f"

Syntax

BL_APPLICATIONID= *"string"*

Arguments

string
specifies the Microsoft Entra Application ID (a GUID string) for accessing the external storage system.

Details

This option is used to bulk load data to Snowflake in an ADLS location.

When the BL_APPLICATIONID= option is specified, the Microsoft Entra Application ID is the principal that is authorized to access the data instead of a specific user. Any person who has credentials for the Microsoft Entra Application ID can access the data.

See Also

Table Options:

- "BL_ACCOUNTNAME= Table Option" on page 1366
- "BL_AZURE_SAS= Table Option" on page 1369
- "BL_AZURE_TENANTID= Table Option" on page 1370
- "BL_FILESYSTEM= Table Option" on page 1380
- "BULKLOAD= Table Option" on page 1400

BL_AZURE_SAS= Table Option

Specifies the Azure shared access signature token value.

Category:	Bulk Loading
Default:	none
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Snowflake

Note: Support for this option was added in SAS 9.4M9.

Syntax

BL_AZURE_SAS= *value*

Arguments

value
specifies the Azure shared access signature token value.

Details

This option provides the stage application signature for the ADLS connection.

See Also

Table Options:

- "BL_ACCOUNTNAME= Table Option" on page 1366
- "BL_APPLICATIONID= Table Option" on page 1368
- "BL_AZURE_TENANTID= Table Option" on page 1370
- "BL_FILESYSTEM= Table Option" on page 1380
- "BULKLOAD= Table Option" on page 1400

BL_AZURE_TENANTID= Table Option

Specifies the tenant ID for connecting to a Microsoft Azure storage system.

Category: Bulk Loading

Default: none

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirements: Must be specified within the "BULKOPTS= Table Option" on page 1403.
The user ID that is used to authenticate must be assigned Contributor and Storage Blob Data Contributor roles in the ADLS account configuration.

Data source: Snowflake

Note: Support for this option was added in SAS 9.4M9.

Important: User interaction is required on a user's first attempt to access data. For more information, see "Using Device-Code Authentication" in AZURETENANTID= System Option.

Syntax

BL_AZURE_TENANTID= *tenant-id*

Arguments

tenant-id

specifies the tenant ID, which is a GUID, for connecting to Microsoft Azure Data Lake Storage Gen2.

Length The maximum length of the value is 64 bytes.

Restriction If the name includes spaces or non-alphanumeric characters, then enclose the name in single or double quotation marks. Usually, this value includes the hyphen (-) character.

Details

The AZURETENANTID= table option is used with device-code authentication in Microsoft Azure Data Lake Storage Gen2. This table option provides the functionality of the AZURETENANTID= system option to DS2.

See Also

Table Options:

- "BL_ACCOUNTNAME= Table Option" on page 1366
- "BL_APPLICATIONID= Table Option" on page 1368
- "BL_AZURE_SAS= Table Option" on page 1369
- "BL_FILESYSTEM= Table Option" on page 1380
- "BULKLOAD= Table Option" on page 1400

BL_COMPRESS= Table Option

Specifies whether to compress data files using the gzip format.

Category: Bulk Loading

Default: NO

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.

This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Snowflake

Note: Support for this option was added in SAS 9.4M9.

Syntax

BL_COMPRESS= YES | NO

Details

If you have a slow network or very large data files, using BL_COMPRESS=YES might improve performance.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_COPY_LOCATION= Table Option

Specifies the directory to which DB2 saves a copy of the loaded data.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_COPY_LOCATION= *pathname*

Arguments

pathname
specifies the path where the loaded data is copied.

See Also

Table Options:

- "BL_RECOVERABLE= Table Option" on page 1391
- "BULKLOAD= Table Option" on page 1400

BL_CPU_PARALLELISM= Table Option

Specifies the number of processes or threads to use when building table objects.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_CPU_PARALLELISM= *number-of-processes-or-threads*

Arguments

number-of-processes-or-threads
specifies the number of processes or threads that the load utility uses to parse, convert, and format data records when building table objects.

Details

This option exploits intrapartition parallelism and significantly improves load performance. It is particularly useful when loading presorted data, because record order in the source data is preserved.

The maximum number that is allowed is 30. If the value is 0 or has not been specified, the load utility selects an intelligent default. This default is based on the number of available CPUs on the system at run time. If there is insufficient memory to support the specified value, the utility adjusts the value.

When BL_CPU_PARALLELISM is greater than 1, the flushing operations are asynchronous, permitting the loader to exploit the CPU. If tables include either LOB or LONG VARCHAR data, parallelism is not supported. The value is set to 1, regardless of the number of system CPUs or the specified value.

Although use of this parameter is not restricted to symmetric multiprocessor (SMP) hardware, you might not obtain any discernible performance benefit from using it in non-SMP environments.

For more information about using BL_CPU_PARALLELISM=, see the CPU_PARALLELISM parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BL_DATA_BUFFER_SIZE= Table Option" on page 1374
- "BL_DISK_PARALLELISM= Table Option" on page 1378
- "BULKLOAD= Table Option" on page 1400

BL_DATA_BUFFER_SIZE= Table Option

Specifies the total amount of memory to allocate for the bulk load utility to use as a buffer for transferring data.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_DATA_BUFFER_SIZE= *buffer-size*

Arguments

buffer-size

specifies the total amount of memory (in 4KB pages) that is allocated for the bulk load utility to use as buffered space for transferring data within the utility. This setting does not consider the degree of parallelism that is available.

Details

If you specify a value that is less than the algorithmic minimum, the minimum required resource is used and no warning is returned. This memory is allocated directly from the utility heap, the size of which you can modify through the util_heap_sz database configuration parameter. If you do not specify a value, the utility calculates an intelligent default at run time. The calculation is based on a percentage of the free space that is available in the utility heap at the time of instantiation of the loader, as well as some characteristics of the table.

It is recommended that the buffer be several extents in size. An *extent* is the unit of movement for data within DB2, and the extent size can be one or more 4KB pages.

The DATA BUFFER parameter is useful when you are working with large objects (LOBs) because it reduces input and output waiting time. The data buffer is allocated from the utility heap. Depending on the amount of storage that is available on your system, you should consider allocating more memory for use by the DB2 utilities. You can modify the database configuration parameter util_heap_sz accordingly. The default value for the Utility Heap Size configuration parameter is 5,000 4KB pages. Because load is one of several utilities that use memory from the utility heap, it is recommended that no more than 50% of the pages that are defined by this parameter be made available for the load utility.

For more information about using this option, see the DATA BUFFER parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- “BL_CPU_PARALLELISM= Table Option” on page 1373.
- “BL_DISK_PARALLELISM= Table Option” on page 1378
- “BULKLOAD= Table Option” on page 1400

BL_DEFAULT_DIR= Table Option

Specifies the location where bulk load creates all intermediate files.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1403
Data source:	Oracle, Snowflake
Note:	Support for Snowflake was added in SAS 9.4M9.

Syntax

BL_DEFAULT_DIR= *host-specific-directory-path*

Arguments

host-specific-directory-path
specifies the host-specific directory path where intermediate bulk-load files are created. The default is the current directory.

Details

The value that you specify for this option is prepended to the filename. Be sure to provide the complete, host-specific directory path, including the file and directory separator character to accommodate all platforms.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_DELETE_DATAFILE= Table Option

Specifies whether to delete intermediate files created by the bulk loading facility.

Category:	Bulk Loading
Default:	YES
Restriction:	This table option is not supported on the CAS server.
Data source:	Snowflake

Syntax

BL_DELETE_DATAFILE= YES | NO

Arguments

YES
deletes all (data, control, and log) files that the SAS/ACCESS engine creates for the DBMS bulk-load facility.

NO
retains the files.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_DELIMITER= Table Option

Overrides the default delimiter character for separating columns of data during bulk loading.

Category:	Bulk Loading
Default:	The bell character (ASCII 0x07 or octal \007)
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Snowflake
Note:	Support for this option was added in SAS 9.4M9.

Syntax

BL_DELIMITER= '*single-ascii-character* | *octal-representation*'

Details

CSV is the file format used for bulk loading to Snowflake.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_DISK_PARALLELISM= Table Option

Specifies the number of processes or threads to use when writing data to disk.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_DISK_PARALLELISM= *number-of-processes-or-threads*

Arguments

number-of-processes-or-threads
specifies the number of processes or threads that the load utility uses to write data records to the table-space containers.

Details

This option exploits the available containers when it loads data and significantly improves load performance.

The maximum number that is allowed is the greater of four times the BL_CPU_PARALLELISM value, which the load utility actually uses, or 50. By default, BL_DISK_PARALLELISM is equal to the sum of the table-space containers on all table spaces that contain objects for the table that is being loaded except where this value exceeds the maximum number that is allowed.

If you do not specify a value, the utility selects an intelligent default that is based on the number of table-space containers and the characteristics of the table.

For more information about using this option, see the DISK_PARALLELISM parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BL_CPU_PARALLELISM= Table Option" on page 1373

- "BULKLOAD= Table Option" on page 1400

BL_ERRORS= Table Option

Specifies that, after the indicated number of errors is received, the load should stop.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Oracle

Syntax

BL_ERRORS= *number*

Arguments

number
specifies the number of errors that should be received before the load stops. The default is 1000000 errors.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_EXCEPTION= Table Option

Specifies the exception table into which rows in error are copied.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_EXCEPTION= *exception-table-name*

Arguments

exception-table-name
specifies the exception table into which rows in error are copied.

Details

Any row that is in violation of a unique index or a primary key index is copied. DATALINK exceptions are also captured in the exception table. If you specify an unqualified table name, the table is qualified with the CURRENT SCHEMA. Information that is written to the exception table is not written to the dump file. In a partitioned database environment, you must define an exception table for those partitions on which the loading table is defined. However, the dump file contains rows that cannot be loaded because they are not valid or contain syntax errors.

For more information about using this option, see the FOR EXCEPTION parameter in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*. For more information about the load exception table, see the load exception table topics in *IBM DB2 Universal Database Data Movement Utilities Guide and Reference* and *IBM DB2 Universal Database SQL Reference, Volume 1*.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400

BL_FILESYSTEM= Table Option

Specifies the name of the file system within the external storage system.

Category:	Bulk Loading
Default:	none
Restriction:	This table option is not supported on the CAS server.
Requirements:	This table option is required to enable bulk loading to Snowflake in Azure. Must be specified within the “BULKOPTS= Table Option” on page 1403.
Data source:	Snowflake
Note:	Support for this option was added in SAS 9.4M9.

Syntax

BL_FILESYSTEM= *file-system-name*

Arguments

file-system-name
specifies the name of the file system within ADLS that files are to be written to.

Details

This option is used to bulk load data to Snowflake in an ADLS location.

The external data location is generated using the BL_ACCOUNTNAME= option, the BL_APPLICATIONID= option, and optionally the BL_FOLDER= table option. These values are combined to define the URL to access an Azure Data Lake Storage Gen2 location:

```
https://<account-name>.<network-storage-host-name>/<file-system-name>/<file-path-folder>
```

In the HTTPS path, *file-system-name* is the BLOB container.

See Also

Table Options:

- “BL_ACCOUNTNAME= Table Option” on page 1366
- “BL_APPLICATIONID= Table Option” on page 1368
- “BL_AZURE_SAS= Table Option” on page 1369
- “BL_AZURE_TENANTID= Table Option” on page 1370
- “BL_FOLDER= Table Option” on page 1381
- “BULKLOAD= Table Option” on page 1400

BL_FOLDER= Table Option

Specifies the folder or file path for the data in the external storage system.

Category: Bulk Loading

Default: none

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.

This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Snowflake

Note: Support for this option was added in SAS 9.4M9.

Syntax

BL_FOLDER= *file-path-and-folder*

Arguments

file-path-and-folder
specifies the folder or file path for the data in the external storage system. The location path starts with the container name.

Details

This option is used to bulk load data to Snowflake in an ADLS location. When specified, this option is included in the generated URL that is used to access an Azure Data Lake Storage Gen2 location.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_INDEXING_MODE= Table Option

Specifies which scheme the DB2 load utility should use for index maintenance.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_INDEXING_MODE= AUTOSELECT | REBUILD | INCREMENTAL | DEFERRED

Arguments

AUTOSELECT
specifies that the load utility automatically decides between REBUILD or INCREMENTAL mode.

REBUILD
specifies that all indexes are rebuilt.

INCREMENTAL
specifies that indexes are extended with new data.

DEFERRED
specifies that the load utility does not attempt index creation. Indexes are marked as needing a refresh.

Details

For more information about using the values for this option, see *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_INTERNAL_STAGE= Table Option

Specifies the internal stage and path for bulk loading to Snowflake.

Category:	Bulk Loading
Alias:	BL_STAGE=
Default:	none
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirements:	This table option is required to enable internal stage bulk loading to Snowflake. Must be specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Snowflake
Note:	Support for this option was added in SAS 9.4M9.

Syntax

BL_INTERNAL_STAGE=*stage-and-path*

Required Argument

stage-and-path
specifies the Snowflake internal stage and path.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_LOAD= Table Option

Specifies that, after the indicated number of rows is loaded, the load should stop.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	Oracle

Syntax

BL_LOAD= *number-of-rows*

Arguments

number-of-rows
specifies the number of rows that should be loaded. The first rows from the table will be loaded. The default is to load all rows.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_LOAD_REPLACE= Table Option

Specifies whether DB2 appends or replaces rows during bulk loading.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_LOAD_REPLACE= NO | YES

Arguments

NO
specifies that the CLI LOAD interface appends new rows of data to the DB2 table. This is the default value.

YES
specifies that the CLI LOAD interface replaces the existing data in the table.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_LOG= Table Option

Identifies a log file that contains information such as statistics and error information for a bulk load.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC, Oracle

Syntax

BL_LOG= "*path-and-log-file-name*"

Arguments

path-and-log-file-name
is a file to which information about the loading process is written. The default path and log filename is DBMS-specific.

Details

When the DBMS bulk load facility is invoked, it creates a log file. The contents of the log file are DBMS-specific. The BL_ prefix distinguishes this log file from the one created by the SAS log. If the BL_LOG= table option is specified with the same path and filename as an existing log, the new log replaces the existing log.

If the BL_LOG= table option is not specified, the log file is deleted automatically after a successful operation.

Example

The BL_LOG= table option is specified within the BL_BULKOPTS= table option:

```
bulkload=yes; bulkopts=(bl_log="c:\temp\bulkload.log");
```

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400

BL_LOGFILE= Table Option

Specifies the filename for the bulk load log file.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1403
Data source:	Oracle

Syntax

BL_LOGFILE= '*log-file-name*'

Arguments

log-file-name
specifies a name for the bulk load log file. The default is a generated filename that has the template BL_*TablenameUniquenumber*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_NUM_DATAFILES= Table Option

Specifies the number of data files to create to contain the source data.

Category:	Bulk Loading
Alias:	BL_NUM_DATA_FILES
Default:	1
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Snowflake
Note:	Support for this option was added in SAS 9.4M9.

Syntax

BL_NUM_DATAFILES= *value*

Arguments

value
specifies the number of files to create to hold your source data. This value can be an integer between 1 and 100, inclusively.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_OPTIONS= Table Option

Passes options to the DBMS bulk load facility, affecting how it loads and processes data.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	DB2 under UNIX and PC

Syntax

BL_OPTIONS= '*option* [, ...*option*] '

Arguments

option
specifies a valid DB2 option. By default, no options are passed.

Details

The BL_OPTIONS= table option enables you to pass options to the DBMS bulk load facility when it is invoked, thereby affecting how data is loaded and processed. You must separate multiple options with commas and enclose the entire string of options in single quotation marks.

This option passes DB2 file type modifiers to DB2 LOAD or IMPORT commands to affect how data is loaded and processed. Not all DB2 file type modifiers are appropriate for all situations. You can specify one or more DB2 file type modifiers with .IXF files. For a list of file type modifiers, see the description of the LOAD and IMPORT utilities in *DB2 Data Movement Utilities Guide and Reference*.

Example

This option is specified within the BULKOPTS= table option:

```
bulkload=yes; bulkopts=(bl_options='option1, option2');
```

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400
- "BULKOPTS= Table Option" on page 1403

BL_PARALLEL Table Option

Specifies whether to perform a parallel bulk load.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Oracle

Syntax

BL_PARALLEL= YES | NO

Arguments

YES
specifies that a parallel load should be performed.

NO
specifies that a parallel load is not performed. That is the default behavior.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_PORT_MAX= Table Option

Sets the highest available port number for concurrent uploads.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_PORT_MAX= *integer*

Arguments

integer
specifies a positive integer that represents the highest available port number for concurrent uploads.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_PORT_MIN= Table Option

Sets the lowest available port number for concurrent uploads.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_PORT_MIN= *integer*

Arguments

integer
specifies a positive integer that represents the lowest available port number for concurrent uploads.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_RECOVERABLE= Table Option

Specifies whether the LOAD process is recoverable.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: DB2 under UNIX and PC, Oracle

Syntax

BL_RECOVERABLE= YES | NO

Arguments

YES
specifies that the LOAD process is recoverable. For DB2, YES also specifies that BL_COPY_LOCATION= should specify the copy location for the data.

NO
specifies that the LOAD process is not recoverable.

Details

DB2 under UNIX and PC Hosts: The default is NO.

Oracle: The default is YES. Set this option to NO to improve direct load performance. Specifying NO adds the UNRECOVERABLE keyword before the LOAD keyword in the control file.

CAUTION
Be aware that an unrecoverable load does not log loaded data into the redo log file. Therefore, media recovery is disabled for the loaded table. For more information about the implications of using the UNRECOVERABLE parameter in Oracle, see your Oracle utilities documentation.

See Also

Table Options:

- “BL_COPY_LOCATION= Table Option” on page 1372
- “BULKLOAD= Table Option” on page 1400

BL_REMOTE_FILE= Table Option

Specifies the base filename and location of DB2 LOAD temporary files.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1403

Data source: DB2 under UNIX and PC

Syntax

BL_REMOTE_FILE= *pathname-and-base-filename*

Arguments

pathname-and-base-filename
specifies the full pathname and base filename to which DB2 appends extensions (such as .log, .msg and .dat files) to create temporary files during load operations. By default, BL_<*table*>_<*unique-ID*> is the form of the base filename.

table
specifies the table name.

unique-ID
specifies a number that prevents collisions in the event of two or more simultaneous bulk loads of a particular table. The table driver generates this number.

Details

When you specify this option, the DB2 LOAD command is used (instead of the IMPORT command).

For *pathname*, specify a location on a DB2 server that is accessed exclusively by a single DB2 server instance, and for which the instance owner has Read and Write permissions. Make sure that each LOAD command is associated with a unique *pathname-and-base-filename* value.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_SKIP= Table Option

Specifies to skip the indicated number of rows before starting the bulk load.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Oracle

Syntax

BL_SKIP= *number-of-rows*

Arguments

number-of-rows
specifies the number of rows to skip before beginning the load. The default is 0, which means that no records are skipped.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_SKIP_INDEX_MAINTENANCE= Table Option

Specifies whether to perform index maintenance on the bulk load.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the"BULKOPTS= Table Option" on page 1403

Data source: Oracle

Syntax

BL_SKIP_INDEX_MAINTENANCE= YES | NO

Arguments

YES
specifies to stop index maintenance on the load. This causes the index partitions that would have had index keys added to them to be marked Index Unusable. The index segment is inconsistent with the data it indexes. Index segments that are not affected by the load retain the Index Unusable state that they had prior to the load.

NO
specifies that index maintenance is performed on the load. This is the default value.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_SKIP_UNUSABLE_INDEXES= Table Option

Specifies whether to skip index entries that are in an unusable state and continue with the bulk load.

Category: Bulk Loading
Restriction: This table option is not supported on the CAS server.
Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source: Oracle

Syntax

BL_SKIP_UNUSABLE_INDEXES= YES | NO

Arguments

YES
specifies that the unusable index entry should be skipped. This is the default value.

NO
specifies that the unusable index entry should not be skipped.

Details

If an index in an Index Unusable state is encountered, by default, it is skipped and the load operation continues. This allows the SQL*Loader to load a table with indexes that are in an unusable state before beginning the load. Indexes that are not in an unusable state at load time are maintained by the SQL*Loader. Indexes that are in an unusable state at load time are not maintained and remain in an unusable state at load completion.

If this bulk load option is not specified, the default value is specified in the Oracle database configuration parameter, SKIP_UNUSABLE_INDEXES. This value is specified in the initialization parameter file. The BL_SKIP_UNUSABLE_INDEXES bulk load table option overrides the value of the SKIP_UNUSABLE_INDEXES configuration parameter in the initialization parameter file.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_TIMEOUT= Table Option

Specifies the maximum completion time in seconds for a storage task.

Category:	Bulk Loading
Default:	60 seconds
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Snowflake
Note:	Support for this option was added in SAS 9.4M9.

Syntax

BL_TIMEOUT= *value*

Arguments

value
specifies the maximum completion time for a storage task, expressed as a number of seconds. If the task does not complete within the specified time, the storage system might return an error.

Details

This option supports bulk loading of data to Snowflake in an ADLS location.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

BL_USE_ESCAPE= Table Option

Specifies whether to escape a predefined set of special characters during bulk loading or bulk retrieval.

Category:	Bulk Loading
Default:	NO
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Snowflake
Notes:	A user-defined delimiter character is specified with the BL_DELIMITER table option. If you have specified a value for BL_DELIMITER=, that value is also treated as an escape character. Support for this option was added in SAS 9.4M9.

Syntax

BL_USE_ESCAPE= YES | NO

Arguments

YES
specifies that occurrences of a backslash (‘\’), newline (‘\n’), carriage return (‘\r’), or user-defined delimiter receives a backslash inserted before it. The backslash preserves these occurrences in the data file that is used for bulk loading or data retrieval. Any occurrence of the NULL character (0x00) is replaced by a space (‘ ’).

NO
specifies that occurrences of a backslash (‘\’), newline (‘\n’), carriage return (‘\r’), or user-defined delimiter are not escaped in the data file that is used for the bulk loading or data retrieval.

Details

Using this option can increase the time for bulk loading data, because the escaping character needs to be added before the set of escaped characters and then removed again in the final data set.

A bulk-loading operation in Snowflake uses CSV files.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400

BL_WARNING_COUNT= Table Option

Specifies the maximum number of row warnings to allow before the load fails.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1403
Data source:	DB2 under UNIX and PC

Syntax

BL_WARNING_COUNT= *number-of-warnings*

Arguments

number-of-warnings
specifies the maximum number of row warnings to allow before the load fails.

Details

To specify this option, you must first set BULKLOAD=YES and also specify a value for BL_REMOTE_FILE=.

Use this option to limit the maximum number of rows that generate warnings. See the log file for information about why the rows generated warnings.

See Also

Table Options:

- "BL_REMOTE_FILE= Table Option" on page 1392
- "BULKLOAD= Table Option" on page 1400

BUFNO= Table Option

Specifies the number of buffers to be allocated for processing a SAS data set.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set

Syntax

BUFNO= *n* | *n*K | *hex*X | MIN | MAX

Arguments

***n* | *n*K**
specifies the number of buffers in multiples of 1 (bytes); 1,024 (kilobytes). For example, a value of `8` specifies 8 buffers, and a value of `1K` specifies 1024 buffers.

Requirement *K* must be uppercased.

***hex*X**
specifies the number of buffers as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value `2dX` sets the number of buffers to 45 buffers.

Requirement *X* must be uppercased.

MIN
sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default value.

MAX
sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer. The largest four-byte, signed integer is 2^{31}–1, or approximately 2 billion.

Details

The buffer number is not a permanent attribute of the data set; it is valid only for the current operation.

The BUFNO= table option applies to SAS data sets that are opened for input, output, or update.

A larger number of buffers can speed up execution time by limiting the number of input and output (I/O) operations that are required for a particular SAS data set. However, the improvement in execution time comes at the expense of increased memory consumption.

To reduce I/O operations on a small data set as well as speed execution time, allocate one buffer for each page of data to be processed. This technique is most effective if you read the same observations several times during processing.

BUFSIZE= Table Option

Specifies the size of a permanent buffer page for an output SAS data set.

Category: Table Control

Restrictions: This table option is not supported on the CAS server.
Use with output data sets only.

Data source: SAS data set

Syntax

BUFSIZE=*n* | *n*K | *n*M | *n*G | *hex*X | MIN | MAX

Arguments

***n* | *n*K | *n*M | *n*G**
specifies the page size in multiples of 1 (bytes); 1,024 (kilobytes); 1,048,576 (megabytes); or 1,073,741,824 (gigabytes). For example, a value of `8` specifies a page size of 8 bytes, and a value of `4k` specifies a page size of 4096 bytes.

Requirement *K*, *M*, and *G* must be uppercased.

***hex*X**
specifies the page size as a hexadecimal value. You must specify the value beginning with a number (0–9), followed by an X. For example, the value `2dx` sets the page size to 45 bytes.

Requirement *X* must be uppercased.

MIN
sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment.

MAX
sets the page size to the maximum possible number in your operating environment, up to the largest four-byte, signed integer. The largest four-byte, signed integer is 2^{31}–1, or approximately 2 billion bytes.

Details

The page size is the amount of data that can be transferred for a single I/O operation to one buffer. The page size is a permanent attribute of the data set and is used when the data set is processed.

A larger page size can speed up execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the cost of increased memory consumption.

To change the page size, copy the data set and either specify a new page or use the SAS default. To reset the page size to the default value in your operating environment, specify BUFSIZE=0.

Operating Environment Information: The default value for the BUFSIZE= table option is determined by your operating environment and is set to optimize sequential access. To improve performance for direct (random) access, you should change the value for BUFSIZE=.

BULKLOAD= Table Option

Determines whether SAS uses a DBMS facility to insert data into a DBMS table.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Interaction: Used in conjunction with "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift, DB2 under UNIX and PC, Google BigQuery, Impala, MDS, Microsoft SQL Server, Oracle, Snowflake, Teradata

Note: Support for Amazon Redshift, Microsoft SQL Server, and Snowflake was added in SAS 9.4M9.

Syntax

BULKLOAD= YES | NO

Arguments

YES
calls a DBMS-specific bulk-load facility in order to insert or append rows to a DBMS table.

NO
does not call the DBMS-specific bulk-load facility. This is the default value.

Details

Overview

Using BULKLOAD=YES is the fastest way to insert rows into a DBMS table.

You can specify data source-specific options in the BULKOPTS= table option (BL_BULKOPTS= for Oracle). BULKOPTS= functions as a container for the data source-specific options. For more information, see "BULKOPTS= Table Option" on page 1403.

When the BULKLOAD= table option is not set, a simple multi-row insert SQL scheme is used to insert data rows.

Usage Notes

Amazon Redshift

The bulk-load facility uses the Amazon Simple Storage Service (S3) tool to move data to the Amazon Redshift database. The BULKLOAD= table option must be set to YES to enable bulk loading. Data source-specific options for bulk loading are required and must be specified in the BULKOPTS= option. For option usage information, see "Bulk Loading and Bulk Unloading with Amazon Redshift" in *SAS/ACCESS for Relational Databases: Reference*.

Google BigQuery

Bulk-loading options for Google BigQuery apply to Insert operations only. Bulk loading can be enabled in the LIBNAME statement for the BIGQUERY engine or using the BULKLOAD= and BULKOPTS= table options. When the options are specified both ways, the table option overrides the LIBNAME option. See “Bulk Loading for Google BigQuery” in *SAS/ACCESS for Relational Databases: Reference* for valid Google BigQuery bulk-load options.

Impala

Bulk loading to the Impala server can be accomplished in two ways: you can use the WebHDFS interface to Hadoop to push data to HDFS, or you can configure a required set of Hadoop JAR files. Both approaches require Hadoop configuration files needed by SAS to be in one location and available to the client machine. To use WebHDFS, you must additionally set the SAS_HADOOP_RESTFUL= environment variable to 1. To use Java, you must make the Hadoop JAR location known to the client machine and ensure that the SAS_HADOOP_RESTFUL= environment variable is not set to 1 (or TRUE or YES).

Specifying BULKLOAD=YES causes two CREATE TABLE statements to be issued to the Impala server. One creates the target Impala table. The other creates a temporary table. SAS uses WebHDFS to upload table data to the HDFS /tmp directory. The resulting file is a UTF-8 delimited text file. SAS issues a LOAD DATA statement to move the data file from the /tmp directory to the temporary table, and then issues an INSERT INTO statement that copies and transforms the text data from the temporary table to the target table. The temporary table is then deleted from HDFS.

MDS

BULKLOAD= is available for insert operations only. When the BULKLOAD= table option is set, newly inserted rows are committed immediately and become visible to existing transactions. When the BULKLOAD= table option is not set, newly inserted rows are not visible until the existing transactions are committed or rolled back.

Microsoft SQL Server

The table option BULKLOAD=YES enables bulk loading for the SQL Server database. This option sets the attribute SQL_ATTR_BULK_LOAD_ENABLED=1 in the underlying DataDirect driver. There are no data source-specific table options for bulk loading to Microsoft SQL Server.

Snowflake

Bulk loading is available for the Snowflake internal stage and for Snowflake in ADLS storage. Bulk loading can be enabled by specifying BULKLOAD=YES in the LIBNAME statement for the SNOW engine or using the BULKLOAD= table option.

Data source-specific options are required to connect to Snowflake for bulk loading. Data source-specific options from the LIBNAME statement are passed to the FEDSQL procedure internally as connection options. When enabling bulk loading

with table options, you must specify appropriate data source-specific table options in the BULKOPTS= table option. For more information, see "BULKOPTS= Table Option" on page 1403. For an example of how the bulk-load table options are specified, see the BULKOPTS= example for Snowflake in Azure ADLS and the BULKOPTS= example for stage bulk loading in Snowflake.

Teradata

Specifying BULKLOAD=YES invokes the Teradata Parallel Transporter (TPT) API protocol driver. The default TPT operator is the Stream operator. Use the BULKOPTS= table option to specify a different TPT operator.

Example: Modify Mylib.Myexample

```
data mylib.myexample (BULKLOAD=YES
          BULKOPTS=(BL_DELETE_DATAFILE=NO));
   dcl float         f;
   method init();
      f=11.01; output;
   end;
enddata;
run;
```

See Also

Concepts:

- "DS2 Statement Table Options by Data Source" on page 1352

Table Options:

- "BULKOPTS= Table Option" on page 1403

BULKOPTS= Table Option

Provides a container for bulk-load options.

Category:	Bulk Loading
Alias:	In Oracle, this option has the name BL_BULKOPTS=
Restriction:	This table option is not supported on the CAS server.
Requirement:	BULKLOAD=YES must be set in order to use BULKOPTS=
Data source:	Amazon Redshift, DB2 under UNIX and PC, Google BigQuery, Impala, Oracle, Snowflake, Teradata
Note:	Support for Amazon Redshift and Snowflake was added in SAS 9.4M9.

Syntax

BULKOPTS= (*option*[...*option*])

Arguments

option[...option]
specifies one or more bulk load table options separated by spaces.

Details

To specify BULKOPTS= table option, you must first specify BULKLOAD=YES. For more information, see "BULKLOAD= Table Option" on page 1400.

The BULKOPTS= table option is a container option that is required in order to specify other bulk load table options.

Bulk load options are data source-specific. See "DS2 Statement Table Options by Data Source" on page 1352 for a list of the options that can be specified in this container for each data source.

Some data source-specific options for bulk loading are optional. However, some data sources require data source-specific table options to enable bulk loading (for . example, Amazon Redshift and Snowflake). The required options specify the stage for bulk loading or authenticate to the hosting platform.

Amazon Redshift: BULKOPTS= supports the same bulk-load options as the Amazon Redshift LIBNAME statement. For more information, see "Bulk Loading and Bulk Unloading with Amazon Redshift" in *SAS/ACCESS for Relational Databases: Reference*.

Google BigQuery users: DS2 supports the same bulk-load options as the SAS/ACCESS to Google BigQuery LIBNAME engine. See "Bulk Loading for Google BigQuery" in *SAS/ACCESS for Relational Databases: Reference* for valid options.

Examples

Example 1: Amazon Redshift Example

The following example uses the BULKLOAD=YES and BULKOPTS= options to enable bulk loading for an Amazon Redshift S3 bucket. RS_BUCKET= and RS_REGION= are required options. The others are authentication options. That is, you can authenticate using the options shown here, or you can authenticate using RS_KEY=, RS_SECRET=, and RS_TOKEN=.

```
BULKLOAD=YES BULKOPTS=(RS_AWS_CONFIG='path-and-file-name'
RS_CREDENTIALS_FILE='path-and-file-name' rs_region='region-value'
rs_bucket='bucket-name')
```

Example 2: DB2 Example

The following example uses the BULKLOAD=YES and BULKOPTS= table options to submit bulk-load options to DB2 under UNIX and PC:

```
BULKLOAD=YES BULKOPTS=(BL_LOG='c:\temp\bulkload.log'  BL_LOAD_REPLACE=yes
      BL_OPTIONS='ERRORS=999, LOAD=2000');
```

Example 3: Google BigQuery Example

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk-load options to Google BigQuery. The Google BigQuery loader client application is used to move data from the client to the Google BigQuery database. Valid bulk-load options are documented in "Bulk Loading Data in Google BigQuery" in *SAS/ACCESS for Relational Databases: Reference*.

```
BULKLOAD=YES BULKOPTS=(BL_DELETE_DATA_FILE=NO;BL_DEFAULT_DIR='/u/myid/temp')
```

Example 4: Impala Example

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk-load options to Impala. The bulk-load request uses WebHDFS to load the data to HDFS. Note that the SAS_HADOOP_RESTFUL= environment variable must also be set to 1 for the request to succeed.

```
BULKLOAD=YES BULKOPTS=(USER=HDFS PASS="hdfs-pwd"
CONFIG="my-config-path")
```

Example 5: Oracle Example

The following example uses the BULKLOAD=YES and BL_BULKOPTS= options to submit bulk-load options to Oracle:

```
BULKLOAD=YES BL_BULKOPTS=(BL_DEFAULT_DIR='C:\mylogs' BL_LOGFILE='novemberdatalog'
BL_ERRORS=999)
```

Example 6: Snowflake in ADLS

When bulk loading to Snowflake, you must set the bulk-load stage. The BL_ACCOUNTNAME=, BL_FILESYSTEM=, BL_APPLICATIONID=, BL_AZURE_TENANT_ID=, and BL_AZURE_SAS= table options are required to set an external stage for bulk loading to ADLS. These options are required for external stage bulk loading.

```
create table accesslib.mytable {options BULKLOAD=YES BULKOPTS=(BL_DELIMITER=","
BL_FOLDER="folder-name" BL_DEFAULT_DIR="directory-path" BL_ACCOUNTNAME="account-name"
BL_FILESYSTEM="filesystem-name" BL_APPLICATIONID="application-id"
BL_AZURE_TENANTID="tenant-id" BL_AZURE_SAS="value")}
as select * from otherlib.othertable;
```

Example 7: Stage Bulk Loading in Snowflake

These table options enable internal stage bulk loading for Snowflake. BL_INTERNAL_STAGE= is a required option. BL_NUM_DATA_FILES= is required when uploading more than one file.

```
create table accesslib.mytable {options BULKLOAD=YES BULKOPTS=(BL_DELETE_DATA_FILE=NO;
BL_NUM_DATA_FILES=2;BL_INTERNAL_STAGE='MY_INTERNAL_STAGE/user-name')}
as select * from otherlib.othertable;
```

Example 8: Teradata Example

The following example uses the BULKLOAD=YES and BULKOPTS= options to submit bulk-load options to Teradata. The Teradata Parallel Transporter (TPT) is used to load data into Teradata tables when BULKLOAD=YES. The TPT job uses the Stream operator by default. In the example below, the TD_TPT_OPER= table option specifies to use the Load operator instead, among other things.

```
BULKLOAD=YES BULKOPTS=(TD_TPT_OPER=LOAD TD_ERROR_LIMIT=1
TD_TRACE_OUTPUT=tpttrace TD_TRACE_LEVEL=TD_OPER_ALL
TD_MAX_SESSIONS=8 TD_TRACE_LEVEL_INF=TD_OPER_ALL)
```

See Also

Table Options:

- “TD_TPT_OPER= Table Option” on page 1482
- “CONFIG= Table Option” on page 1408

COMPRESS= Table Option

Specifies how rows are compressed in a new output data set or table.

Category: Table Control

Restrictions: This table option is not supported on the CAS server.
Use with output tables only.

Data source: SAS data set, SPD Engine data set, SPD Server table

Syntax

COMPRESS= NO | YES | CHAR | BINARY

Arguments

CHAR | YES
specifies that the rows in a newly created table are compressed (variable length records). SAS data sets are compressed. by using Run Length Encoding (RLE). RLE compresses rows by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations. SPD Engine data sets are compressed by using the run length compression algorithm SPDSRLC2. SPD

Server tables are compressed by using the run-length compression algorithm SPDSRLLC.

Alias ON (SPD Engine data set only)

Tip Use this compression algorithm for character data.

BINARY
specifies that the rows in a newly created data set or table are compressed by SAS using Ross Data Compression (RDC). RDC combines run-length encoding and sliding-window compression to compress the file. SPD Engine data sets are compressed by the SPD Engine by using SPDSRDC.

Tip This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

NO
specifies that the rows in a newly created table are uncompressed (fixed-length records).

Details

Compressing a table is a process that reduces the number of bytes required to represent each row. Advantages of compressing a table include reduced storage requirements for the table and fewer I/O operations necessary to read or write to the data during processing. However, more CPU resources are required to read a compressed table (because of the overhead of uncompressing each row). Also, there are situations where the resulting file size might increase rather than decrease.

Use the COMPRESS= table option to compress an individual table. Specify the option for an output table only—that is, a table name on the CREATE TABLE statement.

Note: In SPD Engine data sets and SPD Server tables, encryption and compression are mutually exclusive. You cannot specify encryption for a compressed SPD Engine data set or SPD Server table. You cannot compress an encrypted SPD Engine data set or SPD Server table.

After a table is compressed, the setting is a permanent attribute of the table, which means that to change the setting, you must re-create the table. That is, to uncompress a table, you must drop the table by using the DROP TABLE statement and re-create the table with the CREATE TABLE statement and the COMPRESS=NO option.

When you specify COMPRESS= to compress an SPD Engine data set or SPD Server table, the table drivers compress, by blocks, the table as it is created. To specify the

size of the compressed blocks, use the "IOBLOCKSIZE= Table Option" on page 1433.

The SPD Engine table driver also supports the PADCOMPRESS= table option when creating or updating the SPD Engine data set. The PADCOMPRESS= option enables you to add padding to the newly compressed blocks. See the "PADCOMPRESS= Table Option" on page 1441.

Comparisons

The COMPRESS= table option overrides the COMPRESS= connection string option.

`SAS data sets only:` When you create a compressed table, you can also specify the REUSE=YES table option in order to track and reuse space. With REUSE=YES, new rows are inserted in space freed when other rows are updated or deleted. When the default REUSE=NO is in effect, new rows are appended to the existing table.

See Also

Table Options:

- "ENCRYPT= Table Option" on page 1415
- "POINTOBS= Table Option" on page 1445
- "IOBLOCKSIZE= Table Option" on page 1433
- "PADCOMPRESS= Table Option" on page 1441
- "REUSE= Table Option" on page 1454

CONFIG= Table Option

Specifies a file or pathname for Hadoop configuration path resolution.

Category:	Bulk Loading
Alias:	CFG=, HD_CONFIG=, CONFIGDIR=, CFGDIR=, HD_CONFIGDIR=
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Impala

Syntax

CONFIG=*"configuration-path"*

Arguments

"configuration-path"
specifies the name of a single file or a directory that contains Hadoop configuration information that is required by SAS on your machine. If this option is not specified, SAS searches for the value in the SAS_HADOOP_CONFIG_PATH environment variable.

Details

The SAS_HADOOP_CONFIG_PATH environment variable can list multiple directories (defined similarly to a PATH variable) to search for configuration information. Use the CONFIG= option when you want to use a specific configuration file or to search a specific directory for configuration information.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400

DATAFILE= Table Option

Specifies an alternate name and location for the temporary HDFS file.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1403.

Data source: Impala

Syntax

DATAFILE= *"filename"*

Arguments

"filename"
specifies the full pathname to a file. When this option is not specified, the default pathname for the temporary file is "/tmp/bl_*tablename_unixtimeval*.dat".

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

DBCREATE_INDEX_OPTS= Table Option

Allows DBMS-specific options to be added to the CREATE INDEX statement.

Category: Index Control

Restriction: This table option is not supported on the CAS server.

Data source: Microsoft SQL Server, ODBC

Syntax

DBCREATE_INDEX_OPTS= '*DBMS-SQL-clauses*'

Arguments

DBMS-SQL-clauses
specifies one or more DBMS-specific clauses that can be appended to the end of an SQL CREATE INDEX statement.

Details

This option enables you to add DBMS-specific clauses to the end of the CREATE INDEX statement. The interface passes the CREATE INDEX statement and its clauses to the DBMS, which executes the statement and creates the DBMS index.

Example

In the following example, a Hive index is created with the value of the DBCREATE_INDEX_OPTS= “as 'compact' with deferred rebuild” option appended to the CREATE INDEX statement:

```
create index "COL1_1A03" on "TKTS002_1A03"{options
DBCREATE_INDEX_OPTS="as 'compact'
with deferred rebuild"}("COL1")
```

The following CREATE INDEX statement is passed to the DBMS in order to create the index:

```
create index 'COL1_1A03' on table 'TKTS002_1A03'('COL1') as 'compact'
with
deferred rebuild
```

DBCREATE_TABLE_OPTS= Table Option

Specifies DBMS-specific options to be added to the DATA statement.

Restriction: This table option is not supported on the CAS server.

Data source: Amazon Redshift, DB2 under UNIX and PC, Google BigQuery, Greenplum, HAWQ, Hive, Microsoft SQL Server, MySQL, Netezza, Oracle, SAP HANA, Spark, Teradata

Syntax

DBCREATE_TABLE_OPTS= '*DBMS-options*'

Required Argument

DBMS-options
specifies one or more valid DBMS-specific options. If more than one option is specified, the options should be separated in the same way as options are separated in the DBMS.

Details

Basics

This option enables you to add DBMS-specific options to the DATA statement. The interface passes the DATA statement and its clauses to the DBMS, which executes the statement and creates the DBMS table. An example of this would be to use the COLUMN-DELIMITER= option to specify a column delimiter for Hadoop files.

Note: If the SAS/ACCESS DBCREATE_TABLE_OPTS LIBNAME option and the DS2 DBCREATE_TABLE_OPTS table option are used, the DS2 table option setting takes precedence.

Quoting DBMS Options

The *DBMS-options* must be enclosed with single-quotation marks. Elements within *DBMS-options* that are quoted should use double the quotation marks around the element. For example, if single quotation marks are used, you use two single quotation marks. If double quotation marks are used, you use two sets of double quotation marks. Here are some examples:

```
/* Using double quotes around DBMS-option element causes an error */
DBCREATE_TABLE_OPTS="FIELDS TERMINATED BY '\012' "

/* Using single quotes around DBMS-option element works */
DBCREATE_TABLE_OPTS='FIELDS TERMINATED BY "\012" '

/* You can double the quote character to insert one into a quoted value*/

/* doubled single quote passes single quote to database */
/* trailing space after last set of single quotes added for emphasis */
DBCREATE_TABLE_OPTS='FIELDS TERMINATED BY ''\012'' '

/* doubled double quotes passes double quotes to database */
/* trailing space after last set of double quotes added for emphasis */
DBCREATE_TABLE_OPTS='FIELDS TERMINATED BY ""\012"" '
```

Note: If your program is run in Spark, do not use two single or double quotation marks within the *DBMS-options* value. When Spark creates the output table, it does not remove one of the sets of quotation marks. Here is an example:

```
/* This value with doubled single quotes */
DBCREATE_TABLE_OPTS='FIELDS TERMINATED BY ''\012'' '
/* gets translated to this for any data source except Spark */
FIELDS TERMINATED BY '\012'
/* and gets translated like this for Spark jobs which causes an error */
FIELDS TERMINATED BY ''\012''
```

Examples

Example 1: Teradata Example

In the following example, the Teradata table Teralib is created with the value of the `primary index ()` option appended to the DATA statement.

```
libname teralib teradata server=terasvr database=model user=myid password=xxxx;

proc delete data=teralib.outtable;
proc ds2;
data teralib.outtable(overwrite=yes dbcreate_table_opts='primary index(i)');
```

```
   retain x 0;
   method run();
      do i=1 to 10 by 1 ;
         x=i;
         output;
      end;
   end;
enddata;
run;
quit;
```

Example 2: Hadoop Example

In the following example, the Hive table **Hivelib.Dzoutput** is created and the `partition by` option is appended to the DATA statement. In this example, **Col1** exists in the table in the SET statement, **Hivelib.Dzpt**, although this is not required.

```
options set=SAS_HADOOP_JAR_PATH="jar-path";
options set=SAS_HADOOP_CONFIG_PATH="config-path";
libname hivelib hadoop server="hostname" user=username password=xxxx
schema=ds2ip
dbmax_text=300;

proc ds2;
data hivelib.dzoutput (dbcreate_table_opts="partitioned by (col1 int)"
overwrite=yes);
   method run();
       set hivelib.dzpt;
          x+1;
          output; output;
   end;
enddata;
run;
quit;
```

DROP= Table Option

For an input table, excludes the specified columns from processing; for an output table, excludes the specified columns from being written to the table.

Data source: All

Syntax

DROP= (*column-list*)

Required Argument

column-list
specifies the names of the columns to omit from the output table.

Restriction Numbered range lists in the format *col1–col5* and name prefix lists in the format *col:* are not supported.

Details

The DROP= table option specifies that all columns in the *column-list* should not be included in the creation of output rows. Normally, all columns in the program data vector are included in the output rows. If the drop attribute is specified, all columns not included in the drop statement are used to create columns in the output rows.

If the DROP= table option is associated with an input table, the columns are not available for processing during program execution.

Comparisons

The DROP= table option differs from the DROP statement in these ways:

- In DS2 programs, the DROP= table option can apply to both input and output tables. The DROP statement applies only to output tables.
- In DS2 programs, when you create multiple output tables, use the DROP= table option to write different columns to different tables. The DROP statement applies to all output tables.
- The KEEP= table option specifies a list of columns to be included in processing or to be written to the output table.

Examples

Example 1: Excluding Columns from Output Tables

In this example, the variables SALARY and GENDER are not included in processing and they are not written to either output tables.

```
data plan1 plan2;
   method run ();
      set payroll(drop=(salary gender));
      if hired<'01jan07'd then output plan1;
         else output plan2;
   end;
enddata;
```

You cannot use SALARY or GENDER in any logic in the DS2 program because DROP= prevents the SET statement from reading them from PAYROLL.

Example 2: Processing Columns without Writing Them to the Output Table

In this example, SALARY and GENDER are not written to PLAN2, but they are written to PLAN1.

```
data plan1 plan2(drop=(salary gender));
   method run ();
       set payroll;
       if hired<'01jan07'd then output plan1;
          else output plan2;
   end;
enddata;
```

See Also

Statements:

- "DROP Statement" on page 1243

Table Options:

- "KEEP= Table Option" on page 1434

ENCRYPT= Table Option

Specifies whether to encrypt an output SAS data set.

Restrictions: Use with output data sets only.
This table option is not supported in DS2 programs that run in CAS.
In SPD Engine data sets, this table option cannot be used with COMPRESS=.

Data source: SAS data set, SPD Engine data set, SPD Server table

Syntax

ENCRYPT==AES | AES2 | NO | YES

Required Arguments

AES | AES2 | NO | YES
specifies whether to encrypt an output SAS data set.

AES or AES2
encrypts the file using the AES (Advanced Encryption Standard) algorithm. AES provides enhanced encryption by using SAS/SECURE software, which is included with Base SAS software. AES2 specifies to use a stronger key

generation algorithm. You must specify the ENCRYPTKEY= table option when you are using ENCRYPT=AES or ENCRYPT=AES2 or have a recorded key in the metadata-bound library where the data set is created. For more information, see "ENCRYPTKEY= Table Option" on page 1418.

Restriction SPD Engine and SPD Server do not support AES2 encryption.

Note AES2 encryption is supported beginning with SAS 9.4M5.

CAUTION **Record all ENCRYPTKEY= values when you are using ENCRYPT=AES or ENCRYPT=AES2.** If you forget to record the ENCRYPTKEY= value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= value. The following note is written to the log:

```
Note: If you lose or forget the ENCRYPTKEY= value, there will
be no way to open the file or recover the data.
```

NO
does not encrypt the data set.

YES
encrypts the data set using the SAS Proprietary algorithm. This encryption uses passwords that are stored in the data set. You must specify the READ= table option or the PW= table option at the same time that you specify ENCRYPT=YES. For more information, see "READ= Table Option" on page 1450 and "PW= Table Option" on page 1449.

Interaction Because the encryption method uses passwords, you cannot change *any* password on an encrypted file without re-creating the table.

CAUTION **Record all passwords when you are using ENCRYPT=YES.** If you forget the passwords, you cannot reset them without assistance from SAS. This is a time-consuming and resource-intensive process.

Details

When you use ENCRYPT=YES, the following rules apply:

- If the data file is encrypted, all associated indexes are also encrypted, except for SPD Server tables. SPD Server does not encrypt table indexes or metadata. Only table row data are encrypted.
- In order to copy an encrypted data file, the output engine must support encryption. Otherwise, the data file is not copied.
- Encryption requires approximately the same amount of CPU resources as compression.
- You cannot use PROC CPORT on SAS Proprietary encrypted SAS data files.

When you use ENCRYPT=AES or ENCRYPT=AES2, the following rules apply:

- You must have SAS/SECURE software.
- You must use the ENCRYPTKEY= table option when encrypting a table.

- You must use the ENCRYPTKEY= table option to enable decryption.
- A data set encrypted with AES encryption can be decrypted only by engines that support AES encryption.
- You cannot change the ENCRYPTKEY= value in an AES encrypted data file without re-creating the data file. The AUTHLIB procedure must be used to change the recorded key of a metadata-bound library.
- In Base SAS, data files with referential integrity constraints can use AES encryption. All primary key and foreign key data files must use the same encryption key that opens all referencing foreign key and primary key data files.

If a default Base SAS data set with AES2 encryption is copied to create a new SPD Engine data set or SPD Server table, the encryption converts to AES. A warning is written to the log.

Note: You can create an encrypted DS2 package or thread program by using the ENCRYPT argument in the PACKAGE and THREAD statements.

For more information about the encryption provided by SAS, see *Encryption in SAS*.

Example

This example creates an encrypted SAS data set:

```
data myfiles.mytable (encrypt=yes pw=mine);
      declare double j j2;
      method run();
         do j = 1 to 1000;
            j2 = 2*j;
            output;
         end;
      end;
   enddata;
run;
```

This example reads the encrypted SAS data set:

```
proc print data=myfiles.mytable (pw=mine obs=10);
run;
```

See Also

Statements:

- “PACKAGE Statement” on page 1286
- “THREAD Statement” on page 1318

Table Options:

- “ENCRYPTKEY= Table Option” on page 1418

ENCRYPTKEY= Table Option

Specifies a key value for AES encryption.

Restrictions: Use only with AES and AES2 encrypted data files.
This table option is not supported in DS2 programs that run in CAS.

Data source: SAS data set, SPD Engine data set, SPD Server table

Note: Check your log after this operation to ensure that the encrypt key values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in the *SAS Language Reference: Concepts*.

Syntax

ENCRYPTKEY=*key-value*

Required Argument

key-value
assigns an encrypt key value. You must specify the ENCRYPTKEY= table option when you are using ENCRYPT=AES or ENCRYPT=AES2, unless you are creating a data set in a metadata-bound library with a recorded key. The key value can be up to 64-bytes long.

Requirement The ENCRYPTKEY= key value can be created with or without quotation marks. For more information, see “Rules for Using Quotation Marks When Creating the ENCRYPTKEY Value” on page 1420.

Note When the ENCRYPTKEY= key value uses DBCS characters, the 64-byte limit applies to the character string after it has been transcoded to UTF-8 encoding. You can use the following DATA step to calculate the length in bytes of a key value in DBCS:

```
data _null_;
    key=length(unicodec('key-value','UTF8'));
    put 'key length=' key;
run;
```

Details

The Basics

CAUTION

Record the key value. If you forget the ENCRYPTKEY= key value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= key value because the key value is not stored with the data set. The following warning is written to the log:

```
WARNING: If you lose or forget the ENCRYPTKEY= value, there
will be not be any way to open the file or recover the data.
```

You must use the ENCRYPTKEY= option when you are creating or accessing a SAS data set with AES or AES2 encryption.

The ENCRYPTKEY= table option will not protect the table from deletion or replacement. Encrypted tables can be deleted by using any of the following scenarios without having to specify an ENCRYPTKEY= key value:

- the KILL option in PROC DATASETS
- the DROP statement in PROC SQL
- the DELETE statement in PROC DATASETS or the DELETE procedure

The ENCRYPTKEY= option prevents access to the contents of the file only. To protect the file from deletion or replacement, the file must also contain an ALTER= password.

The following DATASETS procedure statements require you to specify the ENCRYPTKEY= key value when working with protected files: AGE, AUDIT, APPEND, CHANGE, CONTENTS, MODIFY, REBUILD, and REPAIR statements.

```
append base=name data=name(encryptkey=key-value);
run;
```

The option can be specified either in parentheses after the name of the SAS data file or after a forward slash.

It is possible to use a macro variable as the ENCRYPTKEY= key value. When you specify a macro variable for the ENCRYPTKEY= key value, you must enclose the macro variable in double quotation marks. If you do not use the double quotation marks, unpredictable results can occur. The following example defines a macro variable and uses the macro variable as the ENCRYPTKEY= key value:

```
%let secret=myvalue;
data my.dsname(encrypt=aes encryptkey="&secret");
```

The following example uses the COPY statement from the DATASETS procedure and the SELECT statement:

```
copy in=OldLib out=NewLib;
    select salary(encryptkey=key-value);
run;
```

The option can be specified either in parentheses after the name of the table or after a forward slash.

CAUTION

When using referential integrity constraints, all primary key and foreign key tables that reference each other must use the same encryption key. For more information, see the topic on encryption and integrity constraints in *SAS Language Reference: Concepts*.

Note: When DS2 runs outside of SAS, such as in the SAS Federation Server and in grid computing environments, the SAS macro facility is not available and DS2 programs with macros fail to compile.

You cannot change the ENCRYPTKEY= value in an AES encrypted data file without re-creating the data file. The AUTHLIB procedure must be used to change the recorded key of a metadata-bound library.

Rules for Using Quotation Marks When Creating the ENCRYPTKEY Value

The ENCRYPTKEY= key value can be created with or without quotation marks following these rules:

no quotation marks

- alphanumeric characters and underscores only
- up to 64 bytes
- uppercase and lowercase letters
- must start with a letter
- cannot include blank spaces
- is not case sensitive

Here is an example:

```
encryptkey=key-value
encryptkey=key-value1
```

single quotation marks

- alphanumeric, special, and DBCS characters
- up to 64 bytes
- uppercase and lowercase letters
- can include blank spaces, but cannot contain all blanks
- is case sensitive

Here is an example:

```
encryptkey='key-value'
encryptkey='1234*#mykey'
```

double quotation marks

- alphanumeric, special, and DBCS characters
- up to 64 bytes
- uppercase and lowercase letters
- enables macro resolution
- can include blank spaces, but cannot contain all blanks
- is case-sensitive

Here is an example:

```
encryptkey="key-value"
encryptkey="1234*#mykey"

%let mykey=abcdefghi12;
encryptkey=&key-value
```

Example

This example uses the ENCRYPT=AES option.

```
data salary (encrypt=aes encryptkey=green overwrite=yes);
   dcl char(8) name;
   dcl double yrsal;
   dcl double bonuspct;
   method init();
      name='Muriel'; yrsal=34567; bonuspct=3.2;
        name='Bjorn'; yrsal=74644; bonuspct=2.5;
      name='Freda'; yrsal=38755; bonuspct=4.1;
        name='Benny'; yrsal=29855; bonuspct=3.5;
        name='Agnetha'; yrsal=70998; bonuspct=4.1;
   end;
run;
```

When you run the CONTENTS procedure, you will be prompted to specify the ENCRYPTKEY= key value.

```
proc contents data=salary;
run;
```

See Also

Table Options:

- “ENCRYPT= Table Option” on page 1415
- “SAS Data File Encryption” in *SAS Language Reference: Concepts*

ENDOBS= Table Option

Specifies the end observation number in a user-defined range of observations to be processed.

Valid in:	SELECT statement
Category:	Observation Control
Restrictions:	This table option is not supported on the CAS server. Use with input data sets only
Interaction:	Use in conjunction with STARTOBS= table option

Data source: SPD Engine data set, SPD Server table

Syntax

ENDOBS= *n*

Arguments

n
is the number of the end observation.

Details

The software processes all of the observations in the entire data set unless you specify a range of observations with the STARTOBS= or ENDOBS= options. If the STARTOBS= option is used without the ENDOBS= option, the implied value of ENDOBS= is the end of the data set. When both options are used together, the value of ENDOBS= must be greater than the value of STARTOBS=.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Server options can be used for WHERE clause processing in addition to table input operations. When ENDOBS= is used in a WHERE expression, the ENDOBS= value represents the last observation to process, rather than the number of observations to return.

Example

The following code shows how the ENDOBS= table option is specified in the FedSQL SELECT statement:

```
select * from spdslib.table1(startobs=3 endobs=4);
```

See Also

Table Options:

- "STARTOBS= Table Option" on page 1461

EXTENDOBSCOUNTER= Table Option

Specifies whether to extend the maximum observation count in a new output SAS data set.

Valid in: CREATE TABLE statement

Category: Table Control

Alias: EOC=

Default: YES

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set

Syntax

EXTENDOBSCOUNTER=YES | NO

Arguments

YES

requests an enhanced file format in a newly created SAS data set that counts observations beyond the 32-bit limitation. Although this SAS data set is created for an operating environment that stores the number of observations with a 32-bit integer, the data set behaves like a 64-bit file with respect to counters. This is the default value.

Restrictions EXTENDOBSCOUNTER=YES is valid only for an output SAS data set whose internal data representation stores the observation count as a 32-bit integer. EXTENDOBSCOUNTER= is ignored for SAS data sets with a 64-bit integer.

A SAS data set that is created with an extended observation count is incompatible with releases prior to SAS 9.3.

NO

specifies that the maximum observation count in a newly created SAS data file is determined by the long integer size for the operating environment. In operating environments with a 32-bit integer, the maximum number is 2**31-1 or approximately two billion observations (2,147,483,647). In operating environments with a 64-bit integer, the maximum number is 2**63-1 or approximately 9.2 quintillion observations.

Details

Historically, Base SAS had a limitation in which up to approximately two billion observations could be counted and fully supported for operating environments with a 32-bit long integer. The EXTENDOBSCOUNTER= table option extends the limit to match that of operating environments with a 64-bit long integer. EXTENDOBSCOUNTER=NO is provided for backward compatibility.

FETCH_NUMERIC_TYPE= Table Option

Specifies how to handle NUMERIC data from Google BigQuery.

Category: Table Control

Default: FLOAT64

Restrictions: This table option is not supported on the CAS server.
This table option is not supported for SAS Viya 3.5.

Data source: Google BigQuery

Note: Support for this option starts with the September 2024 update of SAS/ACCESS Interface to Google BigQuery software on SAS 9.4M7.

Syntax

FETCH_NUMERIC_TYPE=FLOAT64 | NUMERIC

Required Arguments

FLOAT64
specifies that NUMERIC(p,s) and BIGNUMERIC (p,s) values that are retrieved from Google BigQuery are treated as DOUBLEs. (The DS2 DOUBLE data type is mapped to the Google BigQuery FLOAT64 data type.)

NUMERIC
specifies that NUMERIC(p,s) and BIGNUMERIC (p,s) values that are retrieved from Google BigQuery are treated as NUMERIC.

Details

The default value improves read performance. This table option enables you to revert to NUMERIC behavior.

Data definition is not affected.

GP_DISTRIBUTED_BY= Table Option

Specifies the distribution key for the table being created.

Category: Table Control

Alias: DISTRIBUTED_BY=

Restriction: This table option is not supported on the CAS server.

Data source: Greenplum, HAWQ

Syntax

GP_DISTRIBUTED_BY= 'DISTRIBUTED BY (*column*[, ...*column*])
| DISTRIBUTED RANDOMLY'

Arguments

DISTRIBUTED BY (*column*, [...*column*])
specifies one or more DBMS column names to use as the distribution key.

DISTRIBUTED RANDOMLY
specifies to determine the column or set of columns to use to distribute table rows across database segments. This is known as round-robin distribution.

Details

DISTRIBUTED BY uses hash distribution with one or more columns declared as the distribution key. For the most even data distribution, the distribution key should be the primary key of the table or a unique column (or set of columns). If that is not possible, then you might choose DISTRIBUTED RANDOMLY, which sends the data round-robin to the segment instances. If a value is not supplied, then hash distribution is chosen using the primary key (if the table has one) or the first eligible column of the table as the distribution key.

DISTRIBUTED_BY can be submitted as shown above or within the DBCREATE_TABLE_OPTS= table option. Here is an example of how it is specified in DBCREATE_TABLE_OPTS=:

```
dbcreate_table_opts='distributed by ("b")'
```

See Also

Table Options:

- “DBCREATE_TABLE_OPTS= Table Option” on page 1411

IDXNAME= Table Option

Directs SAS to use a specific index to match the conditions of a WHERE clause.

Category: User Control of SAS Index Usage

Restrictions: This table option is not supported on the CAS server.
Use with input data sets only
Cannot be used with the IDXWHERE= table option

Data source: SAS data set

Syntax

IDXNAME= *index-name*

Arguments

index-name
specifies the name (up to 32 characters) of a simple or composite index for the SAS data set. SAS does not attempt to determine whether the specified index is the best one or whether a sequential search might be more resource efficient.

Interaction The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

Details

By default, to satisfy the conditions of a WHERE clause for an indexed SAS data set, SAS identifies zero or more candidate indexes that could be used to optimize the WHERE clause. From the list of candidate indexes, SAS selects the one that it determines will provide the best performance, or rejects all of the indexes if a sequential pass of the data is expected to be more efficient.

Because the index that SAS selects might not always provide the best optimization, you can direct SAS to use one of the candidate indexes by specifying the IDXNAME= table option. If you specify an index that SAS does not identify as a candidate index, then IDXNAME= table option does not process the request. That is, IDXNAME= does not allow you to specify an index that would produce incorrect results.

Comparisons

The IDXWHERE= table option enables you to override the SAS decision about whether to use an index.

Example

This example uses the IDXNAME= table option to direct SAS to use a specific index to optimize the WHERE clause. SAS then disregards the possibility that a

sequential search of the data set might be more resource efficient and does not attempt to determine whether the specified index is the best one. (Note that the EMPNUM index was not created with the NOMISS option.)

```
create table mydata.empnew
   as select * from mydata.employee (option idxname=empnum)
   where empnum < 2000;
```

See Also

Table options

- "IDXWHERE= Table Option" on page 1427

IDXWHERE= Table Option

Specifies whether SAS uses an index search or a sequential search to match the conditions of a WHERE clause.

Category: User Control of SAS Index Usage

Restrictions: This table option is not supported on the CAS server.
Use with input data sets only
SAS data sets: Cannot be used with IDXNAME=
SPD Engine data sets: IDXWHERE=NO cannot be used with WHERENOINDEX=

Data source: SAS data set, SPD Engine data set

Syntax

IDXWHERE= YES | NO

Arguments

YES
tells SAS to choose the best index to optimize a WHERE clause, and to disregard the possibility that a sequential search of the data set might be more resource-efficient. This is the default value.

NO
tells SAS to ignore all indexes and satisfy the conditions of a WHERE clause with a sequential search of the data set.

Notes You cannot use the IDXWHERE= table option to override the use of an index to process a BY statement.

You cannot use the WHERENOINDEX= table option when IDXWHERE=NO is used.

Details

By default, to satisfy the conditions of a WHERE clause for an indexed data set, the software decides whether to use an index or to read the data set sequentially. The software estimates the relative efficiency and chooses the method that is more efficient.

You might need to override the software's decision by specifying the IDXWHERE= table option because the decision is based on general rules that occasionally might not produce the best results. That is, by specifying the IDXWHERE= table option, you are able to determine the processing method.

Note: The specification is not a permanent attribute of the data set and is valid only for the current use of the data set.

Comparisons

The IDXNAME= table option (which is supported only for SAS data sets) enables you to direct SAS to use a specific index.

The WHERENOINDEX= table option enables you to specify a list of indexes to exclude when making WHERE expression evaluations.

Examples

Example 1: Specifying Index Usage

This example uses the IDXWHERE= table option to tell SAS to decide which index is the best to optimize the WHERE clause. SAS then disregards the possibility that a sequential search of the data set might be more resource-efficient:

```
create table mydata.empnew
   as select * from mydata.employee {option idxwhere=yes}
   where empnum < 2000;
```

Example 2: Specifying No Index Usage

This example uses the IDXWHERE= table option to tell SAS to ignore any index and to satisfy the conditions of the WHERE clause with a sequential search of the data set:

```
   create table mydata.empnew
      as select * from mydata.employee {option idxwhere=no}
```

```
where empnum < 2000;
```

See Also

Table options:

- “IDXNAME= Table Option” on page 1425
- “WHERENOINDEX= Table Option” on page 1492

IN= Table Option

Creates an integer variable that indicates whether the table contributed data to the current row.

Restriction: Use with the SET and MERGE statements only.

Data source: All

Syntax

IN=*variable*

Required Argument

variable
names the new variable whose value indicates whether that input table contributed data to the current row. Within a DS2 program, the value of the variable is 1 if the table contributed to the current row, and 0 otherwise.

Interaction If the variable is not explicitly declared, it is automatically declared in the local scope of the SET or MERGE statement as INTEGER.

Data type BIGINT, INTEGER, SMALLINT, TINYINT

Details

Specify the IN= table option in parentheses after a table name in the SET or MERGE statements. Values of IN= variables are available to program statements during the time the DS2 program is running, but the variables are not included in the table that is being created. To capture the value of the IN= variable in the table being created, assign it another variable.

When you use IN= with BY-group processing, the IN= variable is set to 1 if that table contributed to the current row. The IN= variable is set to 0 if the table did not

contribute to the current row. The IN= variable is always set to 1 or 0 by the SET or MERGE statement, but it can be changed by subsequent programming logic.

Note: If the IN= variable is set before the SET or MERGE statement, that value is lost during execution of the SET or MERGE statement.

Example

The following example illustrates the IN= table option.

```
data _null_;
   dcl int gro;
   method run();
      dcl smallint gpo;
      dcl tinyint gpf;
      set gas_price_option (in=gpo) gas_rbid_option (in=gro)
         gas_price_forward (in=gpf);
      put gpo= gro= gpf=;
   end;
enddata;
```

See Also

Statements:

- “DATA Statement” on page 1220
- “SET Statement” on page 1305

INLINE Table Option

Specifies that the package or thread source code is not saved to a table for reuse and is validated and compiled only when loaded by a data program.

Restrictions: This table option is not supported on the CAS server.
Use with PACKAGE and THREAD statements only.

Data source: All

Syntax

INLINE

Details

By default, DS2 attempts to validate and compile package and thread source code and save it to a table. When you specify the INLINE table option, the following actions occur:

- Package and thread source code is stored in memory.
- Package and thread source code is not written to a table for future use.
- Package and thread source code is not validated and compiled if it is not loaded by a data program.

When running with PROC DS2, inlined package and thread source code is available only from the point that it is defined until the end of the current RUN block. At the end of a RUN block, inlined package and thread source code is deleted.

Consider the following example. The declaration of **mypkg** variable **p1** succeeds because the data program is defined in the same RUN block as package **mypkg**. The declaration of **mypkg** variable **p2** fails because the package, **mypkg**, is not defined in this RUN block.

```
package mypkg / inline;
method mypkg();
   put 'constructing mypkg instance';
end;
endpackage;

data _null_;
   dcl package mypkg p1();  /* OK */
enddata;

run;

data _null_;
   dcl package mypkg p2();  /* ERROR */
enddata;

run;
```

Example

In the following example, the INLINE table option is used to suppress saving the package and thread source code.

```
package pkg1 / inline;
  method pkg1();
    put 'hello from pkg1';
  end;
endpackage;

package pkg2 / inline;
  method pkg2();
    dcl package pkg1 p1();
    put 'hello from pkg2';
```

```
  end;
endpackage;

package pkg3 / inline;
  method pkg3();
    dcl package pkg1 p1();
    dcl package pkg2 p2();
    put 'hello from pkg3';
  end;
endpackage;

thread thd1 / inline;
  dcl double x;
  method init();
    put 'BEGIN EXPECTED OUTPUT';
    put 'hello from thd1 ';
    put 'hello from pkg1 ';
    put 'hello from pkg1 ';
    put 'hello from pkg2 ';
    put 'hello from pkg3 ';
    put 'hello from thd1';
    put 'END EXPECTED OUTPUT';
    put;
    put 'BEGIN TEST OUTPUT';
  end;

  method run();
    dcl package pkg3 p3();
  end;

  method term();
    put 'END TEST OUTPUT';
  end;
endthread;

data _null_;
  dcl thread thd1 t();

  method run();
    set from t threads=1;
  end;
enddata;
run;
```

The following lines are written to the SAS log:

```
BEGIN EXPECTED OUTPUT
hello from thd1
hello from pkg1
hello from pkg1
hello from pkg2
hello from pkg3
hello from thd1
END EXPECTED OUTPUT

BEGIN TEST OUTPUT
hello from pkg1
hello from pkg1
hello from pkg2
hello from pkg3
END TEST OUTPUT
```

If the INLINE table option were not used, the package and thread code would be saved to a temporary table and you would also see these lines written to the SAS log:

```
NOTE: Created package pkg1 in data set work.pkg1.
NOTE: Created package pkg2 in data set work.pkg2.
NOTE: Created package pkg3 in data set work.pkg3.
NOTE: Created thread thd1 in data set work.thd1.
```

IOBLOCKSIZE= Table Option

Specifies the number of rows in a block to be used in an I/O operation.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SPD Engine data set, SPD Server table

Syntax

IOBLOCKSIZE= *n*

Arguments

n

specifies the size of the block in bytes. The default value is smallest block size supported by the data source.

Details

The software reads and stores rows in the table in blocks. IOBLOCKSIZE= is useful on compressed or encrypted tables. The software does not use IOBLOCKSIZE= on noncompressed or nonencrypted tables.

For tables that you compress or encrypt, the IOBLOCKSIZE= specification determines the number of rows to include in the block. The specification applies to block compression as well as data I/O to and from disk. The IOBLOCKSIZE= value affects the table's organization on disk.

When using compression or encryption, specify an IOBLOCKSIZE= value that complements how the data is to be accessed, sequentially or randomly. Sequential access or operations requiring full table scans favor a large block size. In contrast, random access favors a smaller block size. On SPD Engine, a large block size would be 131,072 bytes. The smallest allowed value is 32,768 bytes. For SPD Server, a large block size is 64,000 bytes. The smallest allowed value is 8,000 bytes.

See Also

Table Options:

- "COMPRESS= Table Option" on page 1406
- "ENCRYPT= Table Option" on page 1415
- "PADCOMPRESS= Table Option" on page 1441

KEEP= Table Option

For an input table, specifies the columns to process; for an output table, specifies the columns to write to the table.

Data source: All

Syntax

KEEP=(*column-list*)

Required Argument

column-list
specifies the names of the columns to keep in the output table.

Restriction Numbered range lists in the format *col1–col5* and name prefix lists in the format *col:* are not supported.

Details

The KEEP= table option specifies that all columns in the *column-list* should be included in the creation of output rows. Normally, all columns in the program data vector are included in the output rows. When the KEEP= table option is specified, all columns that are not included in the KEEP= table option are dropped from the output rows.

If the KEEP= table option is associated with an input table, only the columns that are specified by the KEEP= table option are available for processing during program execution.

Only global variables, by default, are included in the output. Local variables used for program loops and indexes do not need to be explicitly dropped from the output.

Comparisons

The KEEP= table option differs from the KEEP statement in these ways:

- In DS2 programs, the KEEP= table option can apply to both input and output tables. The KEEP statement applies only to output tables.
- In DS2 programs, when you create multiple output tables, use the KEEP= table option to write different columns to different tables. The KEEP statement applies to all output tables.
- The DROP= table option specifies columns to omit during processing or to omit from the output table.

Example

In this example, only IDNUM and SALARY are read from PAYROLL, and they are the only variables in PAYROLL that are available for processing.

```
data bonus;
   method run();
      set payroll(keep=(idnum salary));
      bonus=salary*1.1;
   end;
enddata;
```

See Also

Table Options:

- “DROP= Table Option” on page 1413

LABEL= Table Option

Specifies a label for a table.

Data source: SAS data set, SPD Engine data set

Syntax

LABEL=*'label'*

Required Argument

'label'

specifies a text string of up to 256 characters. If the label text contains single quotation marks, use double quotation marks around the label, or use two single quotation marks in the label text and enclose the string in single quotation marks. To remove a label from a table, assign a label that is equal to a blank that is enclosed in quotation marks.

Details

You can use the LABEL= option on both input and output tables. When you use LABEL= on input tables, it assigns a label for the table for the duration of the DS2 program. When it is specified for an output table, the label becomes a permanent part of that table.

A label assigned to a table remains associated with that table when you update a table in place. However, a label is lost if you use a table with a previously assigned label to create a new table in the DS2 program. For example, a label previously assigned to table ONE is lost when you create the new output table ONE:

```
data one;
   set one;
enddata;
```

Comparisons

The LABEL= option in the HAVING clause of the DECLARE statement also enables you to assign labels to variables.

Example

These examples assign labels to tables:

```
data w2 (label = '2009 W2 Info, Hourly);
data new (label = 'Sales'' list');
data acct (label = "Hillside''s Daily Account");
data sales (label = 'May (South)');
```

LOCKTABLE= Table Option

Places shared or exclusive locks on tables.

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set

Syntax

LOCKTABLE=SHARE | EXCLUSIVE

Required Argument

SHARE | EXCLUSIVE
specifies whether a table is locked in shared mode or exclusively.

Note: In shared mode, other users or processes can read data from the tables, but are prevented from updating data. If a table is locked exclusively, other users cannot access any table that you open.

Details

You can lock tables only if you are the owner or have been granted the necessary privilege.

If you use PROC DS2, the default value for the LOCKTABLE option is EXCLUSIVE.

ORHINTS= Table Option

Specifies Oracle hints to pass to Oracle from FedSQL.

Valid in: DELETE statement, INSERT statement, SELECT statement, UPDATE statement

Category: Data Control

Restriction: This table option is not supported on the CAS server.

Data source: Oracle

Syntax

ORHINTS=*'/* Oracle-hint */'*

Arguments

Oracle-hint
specifies an Oracle hint for the FedSQL language processor to pass to the DBMS as part of a query. If you do not specify a hint, FedSQL does not send any hints to Oracle. For more information, see *SAS Federation Server: Administrator's Guide*.

Details

The ORHINTS= table option is used in conjunction with the DRIVER_TRACE= and DRIVER_TRACEFILE= data source connection options. PROC FEDSQL and PROC DS2 do not support use of data source connection options.

Example: Using the ORHINTS= Option

These examples show how to specify ORHINTS=:

```
select * from test{options orhints='/* dummy hint */'} ;

update test{options orhints='/* dummy hint */'}  set c1=1;

insert into test{options orhints='/* dummy hint */'}  values (100);

delete from test{options orhints='/* dummy hint */'} ;
```

ORNUMERIC= Table Option

Specifies how numbers read from or inserted into the Oracle NUMBER column are treated.

Category: Table Control

Default: YES

Restriction: This table option is not supported on the CAS server.

Data source: Oracle

Data type: DECIMAL, NUMERIC

Syntax

ORNUMERIC=YES | NO

Arguments

NO
Indicates that the numbers are treated as TKTS_DOUBLE values. They might not have precision beyond 14 digits.

YES
Indicates that non-integer values with explicit precision are treated as TKTS_NUMERIC values. This is the default setting.

Details

This option defaults to YES so that a NUMBER column with precision or scale is described as TKTS_NUMERIC. This option can be specified as both a connection option and a table option. When specified as both connection and table option, the table option value overrides the connection option.

OVERWRITE= Table Option

For a table, drops the output table before the replacement output table is populated with rows; for packages and threads, drops the existing package or thread if a package or thread by the same name exists.

Restriction: This table option is not supported on the CAS server.

Data source: All

Syntax

OVERWRITE=YES | NO

Required Argument

YES | NO
specifies whether the output table is deleted before a replacement output table is created or whether a package or thread is dropped.

CAUTION

For tables, use the OVERWRITE=YES statement only with data that is backed up or with data that you can reconstruct. Because the output table is deleted first, data will be lost if a failure occurs while the output table is being written.

Default NO

Details

Details for Tables

By default, in DS2, a table is not overwritten unless the OVERWRITE= table option is set to YES. If the output table exists and the OVERWRITE= table option is set to NO, an error occurs because the existing table is not overwritten.

Details for Packages

If you set OVERWRITE=YES in a PACKAGE statement and a DS2 thread or a regular table exists with the same name as the package being created, the table is not dropped. Only the package is dropped.

Details for Threads

If you set OVERWRITE=YES in a THREAD statement and a DS2 package or a regular table exists with the same name as the thread being created, the table will not be dropped. Only the thread is dropped.

Examples

Example 1: Using DROP, KEEP, and OVERWRITE

The following example uses the DROP=, KEEP=, and OVERWRITE= table options for tables **a** and **b**.

```
data
   a(keep=x overwrite=yes)
   _rowset_(drop=(x z))
   b(keep=z overwrite=yes);
   dcl double x y z;
   method init();
      do x = 1 to 10;
         y = 2*x;
         z = 3*x;
         output;
      end;
   end;
enddata;
run;
data;
```

```
   method run();
      set a;
   end;
enddata;
run;
data;
   method run();
      set b;
   end;
enddata;
run;
```

Example 2: Overwriting a Table

The following example creates a table, and then attempts to overwrite it without OVERWRITE=YES.

```
data a;
   method init();
      do x = 1 to 10;
         y = 2*x;
         z = 3*x;
         output;
      end;
   end;
enddata;
run;
/* This program fails because it is impossible to overwrite table A */
data a;
   method run();
      x = y + z;
      output;
   end;
enddata;
run;
/* This program deletes (drops) table A before attempting to create it, */
/*  so the program executes without error.  If there is an error during
/* execution, the old version of A is lost.  */
data a(overwrite=yes);   method run();
      x = y + z;
      output;
   end;
enddata;
run;
```

PADCOMPRESS= Table Option

Specifies the number of bytes to add to compressed blocks in an SPD Engine data set that is opened for OUTPUT or UPDATE.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SPD Engine data set

Syntax

PADCOMPRESS= *n*

Arguments

n
specifies the number of bytes to add. The default number is 0 (zero).

Details

Compressed SPD Engine data sets occupy blocks of space on the disk. The size of a block is derived from the IOBLOCKSIZE= table option that is specified when the data set is created. When the data set is updated, a new block fragment might need to be created to hold the update. More updates might then create new fragments, which, in turn, increases the number of I/O operations needed to read a data set.

By increasing the block padding in certain situations where many updates to the data set are expected, fragmentation can be kept to a minimum. However, adding padding can waste space if you do not update the data set.

You must weigh the cost of padding all compression blocks against the cost of possible fragmentation of some compression blocks.

Specifying the PADCOMPRESS= table option when you create or update a data set adds space to all of the blocks as they are written back to the disk. The PADCOMPRESS= setting is not retained in the data set's metadata.

See Also

Table Options:

- "COMPRESS= Table Option" on page 1406
- "IOBLOCKSIZE= Table Option" on page 1433

PARTITION_KEY= Table Option

Specifies the column name to use as the partition key for creating fact tables.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: Aster

Syntax

PARTITION_KEY= "*column-name*"

Arguments

column-name
specifies the name of the column, in quotation marks.

Details

Aster uses two table types: dimension and fact. To create a fact table in Aster without error, you must set the PARTITION_KEY= table option.

PARTSIZE= Table Option

Specifies the size of the table partitions.

Valid in: CREATE TABLE statement

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SPD Engine data set, SPD Server table

Syntax

PARTSIZE= *n*

Arguments

n
specifies the size of the partition.

Requirements The size can be specified in megabytes, gigabytes, and terabytes. If *n* is specified without M, G, or T, the default is megabytes. For example, PARTSIZE=128 is the same as PARTSIZE=128M. For SPD Engine data sets, the default value is 128 megabytes. The maximum value is 8,796,093,022,207 megabytes. For SPD Server tables, the default value is the setting of the MINPARTSIZE=

server option. If the MINPARTSIZE= server option is not set, the default is 16 megabytes.

When creating an SPD Engine data set, if the row length is greater than 65K, you might find that the PARTSIZE= that you specify and the actual partition size do not match. To get these numbers to match, specify a PARTSIZE= that is a multiple of 32 and the row length.

Details

For more information, see "PARTSIZE= Data Set Option in *SAS Scalable Performance Data Engine: Reference* and "PARTSIZE= Table Option" in *SAS Scalable Performance Data Server: User's Guide*, as appropriate.

PASSWORD= Table Option

Specifies the password for the HDFS user.

Category: Bulk Loading

Alias: PASS=, PWD=

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403.

Interaction: Used in conjunction with "USER= Table Option" on page 1491.

Data source: Impala

Syntax

PASSWORD=*"value"*

Arguments

"value"
specifies the password for the HDFS user.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

- “USER= Table Option” on page 1491

PICKLIST= Table Option

Specifies the picklist to use for the bulk-loading operation.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1403.

Data source: Impala

Syntax

PICKLIST=*"filename"*

Arguments

"filename"
specifies the full pathname to a text file that contains the Hadoop picklist.

Default When this table option is omitted, the file “hadoopbasics/hadoopwrapr.txt” is used.

Details

Use the PICKLIST= option when you want to override the default picklist of “hadoopbasics/hadoopwrapr.txt”.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400

POINTOBS= Table Option

Specifies whether SAS creates compressed data sets whose observations can be randomly accessed or sequentially accessed.

Category: Table Control

Restrictions: This table option is not supported on the CAS server.
Use with output tables only.

Interaction: Used with COMPRESS= table option

Data source: SAS data set

Syntax

POINTOBS= YES | NO

Syntax Description

YES
causes the FedSQL language to produce a compressed data set that might be randomly accessed by observation number. This is the default.

Here are examples of accessing data directly by observation number:

- the POINT= option of the MODIFY and SET statements in the DATA step
- going directly to a specific observation number with PROC FSEDIT.

> **TIP** Specifying POINTOBS=YES does not affect the efficiency of retrieving information from a data set. It does increase CPU usage by approximately 10% when creating a compressed data set and when updating or adding information to it.

NO
suppresses the ability to randomly access observations in a compressed data set by observation number.

> **TIP** Specifying POINTOBS=NO is desirable for applications where the ability to point directly to an observation by number within a compressed data set is not important. If you do not need to access data by observation number, then you can improve performance by approximately 10% by specifying POINTOBS=NO:
>
> - when creating a compressed data set
> - when updating or adding observations to it

Details

Note that REUSE=YES takes precedence over POINTOBS=YES. For example:

```
create table test{options compress=yes pointobs=yes reuse=yes};
```

This example code results in a data set that has POINTOBS=NO. Because POINTOBS=YES is the default when you use compression, REUSE=YES causes POINTOBS= to change to NO.

See Also

Table Options:

- “COMPRESS= Table Option” on page 1406
- “REUSE= Table Option” on page 1454

POST_TABLE_OPTS= Table Option

Allows database-specific options to be placed after the table name in the DATA statement.

Category:	Table Control
Default:	None
Restriction:	This table option is not supported on the CAS server.
Data source:	Hive, Spark
Note:	This option is available beginning with SAS 9.4M5.
See:	“DBCREATE_TABLE_OPTS= Table Option” on page 1411 “PRE_TABLE_OPTS= Table Option” on page 1448

Syntax

POST_TABLE_OPTS=*"DBMS-SQL-options"*

Required Argument

"DBMS-SQL-options"
specifies additional database-specific options to be placed after the table name in a DATA statement. You can use single or double quotation marks around the DBMS value.

Details

You can use the POST_TABLE_OPTS= table option alone or with these related table options: PRE_TABLE_OPTS= and POST_STMT_OPTS=. POST_STMT_OPTS= is an alias for DBCREATE_TABLE_OPTS=. For example, you can supply database options according to these templates:

```
data mylib.myexample (pre_table_opts='external'
                      post_table_opts='stored as orc');
          dcl float f;
          method init();
            f=11.01; output;
          end;
        enddata;
        run;

data mylib.myexample (pre_table_opts='external'
                      post_table_opts='stored as orc')
              post_stmt_options="orc"} col1(int);
          dcl float f;
          method init();
            f=11.01; output;
          end;
        enddata;
        run;
```

When all three table options are specified together, they are processed as follows:

```
DATA [PRE_TABLE_OPTIONS] mylib.myexample (col1 int) [POST_TABLE_OPTS]
[POST_STMT_OPTS/DBCREATE_TABLE_OPTS]
```

PRE_TABLE_OPTS= Table Option

Allows database-specific options to be placed before the table name in the DATA statement.

Category:	Table Control
Default:	None
Restriction:	This table option is not supported in the CAS server.
Data source:	Hive, Spark
Note:	This option is available beginning with SAS 9.4M5.
See:	“DBCREATE_TABLE_OPTS= Table Option” on page 1411 “POST_TABLE_OPTS= Table Option” on page 1447

Syntax

PRE_TABLE_OPTS=*"DBMS-SQL-options"*

Required Argument

"DBMS-SQL-options"
specifies additional database-specific options to be placed before the table name in a DATA statement. You can use single or double quotation marks around the DBMS value.

Details

You can use the PRE_TABLE_OPTS= table option alone or with these related table options: POST_TABLE_OPTS= and POST_STMT_OPTS=. POST_STMT_OPTS= is an alias for DBCREATE_TABLE_OPTS=. For example, you can supply database options according to these templates:

```
data mylib.myexample (pre_table_opts='external'
                      post_table_opts='stored as orc');
           dcl float f;
           method init();
             f=11.01; output;
           end;
         enddata;
         run;


data mylib.myexample (pre_table_opts='external'
                      post_table_opts='stored as orc')
                      post_stmt_options="orc"} col1(int);
           dcl float f;
           method init();
             f=11.01; output;
           end;
         enddata;
         run;
```

When all three table options are specified together, they are processed as follows:

```
DATA [PRE_TABLE_OPTIONS] mylib.myexample (col1 int) [POST_TABLE_OPTS]
[POST_STMT_OPTS/DBCREATE_TABLE_OPTS]
```

PW= Table Option

Assigns a READ, WRITE, and ALTER password to a SAS data set or an SPD Engine data set and enables access to the password-protected file. Specifies a key value for accessing an encrypted SPD Server table.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set, SPD Server table

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Programmer’s Guide: Essentials*.

Syntax

PW= *password* | *key-value*

Arguments

password
must be a valid SAS name.

key-value
must be a valid SAS name.

Details

The PW= table option applies to a SAS data set, an SPD Engine data set, and an SPD Server table. For a SAS data set and SPD Engine data set, you can use this option to assign a password to the data set or to access a password-protected data set. When a data set that is protected by a password is replaced, the new data set inherits the password.

Note: A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

For SPD Server tables, you can use the PW= option to access an SPD Server table that is already encrypted. The PW= option enables access to table files that are protected by the SAS proprietary encryption algorithm.

Examples

Example 1: Assigning a Password or Encryption Key with PW=

```
create table myfiles.mytable {options pw=green}(column1 double);
```

Example 2: Specifying a Password or Encryption Key with PW=

```
select * from myfiles.mytable {option pw=green};
```

READ= Table Option

Assigns a READ password to a SAS file that prevents users from reading the file, unless they enter the password.

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts*.

Syntax

READ=*read-password*

Required Argument

read-password
must be a valid SAS name.

Details

The READ= option applies to all types of SAS files except catalogs. You can use this option to assign a password to a SAS file or to access a Read-protected SAS file. When the code is written to the SAS log, the password is blotted out. Here is an example:

```
declare package sales (read=XXXXXXX);
```

Note: A SAS password does not control access to a SAS file beyond SAS. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

See Also

Table Options:

- “ENCRYPT= Table Option” on page 1415
- “PW= Table Option” on page 1449
- “WRITE= Table Option” on page 1493

READMODE= Table Option

Specifies the method to use when moving data from Google BigQuery into SAS.

Category:	Table Access
Default:	STANDARD
Restriction:	This table option is not supported on the CAS server.
Data source:	Google BigQuery

Syntax

READMODE=STANDARD | STORAGE

Optional Arguments

STANDARD
uses the SAS/ACCESS method to move data into SAS.

STORAGE
uses the Storage API for Google to move data into SAS.

Details

READMODE=STORAGE is faster than READMODE=STANDARD.

RENAME= Table Option

Changes the name of a column.

Data source: All

Syntax

RENAME=(*old-name* { = | **AS** } *new-name* [...*old-name* { = | AS } *new-name*])

Required Arguments

old-name
the column that you want to rename.

new-name
the new name of the column. It must be a valid name for the data source.

Details

The RENAME= table option enables you to change the names of one or more columns.

If you use RENAME= when you create a table, the new column name is included in the output table. If you use RENAME= on an input table, the new name is used in DS2 programming statements.

If you use RENAME= in the same DS2 program with either the DROP= or the KEEP= table option, the DROP= and the KEEP= table options are applied before RENAME=. You must use the old name in the DROP= and KEEP= table options. You cannot drop and rename the same column in the same statement.

In addition to changing the name of a column, RENAME= also changes the label for the column.

Comparisons

- The RENAME= table option differs from the RENAME statement in the following ways.
  - The RENAME statement applies to all output tables. If you want to rename different columns in different tables, you must use the RENAME= table option.
  - The RENAME= table option enables you to specify the columns that you want to rename for each input or output table. Use it in input tables to rename columns before processing.
  - If you use both the RENAME statement and RENAME= output table option, the RENAME statement has precedence. If X is renamed to Y with a RENAME statement and X is renamed to Z with a RENAME= table option, the RENAME statement takes precedence and X is renamed to Y.
- Use the RENAME statement or the RENAME= table option when program logic requires that you rename columns such as two input tables that have columns with the same name.

Examples

Example 1: Renaming a Column at Time of Output

This example uses RENAME= in the DATA statement to show that the column is renamed when it is written to the output table. The column keeps its original name, X, during DS2 processing.

```
data two(rename=(x=keys))
   method run();
      set one;
      z=x+y;
   run;
enddata;
```

Example 2: Renaming a Column at Time of Input

This example renames column X to a column named KEYS in the SET statement, which is a rename before DS2 processing. KEYS, not X, is the name to use for the column for DS2 processing.

```
data three;
   method run();
      set one(rename=(x AS keys));
      z=keys+y;
   run;
enddata;
```

See Also

Statements:

- “RENAME Statement” on page 1296

REUSE= Table Option

Specifies whether new rows can be written to freed space in a compressed SAS data set.

Category: Table Control

Restrictions: This table option is not supported on the CAS server.

Use with output data sets only

Data source: SAS data set

Syntax

REUSE= NO | YES

Arguments

NO
does not track and reuse space in a compressed SAS data set. New rows are appended to the existing data set. Specifying NO results in less efficient data storage if you delete or update many rows in the data set.

YES
tracks and reuses space in a compressed SAS data set. New rows are inserted in the space that is freed when other rows are updated or deleted.

If you plan to use operations that add rows to the end of a compressed data set, use REUSE=NO. REUSE=YES causes new rows to be added wherever there is space in the file, not necessarily at the end of the file.

Details

By default, new rows are appended to an existing compressed data set. If you want to track and reuse free space by deleting or updating other rows, use the REUSE= table option when you create a compressed SAS data set.

The REUSE= table option has meaning only when you are creating a new data set with the COMPRESS=YES option. Using the REUSE= table option when you are accessing an existing SAS data set has no effect.

See Also

Table Options:

- "COMPRESS= Table Option" on page 1406
- "POINTOBS= Table Option" on page 1445

RS_AWS_CONFIG= Table Option

Specifies the location of the Amazon Web Services (AWS) configuration file.

Category:	Bulk Loading
Default:	~/.aws/config
Restrictions:	This table option is supported only in FedSQL statements that are submitted from DS2. This table option is not supported on the CAS server.
Requirements:	To specify this option, you must first specify BULKLOAD=YES. This option is specified within the "BULKOPTS= Table Option" on page 1403.
Data source:	Amazon Redshift
Note:	Support for this option was added in SAS 9.4M9.
See:	BL_AWS_CONFIG_FILE= LIBNAME Statement Option

Syntax

RS_AWS_CONFIG='*path-and-file-name*'

Required Argument

path-and-file-name
is the full directory path and name of the AWS configuration file.

Example

```
data x.mytable (BULKLOAD=YES BULKOPTS=(RS_AWS_CONFIG='/my/.aws/config'
RS_AWS_CREDENTIALS='/my/.aws/creds' RS_BUCKET=mybucket RS_DEFAULT_DIR='/temp/'
RS_KEY='99999' RS_SECRET='12345' RS_REGION='us-east-1'));
   method run();
   set rstfile.othertable_1000000;
   end;
enddata;
run;
```

RS_BUCKET= Table Option

Specifies the name of the AWS S3 bucket to use when bulk loading data.

Category: Bulk Loading

Default: None

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.

This table option is not supported on the CAS server.

Requirements: To specify this option, you must first specify BULKLOAD=YES.

This option is specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option was added in SAS 9.4M9.

Syntax

RS_BUCKET=*'bucket-name'*

Required Argument

bucket_name
specifies an Amazon S3 bucket.

RS_CREDENTIALS_FILE= Table Option

Specifies the AWS credentials file to use when bulk loading data.

Category: Bulk Loading

Default: ~/.aws/credentials

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirements: To specify this option, you must first specify BULKLOAD=YES.
This option is specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option starts in SAS 9.4M9.

Syntax

RS_CREDENTIALS_FILE=*'path-and-file-name'*

Required Argument

path-and-file-name
specifies the full directory path and name of the AWS credentials file.

RS_DEFAULT_DIR= Table Option

Specifies where intermediate bulk-load files are created.

Category: Bulk Loading

Default: None

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirements: To specify this option, you must first specify BULKLOAD=YES.
This option is specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option starts in SAS 9.4M9.

Syntax

RS_DEFAULT_DIR=*'directory-path'*

Required Argument

directory-path
specifies the host-specific directory path where intermediate bulk-load files are to be created.

Note The directory path must end with a slash. Example: `BL_DEFAULT_DIR='/temp/'`

RS_DELETE_DATAFILE= Table Option

Specifies whether to delete the data file or all files created by the bulk-loading facility.

Category: Bulk Loading

Default: TRUE

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirement: This option must be specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option starts in SAS 9.4M9.

Syntax

RS_DELETE_DATAFILE=TRUE | FALSE

Required Arguments

TRUE
specifies to delete all files created by the bulk-load facility.

FALSE
specifies to delete only the data file.

RS_KEY= Table Option

Specifies the Amazon Web Services access key that is used with key-based access control. If you are using temporary token credentials, this is the temporary access key ID.

Category: Bulk Loading

Default: None

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirements: To specify this option, you must first specify BULKLOAD=YES.
This option is specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option starts in SAS 9.4M9.

Syntax

RS_KEY='*key-value*'

Required Argument

key-value
specifies an AWS key that you use to access the AWS environment.

Details

For more information about managing access keys, see the Amazon Web Services (AWS) documentation.

RS_REGION= Table Option

Specifies the AWS region for Amazon S3 data.

Category: Bulk Loading

Default: None

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirements: To specify this option, you must first specify BULKLOAD=YES.
This option is specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option starts in SAS 9.4M9.

Syntax

RS_REGION='*region*'

Required Argument

region
specifies the AWS region from which S3 data is being loaded. You must specify the region when using the S3 tool to load data.

RS_SECRET= Table Option

Specifies the secret access key that is used to bulk load data.

Category: Bulk Loading

Default: None

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirements: To specify this option, you must first specify BULKLOAD=YES.
This option is specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option was added in SAS 9.4M9.

Syntax

RS_SECRET=*'secret-access-key'*

Required Argument

secret-access-key
specifies the secret access key that is associated with the AWS key ID specified in the RS_KEY= option.

Tip To increase security, you can encode the key value by using PROC PWENCODE.

Details

When the secret access key is encoded with PROC PWENCODE, the value is decoded by SAS/ACCESS before passing the value to the data source.

RS_TOKEN= Table Option

Specifies a temporary token that is associated with the temporary credentials that you specify with the RS_KEY= and RS_SECRET= table options.

Category: Bulk Loading

Default: None

Restrictions: This table option is supported only in FedSQL statements that are submitted from DS2.
This table option is not supported on the CAS server.

Requirements: To specify this option, you must first specify BULKLOAD=YES.
This option is specified within the "BULKOPTS= Table Option" on page 1403.

Data source: Amazon Redshift

Note: Support for this option starts in SAS 9.4M9.

Syntax

RS_TOKEN='*token*'

Required Argument

token
specifies the temporary token that is associated with the temporary credentials specified in the RS_KEY= and RS_SECRET= table options.

STARTOBS= Table Option

Specifies the starting row number in a user-defined range of rows to be processed.

Valid in: SELECT statement

Category: Observation Control

Restrictions: This table option is not supported on the CAS server.
Use with input data sets only

Data source: SPD Engine data set, SPD Server tables

Syntax

STARTOBS= *n*

Arguments

n
is the number of the starting row.

Details

The software processes the entire data set unless you specify a range of rows with the STARTOBS= and ENDOBS= options. If the ENDOBS= option is used without the STARTOBS= option, the implied value of STARTOBS= is 1. When both options are used together, the value of STARTOBS= must be less than the value of ENDOBS=.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= options can be used for WHERE clause processing in addition to table input operations. When STARTOBS= is used in a WHERE expression, the STARTOBS= value represents the first row on which to apply the WHERE expression.

See Also

Table Options:

- "ENDOBS= Table Option" on page 1421

TABLE_TYPE= Table Option

Specifies the type of table storage FedSQL will use when creating tables in SAP HANA.

Valid in:	CREATE TABLE statement
Category:	Data Access
Restriction:	This table option is not supported on the CAS server.
Interactions:	GLOBAL and GLOBAL TEMPORARY have the same behavior LOCAL and LOCAL TEMPORARY have the same behavior
Data source:	SAP HANA

Syntax

TABLE_TYPE= [COLUMN] [GLOBAL] [GLOBAL TEMPORARY] [LOCAL] [LOCAL TEMPORARY] [ROW]

Arguments

COLUMN
creates a table using column-based storage in SAP HANA.

GLOBAL | GLOBAL TEMPORARY
creates a global, temporary table in SAP HANA. The tables are globally available; however, the data is visible only in the current session.

LOCAL | LOCAL TEMPORARY
creates a local, temporary table in SAP HANA. The table definition and data are visible only in the current session.

ROW
creates a table using row-based storage in SAP HANA.

Details

The SAP HANA TABLE_TYPE= option is available as a connection option and as a table option. If neither are specified, tables that are created in SAP HANA follow the SAP HANA default for row or column store.

See Also

Table Options:

- "DBCREATE_TABLE_OPTS= Table Option" on page 1411

TD_BUFFER_MODE= Table Option

Specifies whether the LOAD method is used.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	Teradata

Syntax

TD_BUFFER_MODE= YES | NO

Arguments

YES
enables the bulk load feature. This option must be set to YES for the bulk load feature to work.

NO
disables the bulk load feature. This is the default value.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_CHECKPOINT= Table Option

Specifies when the TPT operation issues a checkpoint or savepoint to the database.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_CHECKPOINT= *number*

Arguments

number
specifies the number of rows after which the TPT operation issues a checkpoint or savepoint to the database. The default is 0, which means no checkpoint is taken. All rows are saved at the end of the job.

TD_DATA_ENCRYPTION= Table Option

Activates data encryption.

Category: Bulk Loading

Alias: TPT_DATA_ENCRYPTION=

Default: NO

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_DATA_ENCRYPTION= YES | NO

Arguments

YES
encrypts all SQL requests, responses, and data.

Alias ON

NO
no encryption occurs. This is the default setting.

Alias OFF

Details

The TD_DATA_ENCRYPTION= session attribute controls encryption for all connections that the Teradata TPT (Teradata Parallel Transporter) creates for LOAD, UPDATE, and STREAM operations.

The TD_DATA_ENCRYPTION= table option overrides the setting of the TPT_DATA_ENCRYPTION= LIBNAME option.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400

TD_DROP_ERROR_TABLE= Table Option

Drops the log table at the end of the job, whether the job was completed successfully or not.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1403

Data source: Teradata

Syntax

TD_DROP_ERROR_TABLE= YES | NO

Arguments

YES
specifies to drop the error tables at the end of the job, whether the job was completed successfully or not.

NO
keeps error tables that contain errors after the job is complete. Error tables that are empty after successful completion are deleted. This is the default setting.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400

TD_DROP_LOG_TABLE= Table Option

Drops the log table at the end of the job, whether the job completed successfully or not.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1403

Data source: Teradata

Syntax

TD_DROP_LOG_TABLE= YES | NO

Arguments

YES
specifies to drop the log table at the end of the job, whether the job is completed successfully or not.

NO
keeps log tables for jobs that were not completed successfully. This is the default setting.

TD_DROP_WORK_TABLE= Table Option

Drops the work table at the end of the job, whether the job was completed successfully or not.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_DROP_WORK_TABLE= YES | NO

Arguments

YES
specifies to drop the work table at the end of the job, whether the job was completed successfully or not.

NO
keeps work tables for jobs that were not completed successfully. This is the default setting.

TD_ERROR_LIMIT= Table Option

Specifies the maximum number of records that can be stored in an error table.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_ERROR_LIMIT= *number-of-records*

Arguments

number-of-records
specifies the maximum number of records that can be stored in an error table before the Load, Stream, or Update operator job is terminated. By default, the ErrorLimit value is unlimited. The number of records must be greater than zero.

Details

The ErrorLimit specification applies to each instance of an operator job. Specifying an invalid value terminates the job.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_ERROR_TABLE_1= Table Option

Specifies a name for the first error table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKLOAD= Table Option" on page 1400
Data source:	Teradata

Syntax

TD_ERROR_TABLE_1= *name-of-table*

Arguments

name-of-table
specifies a name for the error table. The default name for ErrorTable1 is *ttname*_ET, where *ttname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restriction The name of an existing table cannot be used, unless you are restarting a paused job.

Details

ErrorTable1 contains records that were rejected during the acquisition phase of a Load, Stream, or Update operator job because of the following:

- Data conversion errors
- Constraint violations
- Access Module Processor (AMP) configuration changes.

The error table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

Error tables that contain errors are retained at the end of a job. Specify the TD_DROP_ERROR_TABLE= table option if you do not want to retain them.

For information about the error table format and the procedure to correct errors, see *Teradata Parallel Transporter Reference*.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400
- "TD_DROP_ERROR_TABLE= Table Option" on page 1465
- "TD_LOGDB= Table Option" on page 1473

TD_ERROR_TABLE_2= Table Option

Specifies a name for the second error table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	Teradata

Syntax

TD_ERROR_TABLE_2= *name-of-table*

Arguments

name-of-table
specifies a name for the error table. The default name for ErrorTable2 is *ttname*_UV, where *ttname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restrictions
This option is provided for the Load and Update operators. It is ignored by the Stream operator.

The name of an existing table cannot be used, unless you are restarting a paused job.

Details

ErrorTable2 contains records that violated the unique primary index constraint. This type of error occurs during the application phase of a Load or Update operator job.

The error table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

For information about the error table format and the procedure to correct errors, see *Teradata Parallel Transporter Reference*.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400
- “TD_LOGDB= Table Option” on page 1473

TD_LOG_MECH_DATA= Table Option

Specifies additional data for the logon mechanism.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the “BULKOPTS= Table Option” on page 1403

Data source: Teradata

Syntax

TD_LOG_MECH_DATA=*"string"*

Arguments

string
used in conjunction with the TD_LOG_MECH_TYPE= table option, specifies credentials for authentication. Currently, only LDAP credentials are accepted. The credentials must be in the following form:

```
TD_LOG_MECH_DATA="authcid=value  password=value  realm=value"
```

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400
- "TD_LOG_MECH_TYPE= Table Option" on page 1471

TD_LOG_MECH_TYPE= Table Option

Specifies the logon mechanism for a bulk load.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	Teradata

Syntax

TD_LOG_MECH_TYPE= *mechanism*

Arguments

mechanism
specifies the logon authentication mechanism. Currently, the valid value is LDAP.

Requirement The driver requires an 8-character input value. To submit, pad the value with four spaces as follows:

```
TD_LOG_MECH_TYPE="LDAP    "
```

Details

If the user connected via LDAP with the UID connection option, there is no reason to use this option. Use TD_LOG_MECH_TYPE= and TD_LOG_MECH_DATA= to allow LDAP authentication for TPT if the user did not specify LDAP authentication for UID in the connect string. You can also use this option with TD_LOG_MECH_DATA= to override the LDAP ID specified in the UID option with a different LDAP ID.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_LOG_TABLE= Table Option

Specifies the name of the restart log table.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	Teradata

Syntax

TD_LOG_TABLE= *name-of-table*

Arguments

name-of-table

specifies the name of the restart log table for the Load, Stream, and Update operators. The restart log table contains restart information. The default name for the restart log table is *ttname*_RS, where *ttname* is the name of the target table. Table names exceeding 30 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Details

The restart log table is created in the user's default (logon) database, unless the TD_LOGDB= table option is used to specify a different location.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400
- "TD_DROP_LOG_TABLE= Table Option" on page 1466
- "TD_LOGDB= Table Option" on page 1473
- "TD_WORKING_DB= Table Option" on page 1487

TD_LOGDB= Table Option

Specifies the database where the TPT utility tables are created.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the "BULKOPTS= Table Option" on page 1403
Data source:	Teradata

Syntax

TD_LOGDB= *database-name*

Arguments

database-name
specifies the name of the database where the utility tables are to be created. If this option is not specified, the utility tables are created in the specified SCHEMA. If SCHEMA is not specified, the tables are created in the default user (logon) database. The utility tables include ErrorTable1, ErrorTable2, the restart log table, and the work table.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_MAX_SESSIONS= Table Option

Specifies the maximum number of logon sessions that TPT can acquire for a job.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_MAX_SESSIONS= *integer*

Arguments

integer
specifies the maximum number of logon sessions that the Teradata Parallel Transporter can acquire for an operator job. The default value is four sessions, if a value is not specified. The maximum value cannot be more than the number of AMPs available. See your Teradata documentation for details.

Details

The TD_MAX_SESSIONS= value must be greater than zero. Specifying a value less than 1 causes the job to terminate.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_MIN_SESSIONS= Table Option

Specifies the minimum number of sessions for TPT to acquire before a job starts.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_MIN_SESSIONS= *integer*

Arguments

integer
specifies the minimum number of sessions required for TPT to start a job. The number is applied to the Load, Stream, and Update operators. The default is one session.

Details

The TD_MIN_SESSIONS= value must be greater than zero and less than or equal to the maximum number of sessions. Specifying a value less than 1 causes the active operator to terminate.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_NOTIFY_LEVEL= Table Option

Specifies the level at which log events are recorded.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified with the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_NOTIFY_LEVEL= *level*

Arguments

level

Specifies the level at which events are reported. Valid settings are any of the following:

OFF	No notification of events is provided (the default value).
LOW	Initialize, CLIv2/DBS error, Exit events only
MEDIUM	All events except Err1 and Err2
HIGH	All events.

Details

If you set TD_NOTIFY_LEVEL=, set TD_NOTIFY_METHOD= to indicate how you want the events reported.

See Also

Table Options:

- "TD_NOTIFY_METHOD= Table Option" on page 1476

TD_NOTIFY_METHOD= Table Option

Specifies the method for reporting events.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1403
Data source:	Teradata

Syntax

TD_NOTIFY_METHOD= *method*

Arguments

method
specifies the notify method to be used for reporting events. Valid settings are any of the following:

NONE	No notification of events is provided (the default value)
MSG	Sends events to a log. On Windows, the events are sent to an EventLog that can be viewed using the Event Viewer. On UNIX, the destination of events is specified in the /etc/syslog.conf file.
EXIT	Sends events to a user-defined notify exit routine.

Details

If you set TD_NOTIFY_METHOD=, set TD_NOTIFY_LEVEL= to indicate the types of events that you want reported.

See Also

Table Options:

- "TD_NOTIFY_LEVEL= Table Option" on page 1475
- "TD_NOTIFY_STRING= Table Option" on page 1477

TD_NOTIFY_STRING= Table Option

Defines a string that precedes all messages sent to the system log.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified with the "BULKOPTS= Table Option" on page 1403
Data source:	Teradata

Syntax

TD_NOTIFY_STRING= *string*

Arguments

string
Is an optional, user-specified string that precedes all messages that are sent to the system log. This string is also sent to the user-defined notify exit routine. If the NotifyMethod is MSG, the maximum length is 16 bytes. If the NotifyMethod is EXIT, the maximum length is 80 bytes.

See Also

Table Options:

- "TD_NOTIFY_METHOD= Table Option" on page 1476

TD_PACK= Table Option

Specifies the number of statements to pack into a multistatement request.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_PACK= *number*

Arguments

number
specifies the number of statements to pack into a multistatement request. The default number is 20. The maximum number of statements allowed is 600.

Restriction This option is ignored by the Load and Update operators.

See Also

Table Options:

- "TD_PACK_MAXIMUM= Table Option" on page 1479

TD_PACK_MAXIMUM= Table Option

Lets the Stream operator determine the pack factor for the current Stream job.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_PACK_MAXIMUM= YES | NO

Arguments

YES
triggers the Stream operator to dynamically determine the maximum pack factor for the current Stream job.

Restriction This option is ignored by the Load and Update operators.

NO
loads the number of statements indicated by the TD_PACK= option for multistatement requests.

See Also

Table Options:

- "TD_PACK= Table Option" on page 1478

TD_PAUSE_ACQ= Table Option

Forces a pause between the acquisition phase and the application phase of a load job.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified with the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_PAUSE_ACQ= YES | NO

Arguments

YES
specifies to pause the load job after the acquisition phase and skip the application phase.

Restriction This option is provided for the Load and Update operators. It is ignored by the Stream operator.

NO
distributes all of the rows that were sent to the Teradata database during the acquisition phase to their final destination on the AMPs. This is the default value.

TD_SESSION_QUERY_BAND= Table Option

Passes a string of user-specified name=value pairs for use by the TPT session.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified with the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_SESSION_QUERY_BAND= *string*

Arguments

string
specifies one or more query band expressions for use by each TPT session generated for the job or process. The bands are specified as name=value pairs.

Details

The bands are passed to the database for use by the server for load balancing. They are also used as trigger mechanisms by Teradata Multi-System Manager or TASM (Teradata Active System Management).

TD_TENACITY_HOURS= Table Option

Specifies the amount of time the TPT operator continues trying to log on to the Teradata database.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_TENACITY_HOURS= *integer*

Arguments

integer
specifies the number of hours the TPT operator should attempt to log on to the Teradata database, when the maximum number of server utility slots are already occupied by other jobs. The default value is 0 (zero), meaning the operator does not try again and the job fails. It is recommended that you set a value when the Load and Update operators are used.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_TENACITY_SLEEP= Table Option

Specifies the amount of time the TPT operator pauses, before retrying to log on to the Teradata database.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_TENACITY_SLEEP= *integer*

Arguments

integer
specifies the number of minutes that the TPT operator pauses before retrying a job. This option is activated only when TD_TENACITY_HOURS= is set to a value. The default is six minutes.

Details

The TD_TENACITY_SLEEP= value must be greater than zero. If you specify a value of less than 1, the TPT operator responds with an error message and terminates the job.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_TPT_OPER= Table Option

Specifies the load operator that is used by the Teradata Parallel Transporter.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified with the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_TPT_OPER= *operator*

Arguments

operator
Specifies the load operator that the Teradata Parallel Transporter (TPT) uses to load data. Valid values are LOAD, STREAM, or UPDATE.

LOAD	specifies the Load operator, which inserts into an empty table. This is the fastest of the load operators.
STREAM	specifies the Stream operator. It can be used to insert rows into empty Teradata tables or append to existing tables. The Stream operator is used by default when the TD_TPT_OPER= table option is not specified.
UPDATE	specifies the Update operator, which inserts into existing tables. The Update operator is faster than the Stream operator.

Details

When BULKLOAD=YES is specified, the Teradata Parallel Transporter (TPT) is used to load data. TPT jobs are run using operators. These are discrete object-oriented modules that perform specific extraction, loading, and updating processes.

To obtain the best performance when the Update operator is used, it is recommended that you drop all unique secondary indexes, foreign key references, and join indexes onto target tables before the load.

The Teradata server supports a maximum of 16 concurrent utility slots. The Stream operator does not take up a utility slot. The Load and Update operators do take up a slot.

See Also

Table Options:

- “BULKLOAD= Table Option” on page 1400
- “TD_PAUSE_ACQ= Table Option” on page 1479

TD_TRACE_LEVEL=, TD_TRACE_LEVEL_INF= Table Options

Specifies the trace levels for driver tracing. TD_TRACE_LEVEL sets the primary trace level. TD_TRACE_LEVEL_INF sets the secondary trace level.

Category:	Bulk Loading
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1403
Data source:	Teradata

Syntax

TD_TRACE_LEVEL= *trace-level*

Arguments

trace-level

specifies the type of diagnostic messages written by each instance of the driver to an external log file. The diagnostic trace function provides more detailed information (including the version number) in the log file to aid in problem tracking and diagnosis. The following trace levels are available:

TD_OFF	disables driver tracing. TD_OFF is the default setting for both driver tracing and infrastructure tracing. No external log file is produced unless this default is changed. Specifying TD_OFF for both driver tracing and infrastructure tracing is the same as disabling tracing.
TD_OPER	activates the tracing function for driver-specific activities. The absence of any value for the PauseAcq attribute means that the Update driver job executes both the acquisition phase and the application phase without pausing. This distributes all of the rows that were sent to the Teradata database during the acquisition phase to their final destination on the AMPs.
TD_OPER_CLI	activates the tracing function for CLIv2-related activities (interaction with the Teradata database).
TD_OPER_NOTIFY	activates the tracing function for activities related to the Notify feature.
TD_OPER_OPCOMMON	activates the tracing function for activities involving the opcommon library.
TD_OPER_ALL	activates the tracing function for all of the tracing activities.

Details

If the trace level is set to any value other than TD_OFF, an external log file is created for each instance of the driver.

The trace levels for infrastructure tracing should be used only when you are directed to by Teradata support. TD_OFF, which disables infrastructure tracing, should always be specified.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_TRACE_OUTPUT= Table Option

Specifies the name of the external file used for trace messages.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_TRACE_OUTPUT= *filename*

Arguments

filename
specifies the name of the external file to use for tracing. If a file with the specified name already exists, then the existing file is overwritten. The new filename is created with the name of the driver and a time stamp.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_UNICODE_PASSTHRU= Table Option

Allows pass-through characters (PTCs) to be imported into and exported from Teradata.

Category: Bulk Loading

Alias: TPT_UNICODE_PASSTHRU=

Default: NO

Restrictions: This table option is not supported on the CAS server.
This attribute requires version 16.10 of the Teradata database, or later.

Requirement: Must be specified within the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Note: This option is available beginning with SAS 9.4M7.

Syntax

TD_UNICODE_PASSTHRU= YES | NO

Arguments

YES
import and export of PTCs is allowed.

NO
import and export of PTCs is not allowed. This is the default setting.

Details

The TD_UNICODE_PASSTHRU= session attribute controls use of PTCs for all connections that the Teradata TPT (Teradata Parallel Transporter) creates for LOAD, UPDATE, and STREAM operations. PTCs are the set of all possible Unicode characters minus the 6.0 BMP characters that are supported by Teradata and the set of non-characters. For more information, see "Loading Unicode Data with Unicode Pass Through" in Teradata database administration documentation.

The TD_UNICODE_PASSTHRU= table option overrides the setting of the TPT_UNICODE_PASSTHRU= connection option.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

TD_WORK_TABLE= Table Option

Specifies a name for the TPT Work table.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified with the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_WORK_TABLE= *name-of-table*

Arguments

name-of-table

specifies a name for the TPT Work table. The default name for the table is *ttname*_WT, where *ttname* is the name of the target table. Table names exceeding 27 characters are truncated to accommodate the three-character suffix. Therefore, you might want to specify a name for the table that will not be truncated. When truncation occurs, a message is written to the log.

Restriction This option is provided for the Update operator. It is ignored by the Load and Stream operators.

Details

The Work table is created in the default user (logon) database, optionally qualified with a schema, unless the TD_LOGDB= table option is used to specify a different location for utility tables.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400
- "TD_DROP_WORK_TABLE= Table Option" on page 1467
- "TD_LOGDB= Table Option" on page 1473

TD_WORKING_DB= Table Option

Specifies the database where the insert table is to be created.

Category: Bulk Loading

Restriction: This table option is not supported on the CAS server.

Requirement: Must be specified with the "BULKOPTS= Table Option" on page 1403

Data source: Teradata

Syntax

TD_WORKING_DB= *database-name*

Arguments

database-name
specifies a database name. If this option is not specified, the table is created in the specified SCHEMA. If SCHEMA is not specified, the table is created in the default user (logon) database.

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400

THREADNUM= Table Option

Specifies the maximum number of I/O threads the SPD Engine can spawn for processing an SPD Engine data set.

Category: Table Control

Restriction: This table option is not supported on the CAS server.

Data source: SPD Engine data set

Syntax

THREADNUM= *n*

Arguments

n
specifies the number of threads. The default is the value of the SPDEMAXTHREADS= system option, if set. Otherwise, the default is two times the number of CPUs on your computer.

Details

THREADNUM= enables you to specify the maximum number of I/O threads that the SPD Engine spawns for processing an SPD Engine data set. The THREADNUM= value applies to any of the following SPD Engine I/O processing:

- WHERE expression processing
- parallel index creation
- I/O requested by thread-enabled applications

Adjusting THREADNUM= enables the system administrator to adjust the level of CPU resources the SPD Engine can use for any process. For example, in a 64-bit processor system, setting THREADNUM=4 limits the process to, at most, four CPUs, thereby enabling greater throughput for other users or applications.

When THREADNUM= is greater than 1, parallel processing is likely to occur. Therefore, physical order might not be retained in the output. Setting THREADNUM=1 means that no parallel processing occurs

You can also use this option to explore scalability for WHERE expression evaluations.

SPDEMAXTHREADS=, a configurable system option, imposes an upper limit on the consumption of system resources and therefore constrains the THREADNUM= value.

TYPE= Table Option

Specifies the data set type for a specially structured SAS data set.

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set

Syntax

TYPE=*data-set-type*

Required Argument

data-set-type
specifies the special type of the data set.

Details

Use the TYPE= table option in a DS2 program to create a special SAS data set in the proper format, or to identify the special type of the SAS data set in a procedure statement.

You can use the CONTENTS procedure to determine the type of a data set.

Most SAS data sets do not have a specified type. However, there are several specially structured SAS data sets that are used by some SAS/STAT procedures. These SAS data sets contain special variables and observations, and they are usually created by SAS statistical procedures.

Other values are available in other SAS software products and are described in the appropriate documentation.

Note: If you use a DS2 program with a SET statement to modify a special SAS data set, you must specify the TYPE= option in the DATA statement. The *data-set-type* is not automatically copied to the data set that is created.

See Also

Statements:

- "SET Statement" on page 1305

UNIQUESAVE= Table Option

Specifies to save observations with nonunique key values (the rejected observations) to a separate data set when inserting observations into data sets with unique indexes.

Valid in:	INSERT statement
Category:	User Control of SAS Index Usage
Restriction:	This table option is not supported on the CAS server.
Interaction:	Used in conjunction with SPDSUSDS= automatic macro variable
Data source:	SPD Engine data set

Syntax

UNIQUESAVE= YES | NO

Arguments

YES
writes rejected observations to a separate, system-created table that can be accessed by a reference to the macro variable SPDSUSDS=.

NO
does not write rejected observations to a separate table (that is, ignores nonunique key values).

Details

When observations are inserted into a data set that has a unique index, the rejected observations are ignored. With UNIQUESAVE=YES, the rejected observations are saved to a separate data set whose name is stored in the macro variable SPDSUSDS. You can specify the macro variable in place of the data set name to identify the rejected observations.

USER= Table Option

Specifies the HDFS user name.

Category:	Bulk Loading
Alias:	USERID=, UID=
Restriction:	This table option is not supported on the CAS server.
Requirement:	Must be specified within the “BULKOPTS= Table Option” on page 1403.
Interaction:	(Optional) Use with “PASSWORD= Table Option” on page 1444.
Data source:	Impala

Syntax

USER=*HDFS-user*

Arguments

HDFS-user
specifies the name that represents the HDFS user. Quotation marks around the value are optional, unless the name contains spaces.

Example

```
BULKLOAD=YES BULKOPTS=(USER=HADOOP PWD="hdfs-password" CFG="config-
path")
```

See Also

Table Options:

- "BULKLOAD= Table Option" on page 1400
- "PASSWORD= Table Option" on page 1444

WHERENOINDEX= Table Option

Specifies a list of indexes to exclude when making WHERE expression evaluations.

Category: User Control of SAS Index Usage

Restrictions: This table option is not supported on the CAS server.
Cannot be used with IDXWHERE=NO

Data source: SPD Engine data set

Syntax

WHERENOINDEX= (*name1 name2*...)

Arguments

(name1 name2...)
specifies a list of index names that you want to exclude from use.

See Also

Table Options:

- "IDXWHERE= Table Option" on page 1427

WRITE= Table Option

Assigns a WRITE password to a SAS file that prevents users from writing to a file, unless they enter the password.

Restriction: This table option is not supported on the CAS server.

Data source: SAS data set, SPD Engine data set

Note: Check your log after this operation to ensure that the password values are not visible. For more information, see “Blotting Passwords and Encryption Key Values” in *SAS Language Reference: Concepts*.

Syntax

WRITE=*write-password*

Required Argument

write-password
must be a valid SAS name.

Details

The WRITE= option applies to all types of SAS files except catalogs. You can use this option to assign a password to a SAS file or to access a Write-protected SAS file. When the code is written to the SAS log, the password is blotted out. Here is an example:

```
drop thread job2a (write=XXXXXXX);
```

Note: A SAS password does not control access to a SAS file beyond SAS. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside SAS.

See Also

Table Options:

- “ENCRYPT= Table Option” on page 1415
- “READ= Table Option” on page 1450

14

DS2 FCMP Package Methods, Operators, and Statements

Dictionary 1495
DECLARE PACKAGE Statement: FCMP Package 1495
DELETE Method: FCMP Package 1499
NEW Operator: FCMP Package 1500

Dictionary

DECLARE PACKAGE Statement: FCMP Package

Creates a package variable and gives you the option to create an instance of the FCMP package.

Category: Local

Restriction: This statement is not supported on the CAS server.

Requirement: The PACKAGE statement is required before you use the DECLARE PACKAGE statement.

Syntax

DECLARE PACKAGE *fcmp-package-name variable* ( );

Arguments

fcmp-package-name
specifies the name of the FCMP package.

Requirement The package name must match the name of a package created in a PACKAGE statement, or an error will occur.

See "PACKAGE Statement" on page 1286

variable
specifies a name that can reference an instance of the package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use an FCMP package to support calls to functions and subroutines that are available or are created with the FCMP procedure. The FCMP package is predefined for DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

You declare an FCMP package by using the DECLARE PACKAGE statement. This associates an FCMP package with an FCMP name. After you declare the new FCMP package, you can call the functions and subroutines that are created with the FCMP procedure.

There are two ways to construct an instance of an FCMP package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package fcmp pharma;
pharma = _new_ fcmp();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package fcmp pharma();
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

For more information about FCMP packages, see "Using the FCMP Package" in *SAS DS2 Programmer's Guide*.

Examples

Example 1: Using FCMP OUTARGS Parameters

The following example creates FCMP subroutine named **package1** that uses OUTARGS parameters. The FCMP procedure's OUTARGS parameter is treated as an IN_OUT parameter in the METHOD statement.

```
libname base '.';
proc fcmp outlib = base.fcmpsubs.package1;
   subroutine swapper(a,b);
      outargs a,b;
      t1 = b; b = a; a = t1;
   endsub;
run;
quit;

proc ds2;
   package pkg / overwrite=yes language='fcmp' table='base.fcmpsubs';
run;

data _null_;
   dcl package pkg p();
   method init();
      dcl double x y;
      x=10;
      y=42;
      put 'before:' x= y=;
      p.swapper(x,y);
      put 'after:' x= y=;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
before: x=10 y=42
after: x=42 y=10
```

Example 2: FCMP Package Using DOUBLE Arguments

This example walks through creation of a square routine in FCMP and using that routine from a DS2 program. The current directory is used as the "library" of FCMP packages.

```
libname base '.';

* fcmp defines a function, square;
proc fcmp outlib = base.fcmpsubs.package1;
   function square(a);
      return (a*a);
   endsub;
run;
quit;
```

```
* define the ds2 package thru which the fcmp functions will be called;
proc ds2;
   package pkg /overwrite=yes language='fcmp' table='base.fcmpsubs';
run;

* demonstration of calling fcmp thru the ds2 wrapper package;
data _null_;
   dcl package pkg p();
   dcl double a b;
   method init();
      do a = 10 to 20;
         b=p.square(a);
         put a= b=;
      end;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
a=10 b=100
a=11 b=121
a=12 b=144
a=13 b=169
a=14 b=196
a=15 b=225
a=16 b=256
a=17 b=289
a=18 b=324
a=19 b=361
a=20 b=400
```

Example 3: FCMP Package with Character Arguments

```
libname base '.';

proc fcmp outlib = base.fcmpsubs.package1;
   function f(a $) $ 10;
      return (trim(a) !! trim(a));
   endsub;
run;

proc ds2;
   package pkg /overwrite=yes language='fcmp' table='base.fcmpsubs';
run;

data _null_;
   dcl package pkg p();
   method runone(double arg, double expected);
      dcl double actual;
      actual=p.f(arg);
      if (actual ~= expected) then put 'ERROR:' arg= expected= actual=;
   end;
   method init();
      runone(5, 55);
      runone(345, 345345);
      runone(10, 1010);
```

```
      runone(4.2, .);     * can't convert back to double w/ two '.' chars;
      runone(., .);
   end;
enddata;
run;
quit;
```

See Also

- "Using the FCMP Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: FCMP Package" on page 1500

Statements:

- "PACKAGE Statement" on page 1286

DELETE Method: FCMP Package

Deletes an FCMP package.

Restriction: This method is not supported on the CAS server.

Note: The DELETE method is not required. When no FCMP package variables reference a FCMP package instance, the package instance is automatically deleted.

Syntax

package.**DELETE**( );

Required Argument

package
specifies the name of the FCMP package variable.

Details

When you no longer need the FCMP package, delete it by using the DELETE method. If you attempt to use an FCMP package after you delete it, an error will be written to the log.

See Also

"Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

NEW Operator: FCMP Package

Constructs an instance of an FCMP package.

Restriction: This operator is not supported on the CAS server.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable = **_NEW_** *fcmp-package-name*( );

Required Arguments

package-variable
specifies a name that can reference an instance of the package.

fcmp-package-name
specifies the name of the package.

Requirement *fcmp-package-name* must be a predefined FCMP package created with a PACKAGE statement.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You create an FCMP package by using the PACKAGE statement then declare and instantiate the FCMP package by using the DECLARE PACKAGE statement. This associates an FCMP package with an FCMP package variable name. After you declare the new FCMP package, you can call the functions and subroutines that are created with the FCMP procedure.

There are two ways to construct an instance of an FCMP package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package pharma pkg;
pkg = _new_ pharma();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package pharma pkg();
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the FCMP Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: FCMP Package" on page 1495
- "PACKAGE Statement" on page 1286

15

DS2 Hash and Hash Iterator Package Attributes, Methods, Operators, and Statements

Dictionary 1504
ADD Method: Hash Package 1504
CHECK Method 1507
CLEAR Method 1509
DATA Method 1510
DATASET Method 1512
DECLARE PACKAGE Statement: Hash Package 1514
DECLARE PACKAGE Statement: Hash Iterator Package 1523
DEFINEDATA Method 1525
DEFINEDONE Method 1527
DEFINEKEY Method 1528
DELETE Method: Hash, and Hash Iterator Package 1530
DUPLICATE Method 1530
FIND Method 1532
FIND_NEXT Method 1534
FIND_PREV Method 1536
FIRST Method 1538
HASHEXP Method 1541
HAS_NEXT Method 1542
HAS_PREV Method 1544
ITEM_SIZE Attribute 1546
KEYS Method 1547
LAST Method 1548
MULTIDATA Method 1550
NEW Operator: Hash Package 1551
NEW Operator: Hash Iterator Package 1556
NEXT Method 1557
NUM_ITEMS Attribute 1558
ORDERED Method 1560
OUTPUT Method 1562
PREV Method 1564

REF Method ... 1565
REMOVE Method ... 1567
REMOVEALL Method ... 1569
REMOVEDUP Method ... 1571
REPLACE Method ... 1573
REPLACEDUP Method ... 1575
SETCUR Method ... 1578
SUM Method ... 1579
SUMDUP Method ... 1581
SUMINC Method ... 1584

Dictionary

ADD Method: Hash Package

Adds key values, data values, or both to the hash package.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**ADD**();

Form 2: *package*.**ADD**(\[*keys*\], \[*data*\]);

Form 3: *package*.**ADD**(\[*keys*\]);

Required Argument

package
specifies an instance of the hash package variable.

Optional Arguments

[*keys*]
specifies the key values by using a variable list.

Restriction: If you specify only keys, the ADD method works only for key-only hash packages.

See: "Variable Lists" in *SAS DS2 Programmer's Guide*

[*data*]
specifies the variables into which to add the hash data.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

You can store key and data values in the hash package using the ADD method.

There are two ways to pass keys and data values to the ADD method:

- implicit variable method (Form 1)

  The key and data variables are implied in the ADD method invocation and do not have to be specified.

- variable list method (Forms 2 and 3)

  The specified key and data variables are passed explicitly to the ADD method. If the hash package contains only keys, use Form 3.

Note:

- If you add a key that is already in the hash package, then the ADD method returns a nonzero value to indicate that the key is already in the hash package. Use the REPLACE method to replace the data that is associated with the specified key with new data. However, if you set the DUPLICATE constructor parameter or method to `ADD` when you create the hash package, the ADD method returns a zero.
- If you do not specify the data variables with the DEFINEDATA method, the data variables are automatically assumed to be same as the keys.
- The ADD method does not set the value of the data variable to the value of the data item. It sets only the value in the hash package.

Examples

Example 1: Using the Implicit Variable Method

The following example uses the implicit variable method to define the key and data item.

```
data _null_;
   declare char(20) d;
   declare char(20) k;
   declare double rc;
   declare package hash h(4);
   method init();
      /* Define constant value for key and data */
      rc = h.defineKey ('k');
```

```
      rc = h.defineData ('d');
      rc = h.defineDone ();
      /* Define constant value for key and data */
      k='Homer';
      d ='Odyssey';
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add();
      /* Define constant value for key and data */
      k='Joyce';
      d='Ulysses';
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add();
   end;
enddata;
run;
```

Example 2: Adding Key and Data Values Using the Variable List Method

The following example uses the implicit variable method to define the key and data item.

```
data _null_;
   declare char(20) d;
   declare char(20) k;
   declare double rc;
   method init();
      declare package hash h([k], [d]);
      /* Define constant value for key and data */
      k='Homer';
      d ='Odyssey';
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add([k], [d]);
      /* Define constant value for key and data */
      k='Joyce';
      d='Ulysses';
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add([k], [d]);
      h.output('hashdata2');
   end;
enddata;
run;
```

Example 3: Using the ADD and FIND Methods

The following example uses the ADD method to store the data in the hash package and associate the data with the key. The FIND method is then used to retrieve the data that is associated with the key value 'Homer'.

```
data _null_;
   declare char(20) d;
   declare char(20) k;
   declare double rc;
   method init();
      declare package hash h([k], [d], 4);
      /* Define constant value for key and data */
```

```
      k='Homer';
      put k=;
      d='Odyssey';
      put d=;
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add([k], [d]);
      /* Define constant value for key and data */
      k='Joyce';
      d='Ulysses';
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add([k], [d]);
      k='Homer';
      /* Use the FIND method to retrieve the data associated with 'Homer' key */
      if (h.find([k],[d]) = 0) then
     put d=;
     else
     put 'Key Homer not found.';
   end;
enddata;
run;
```

The FIND method assigns the data value 'Odyssey' to the variable D. 'Odyssey' is associated with the key value `Homer'.

The following lines are written to the SAS log:

```
k=Homer
d=Odyssey
d=Odyssey
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "DEFINEDATA Method" on page 1525
- "DEFINEKEY Method" on page 1528
- "REF Method" on page 1565

CHECK Method

Checks whether the specified key is stored in the hash package.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**CHECK**( );

Form 2: *package*.**CHECK**(\[*keys*\]);

Required Argument

package
specifies an instance of the hash package variable.

Optional Argument

[*keys*]
specifies the key values by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

You use the CHECK method to determine whether a key exists in the hash table but the data variable is not updated. The CHECK method returns a zero value if the key is found in the hash table and a nonzero value if the key is not found.

There are two ways to pass keys and data variables to the CHECK method:

- implicit variable method (Form 1)

  The key variables are implied in the CHECK method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified key variables are passed explicitly to the CHECK method.

Comparisons

The CHECK method returns only a value that indicates whether the key is in the hash package. The data variable that is associated with the key is not updated. The FIND method also returns a value that indicates whether the key is in the hash package. However, if the key is in the hash package, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call.

Example

In the following example, the CHECK method is used to determine whether the data is associated with the key value 'Homer'.

```
data _null_;
   declare char(20) d;
   declare char(20) k;
   declare double rc;
   method init();
      declare package hash h([k], [d]);
      /* Define constant value for key and data */
      k='Homer';
      put k=;
      d='Odyssey';
      put d=;
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add([k], [d]);
      /* Define constant value for key and data */
      k='Joyce';
      d='Ulysses';
      /* Use the ADD method to add the key and data to the hash package */
      rc = h.add([k], [d]);
      k='Homer';
      /* Use the CHECK method to verify the data associated with 'Homer' key */
      if (h.check([k]) = 0) then
         put 'Key Homer is found.';
      else
          put 'Key Homer not found.';
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
k=Homer
d=Odyssey
Key Homer is found
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "DEFINEKEY Method" on page 1528
- "FIND Method" on page 1532
- "KEYS Method" on page 1547

CLEAR Method

Removes all items from a hash package without deleting the hash package instance.

Applies to: Hash package

Syntax

package.**CLEAR**( );

Required Argument

package
specifies an instance of the hash package variable.

Details

The CLEAR method removes the items from within the hash package but leaves the hash package instance so that it can be reused. To remove a hash package completely, use the DELETE method.

To clear all items from the hash package MYHASH, use the following code:

```
rc = myhash.clear();
```

See Also

Methods:

- “DELETE Method: Hash, and Hash Iterator Package” on page 1530

DATA Method

Specifies the data variables to be stored in the hash package by using a variable list.

Applies to: Hash package

Notes: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.
All variables that are passed to a hash instance must be global variables.

Syntax

package.**DATA**(\[*data*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*data*]
specifies the name of the data variables by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You use a variable list to specify the data variables for a hash package using the DATA method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

Note: Alternatively, you could use the DEFINEDATA method or constructors in the DECLARE PACKAGE statement to specify the data variables.

Keys and data consist of any number of character or numeric variables.

Example

This example creates a hash package that contains two data variables and one key variable. The output is sorted in descending order.

```
data _null_;
   dcl double x;
   dcl double rc;
   dcl date d;
   /* The output will be in descending order. */
   dcl package hash h(8, '', 'descending', '', '');
   dcl package hiter hi('h');

   method init();
      rc = h.keys([d]);
      rc = h.data([d]);
      rc = h.data([x]);
      rc = h.definedone();
      d = date '1929-08-24'; x = 1; h.add();
      d = date '1930-09-25'; x = 2; h.add();
      d = date '1930-10-26'; x = 3; h.add();
      d = date '1930-10-27'; x = 4; h.add();
      d = date '1933-12-28'; x = 5; h.add();
      d = date '1999-12-01'; x = 999;
      do while (hi.next() = 0);
         put d= x=;
         d = date '1999-12-01'; x = 999;
      end;
   end;
enddata;
```

```
run;
```

The following lines are written to the SAS log.

```
d=1933-12-28 x=5
d=1930-10-27 x=4
d=1930-10-26 x=3
d=1930-09-25 x=2
d=1929-08-24 x=1
```

See Also

- "Defining Key and Data Variables" in *SAS DS2 Programmer's Guide*
- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "DEFINEDATA Method" on page 1525
- "DEFINEDONE Method" on page 1527
- "KEYS Method" on page 1547

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

DATASET Method

Specifies the name of a table to load into the hash package.

Applies to: Hash package

Restriction: This method is not supported on the CAS server

Notes: Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. *sql-text* must be enclosed in braces ({ }).

All variables that are passed to a hash instance must be global variables.

Syntax

package.**DATASET**(['*data-source*'] | '\{*sql-text*\}');

Required Arguments

package
specifies an instance of the hash package variable.

***data-source* | {*sql-text*}**
specifies the name of a table to load into the hash package either as a data source or a FedSQL query..

data-source
specifies the name of a table.

Tip The name of the table can be a string literal or a character variable. If a literal is used, the table name must be enclosed in single quotation marks.

{*sql-text*}
is any valid FedSQL code that resolves to a set of table rows.

Restriction This argument is not supported on the CAS server.

Requirement The FedSQL query must be enclosed in braces ({ }).

Note The FedSQL query is specified in the following form: {`SELECT <select-list> FROM <table-specification>;`}. For more information, see the SELECT statement in the *SAS FedSQL Language Reference*.

Details

You can specify the table to load into the hash package by using the DATASET method.

Note: Alternatively, you can use the *datasource* parameter in the DECLARE PACKAGE statement or the _NEW_ operator to specify the table.

See Also

- "Storing and Retrieving Data" in *SAS DS2 Programmer's Guide*
- "Providing Initialization Data for a Hash Package" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

DECLARE PACKAGE Statement: Hash Package

Creates a package variable and gives you the option of creating an instance of the hash package.

Category: Local

Notes: Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. *sql-text* must be enclosed in braces ({ }).

The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

All variables that are passed to a hash instance must be global variables.

Tip: The PACKAGE statement is not required for a hash package.

Syntax

Form 1: **DECLARE PACKAGE HASH** *variable* ( );

Form 2: **DECLARE PACKAGE HASH** *variable* (*hashexp*, {'*datasource*' | '\{*sql-text*\}'}, '*ordered*', '*duplicate* ', '*suminc*', '*multidata*');

Form 3: **DECLARE PACKAGE HASH** *variable* (\[*keys*\], \[*data*\], [*hashexp*, {'*datasource*' | '\{*sql-text*\}'}, '*ordered*', '*duplicate* ', '*suminc*', '*multidata*']);

Form 4: **DECLARE PACKAGE HASH** *variable* (\[*keys*\], [*hashexp*, {'*datasource*' | '\{*sql-text*\}'}, '*ordered*', '*duplicate* ', '*suminc*', '*multidata*']);

Arguments

variable
specifies a variable that can reference an instance of the hash package.

[*keys*]
specifies the key values by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

[*data*]
specifies the data variables by using a variable list and associates them with the specified keys.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

hashexp
is the hash package's internal table size, where the size of the hash table is 2^n.

The value of hashexp is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for hashexp equates to a hash table size of

2^4, or 16. The maximum value for hashexp is 16, which equates to a hash table size of 2^{16}.

The hash table size is not equal to the number of items that can be stored. Think of the hash table as an array of containers. A hash table size of 16 would have 16 containers. Each container can hold an infinite number of items. The efficiency of the hash tables lies in the ability of the hash function to map items to and retrieve items from the containers.

In order to maximize the efficiency of the hash package lookup routines, you should set the hash table size according to the amount of data in the hash package. Try different hashexp values until you get the best result. For example, if the hash package contains one million items, a hash table size of 16 (hashexp = 4) would not be very efficient. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in better performance.

Range 0–20

Requirement A value for *hashexp* must be entered. If a value less than 0 is entered, then a default value of 8, which equates to a hash table size of 2^8 or 256, is used. If a value greater than 20 is entered, then a default value of 20 is used.

Data type INTEGER

'datasource'
is the name of a table to load into the hash package.

The name of the table can be a literal or a character variable. The table name must be enclosed in single quotation marks.

Restriction If the datasource argument is not null, then the DECLARE PACKAGE statement is not supported on the CAS server.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *datasource* must be entered. If the place holder is entered, then the hash is not loaded from a data source.

{sql-text}
is any valid FedSQL SELECT statement that resolves to a set of table rows.

Restriction This argument is not supported on the CAS server.

Requirement The FedSQL query must be enclosed in braces ({ }).

Note The FedSQL query is specified in the following form: {`SELECT <select-list> FROM <table-specification>;`}. For more information, see the SELECT statement in the *SAS FedSQL Language Reference*.

'ordered'
specifies whether or how the data is returned in key-value order if you use the hash package with a hash iterator package or if you use the hash package OUTPUT method.

ordered can be one of the following values:

'ASCENDING'
Data is returned in ascending key-value order. Specifying `**ASCENDING**' is the same as specifying '**YES**'.

Alias 'A'

'DESCENDING'
Data is returned in descending key-value order.

Alias 'D'

'YES'
Data is returned in ascending key-value order. Specifying '**YES**' is the same as specifying '**ASCENDING**'.

'NO'
Data is returned in an undefined order.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *ordered* must be entered. If the place holder is entered, then a default ordering of 'NO' is used.

'duplicate'
determines whether to ignore duplicate keys when loading a table into the hash package. The default is to store the first key and ignore all subsequent duplicates.

duplicate can be one of the following values:

'REPLACE'
stores the last duplicate key record.

'ERROR'
reports an error to the log if a duplicate key is found.

'ADD'
stores the first key record found and not any of the duplicates.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *duplicate* must be entered. If the place holder is entered, then a default of 'ADD' is used.

See "Example 3: Using the Duplicate Parameter with a Hash Package" on page 1520

'suminc'
specifies a variable that maintains a summary count of hash package keys.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *suminc* must be entered.

Data type INTEGER

Note This variable holds the sum increment–that is, how much to add to the key summary for each reference to the key. The *suminc* value treats a missing or null value as zero, like the SUM function. For

example, a key summary changes using the current value of the variable.

'multidata'
specifies whether multiple data items are allowed for each key.

multidata can be one of the following values:

'YES'
allows multiple data items for each key

Alias 'Y' or 'MULTIDATA'

'NO'
allows only one data items for each key

Alias 'N' or 'SINGLEDATA'

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *multidata* must be entered. If the place holder is entered, then a default of 'NO' is used.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

A particular hash package instance is defined by a set of key variables, a set of data variables, and optional initialization data. A hash package instance can be defined either fully at construction or at construction and through a subsequent series of method calls. If key variables and data variables are specified when the hash package instance is constructed (Forms 3 and 4), then the hash instance is constructed as fully defined. If key variables and data variables are not provided when the hash package instance is constructed (Form 2), then additional definition of the hash instance can be specified with a subsequent series of method calls followed by a single DEFINEDONE method call. The DEFINEDONE method indicates that specification of key variables, data variables, and other initialization data is complete. A hash package instance is not constructed if no variables or initialization data is provided (Form 1). In this instance, the hash instance is constructed with the _NEW_ operator or additional method calls followed by a single DEFINEDONE method call.

In the following example, hash **h1** and hash **h2** have equivalent definition. Hash **h1** is fully defined when the hash is constructed because key and data variables are provided as constructor arguments. Hash **h2** is only partially defined when the hash is constructed, and the hash definition is completed through a series of method calls followed by a single DEFINEDONE method call.

```
declare package hash h1([key1], [data1 data2 data3],
    0, 'testdata', '', '', '', 'multidata');
declare package hash h2();

h2.keys([key1]);
```

```
h2.data([data1 data2 data3]);
h2.dataset('testdata');
h2.multidata();
h2.defineDone();
```

A DECLARE PACKAGE statement can create a hash package variable that is a null package reference. The hash package variable can then be set to reference a hash package instance constructed by a subsequent call of the _NEW_ operator.

```
declare package hash hashgnp;
hashgnp = _new_ hash(10, 'testpkg', 'yes');
```

(Optional) A DECLARE PACKAGE statement can both create the hash package variable and construct the hash package instance.

```
declare package hash hashgnp(10, 'testpkg', 'yes');
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

For more information about the hash package, see "Using the Hash Package" in *SAS DS2 Programmer's Guide*.

Examples

Example 1: Storing and Retrieving Data with a Hash Package

The following example declares a hash package named H that will store several key/value pairs and uses an iterator to write the keys in sorted order.

```
data _null_;
    dcl double x;
    dcl date d;
/* The output will be in descending order. */
    dcl package hash h(8, '', 'descending', '', '');
    dcl package hiter hi('h');
    method init();
      rc = h.defineKey('d');
      rc = h.defineData('d');
      rc = h.defineData('x');
      rc = h.defineDone();

      d = date '1929-08-24'; x = 1; h.add();
      d = date '1930-09-25'; x = 2; h.add();
      d = date '1930-10-26'; x = 3; h.add();
      d = date '1930-10-27'; x = 4; h.add();
      d = date '1933-12-28'; x = 5; h.add();

      d = date '1999-12-01'; x = 999;
      do while (hi.next() = 0);
        put d= x=;
        d = date '1999-12-01'; x = 999;
      end;
    end;
```

```
    enddata;
```

The following lines are written to the SAS log:

```
d=1933-12-28 x=5
d=1930-10-27 x=4
d=1930-10-26 x=3
d=1930-09-25 x=2
d=1929-08-24 x=1
```

Example 2: Loading a Table into a Hash Package

Assume that the table SMALL contains two numeric variables K (key) and S (data) and another table, LARGE, contains a corresponding key variable K. The following code loads the SMALL table into the hash package, and then searches the hash package for key matches on the variable K from the LARGE table.

```
/* create small table */
data small(overwrite=yes);
  dcl char(8) k s;
  method init();
    dcl integer i;
    do i = 1 to 10;
      k = put(i, BEST8.);
      s = put(2*i, BEST8.);
      output;
    end;
  end;
enddata;
run;
/* create large table */
data large(overwrite=yes);
  dcl char(8) k;
  method init();
    dcl integer i;
    do i = -20 to 20;
      k = put(i, BEST8.);
      output;
    end;
  end;
enddata;
run;

/*load SMALL table into the hash package */
data myhash(overwrite=yes);
  declare char(8) k;
  declare char(8) s;
  declare package hash h(8,'small');

 /* define SMALL table variable K as key and S as value */

  method init();
    rc = h.defineKey('k');
    rc = h.defineData('s');
    rc = h.defineDone();
  end;
```

```
/* use the SET statement to iterate over the LARGE table using */
/* keys in the LARGE table to match keys in the hash package */

  method run();
    set large;
    if (h.find() = 0) then output;
  end;

enddata;
run;
```

Example 3: Using the Duplicate Parameter with a Hash Package

Here is an example of using the ADD, REPLACE, and ERROR options with the DUPLICATE parameter.

```
data dups(overwrite=yes);
   dcl double x y;
   method init();
      do x = 1 to 5;
         y = 2*x;
         output;
      end;
      x = 3; y = 99; output;
      x = 4; y = 100; output;
   end;
enddata;
run;

data _null_;
   dcl double x y;
   dcl int rc1 rc2 rc3 rc4;
   dcl package hash h(8, 'dups', 'yes');
   dcl package hiter hi;
   method init();
      rc = h.defineKey('x');
      rc = h.defineData('x');
      rc = h.defineData('y');
      rc = h.defineDone();

      hi = _new_ hiter('h');
        do while (hi.next() = 0);
        put x= y=;
        end;
      put;
   end;
enddata;
run;

data _null_;
   dcl double x y;
   dcl int rc1 rc2 rc3 rc4;
   dcl package hash h(8, 'dups', 'yes', 'add');
   dcl package hiter hi;
   method init();
      rc = h.defineKey('x');
```

```
      rc = h.defineData('x');
      rc = h.defineData('y');
      rc = h.defineDone();

      hi = _new_ hiter('h');
         do while (hi.next() = 0);
           put x= y=;
         end;
      put;
   end;
enddata;
run;

data _null_;
   dcl double x y;
   dcl int rc1 rc2 rc3 rc4;
   dcl package hash h(8, 'dups', 'yes', 'replace');
   dcl package hiter hi;
   method init();
      rc = h.defineKey('x');
      rc = h.defineData('x');
      rc = h.defineData('y');
      rc = h.defineDone();

      hi = _new_ hiter('h');
        do while (hi.next() = 0);
        put x= y=;
        end;
      put;
   end;
enddata;
run;

data _null_;
   dcl double x y;
   dcl int rc1 rc2 rc3 rc4;
   dcl package hash h(8, 'dups', 'yes', 'error');
   dcl package hiter hi;
   method init();
      rc = h.defineKey('x');
      rc = h.defineData('x');
      rc = h.defineData('y');
      rc = h.defineDone();

      hi = _new_ hiter('h');
        do while (hi.next() = 0);
           put x= y=;
        end;
      put;
   end;
enddata;
run;
```

The following lines are written to the SAS log when using the ADD option:

```
x=1 y=2
x=2 y=4
x=3 y=6
x=4 y=8
x=5 y=10

NOTE: Execution succeeded. No rows affected.
```

The following lines are written to the SAS log when using the REPLACE option:

```
x=1 y=2
x=2 y=4
x=3 y=99
x=4 y=100
x=5 y=10

NOTE: Execution succeeded. No rows affected.
```

When using the ERROR option, an error message is written to the SAS log:

```
x=1 y=2
x=2 y=4
x=3 y=6
x=4 y=8
x=5 y=10

NOTE: Execution succeeded. No rows affected.

ERROR: Duplicate key found when loading hash package from data source "dups".
ERROR: Hash data source load failed.
```

Example 4: Using the ORDERED Parameter with a Hash Package

The following example sets an ascending order for the hash package H and a descending order for the hash package H2.

```
data _null_;
    dcl double x y;
    dcl package hash h(8, '', 'ascending');
    dcl package hash h2(8, '', 'descending');
    dcl package hiter hi('h');
    dcl package hiter hi2('h2');
    method init();
      rc = h.defineKey('x');
      rc = h.defineKey('y');
      rc = h.defineDone();

      rc = h2.defineKey('y');
      rc = h2.defineKey('x');
      rc = h2.defineDone();

      do x = 1 to 10;
        y = 2 * x;
        rc = h.add();
        rc = h2.add();
      end;
```

```
      hi.first(); put y=;
      do while (hi.next() = 0);
        put y=;
      end;
      put;

      hi2.first(); put x=;
      do while (hi2.next() = 0);
        put x=;
      end;
    end;
  enddata;
run;
```

See Also

- "Using the Hash Package" in *SAS DS2 Programmer's Guide*
- "Providing Initialization Data for a Hash Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Iterator Package" on page 1523

DECLARE PACKAGE Statement: Hash Iterator Package

Creates a hash iterator package variable and gives you the option of creating an instance of the hash iterator package.

Category: Local

Tip: The PACKAGE statement is not required for a hash iterator package.

Syntax

DECLARE PACKAGE HITER *variable* [(*'hash-name'* | *hash-package-instance*)];

Arguments

variable
specifies a name that can reference an instance of the hash iterator package variable.

'hash-name'
specifies the name of the hash package with which the hash iterator is associated.

hash-package-instance
specifies the instance of the hash package with which the hash iterator is associated.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use the hash and hash iterator packages to quickly and efficiently store, search, and retrieve data based on unique lookup keys. The hash and hash iterator packages are predefined for DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

You declare a hash iterator package by using the DECLARE PACKAGE statement. This associates a hash iterator package with a hash and hash iterator name.

Note: You must declare and instantiate a hash package before you create a hash iterator package.

There are two ways to construct an instance of a hash iterator package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package hiter myhiter;
myhiter = _new_ hiter('h');
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package hiter myiter('h');
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the Hash Iterator Package" in *SAS DS2 Programmer's Guide*
- "Using the Hash Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: Hash Iterator Package" on page 1556

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

DEFINEDATA Method

Defines data variables for the hash package using implicit variables.

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**DEFINEDATA**('*data*');

Required Arguments

package
specifies an instance of the hash package variable.

'data'
specifies the name of the data variable.

Details

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You call the DEFINEDATA method for each data variable that you create. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

Note: Alternatively, you could use the DATA variable list method or constructors in the DECLARE PACKAGE statement to specify the data variables.

Keys and data consist of any number of character or numeric variables.

Example

This example creates a hash package that contains two data variables and one key variable. The output is sorted in descending order.

```
data _null_;
   dcl double x;
   dcl double rc;
   dcl date d;
   /* The output will be in descending order. */
   dcl package hash h(8, '', 'descending', '', '');
   dcl package hiter hi('h');

   method init();
      rc = h.definekey('d');
      rc = h.definedata('d');
      rc = h.definedata('x');
      rc = h.defineDone();
      d = date '1929-08-24'; x = 1; h.add();
      d = date '1930-09-25'; x = 2; h.add();
      d = date '1930-10-26'; x = 3; h.add();
      d = date '1930-10-27'; x = 4; h.add();
      d = date '1933-12-28'; x = 5; h.add();
      d = date '1999-12-01'; x = 999;
      do while (hi.next() = 0);
         put d= x=;
         d = date '1999-12-01'; x = 999;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
d=1933-12-28 x=5
d=1930-10-27 x=4
d=1930-10-26 x=3
d=1930-09-25 x=2
d=1929-08-24 x=1
```

See Also

- "Defining Key and Data Variables" in *SAS DS2 Programmer's Guide*
- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- “DATA Method” on page 1510
- “DEFINEDONE Method” on page 1527

Statements:

- “DECLARE PACKAGE Statement: Hash Package” on page 1514

DEFINEDONE Method

Indicates that all key and data definitions are complete.

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**DEFINEDONE**( );

Required Argument

package
specifies an instance of the hash package variable.

Details

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You can define the key and data variables in one of three ways.

- Use the implicit variable methods DEFINEDATA and DEFINEKEY.
- Use the variable list methods DATA and KEYS.
- Use key and data variable lists as constructors in the DECLARE PACKAGE statement.

If the hash package is not completely defined using constructors in the DECLARE PACKAGE statement, you must call the DEFINEDONE method to complete initialization of the hash package.

Example

The following example creates a hash package, defines the key and data variables, and completes the initialization of the hash package:

```
/* definedone with definedata method */
declare hash h(h);
 rc = h.defineKey('k');
 rc = h.defineData('d');
 rc = h.defineDone();

/* same package using definedone with data method */
declare hash h;
rc=h.keys([k]);
rc=h.data([d]);
rc=definedone();
```

See Also

- "Defining Key and Data Variables" in *SAS DS2 Programmer's Guide*
- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "DEFINEDATA Method" on page 1525
- "DEFINEKEY Method" on page 1528

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

DEFINEKEY Method

Defines key variables for the hash package using implicit variables.

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**DEFINEKEY**('*key*');

Required Arguments

package
specifies an instance of the hash package variable.

'key'
specifies the name of the key variable.

Details

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You call the DEFINEKEY method for each key variable that you create. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

Note: Alternatively, you could use the KEYS variable list method or constructors in the DECLARE PACKAGE statement to specify the key variables.

Keys and data consist of any number of character or numeric variables.

Example

The following example creates a hash package and defines the key and data variables:

```
declare hash h();
rc = h.definekey('k');
rc = h.definedata('d');
rc = h.definedone();
end;
```

See Also

- "Defining Key and Data Variables" in *SAS DS2 Programmer's Guide*
- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "DEFINEDONE Method" on page 1527
- "KEYS Method" on page 1547

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

DELETE Method: Hash, and Hash Iterator Package

Deletes a hash or hash iterator package.

Applies to: Hash and hash iterator packages

Note: The DELETE method is not required. When a hash or hash iterator package variables reference a hash or hash iterator package instance, the package instance is automatically deleted.

Syntax

package.**DELETE**( );

Required Argument

package
specifies an instance of the hash or hash iterator package variable.

Details

When you no longer need the hash or hash iterator package, delete it by using the DELETE method. If you attempt to use a hash or hash iterator package after you delete it, an error will be written to the log.

If you want to delete all the items from within a hash package and save the hash package to use again, use the CLEAR method.

See Also

"Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

DUPLICATE Method

Determines whether to ignore duplicate keys when loading a table into the hash package. The default is to store the first key and ignore all subsequent duplicates.

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**DUPLICATE**('*option*');

Required Arguments

package
specifies an instance of the hash package variable.

'option'
option can be one of the following values:

'REPLACE'
stores the last duplicate key record.

'ERROR'
reports an error to the log if a duplicate key is found.

'ADD'
stores the first key record found and not any of the duplicates.

Default ADD

Details

By default, all of the keys in a hash package are unique. This means one set of data variables exists for each key. In some situations, you might want to have duplicate keys in the hash package, that is, associate more than one set of data variables with a key.

If the table contains duplicate keys, by default, the first instance is stored in the hash package and subsequent instances are ignored. To store the last instance in the hash package, use the DUPLICATE method. The DUPLICATE method also writes an error to the SAS log if there is a duplicate key.

However, the hash package allows storage of multiple values for each key if you use the MULTIDATA parameter or method. The hash package keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS_NEXT or FIND_NEXT.

Note: Alternatively, you can use the *duplicate* parameter as a constructor in the DECLARE PACKAGE statement or the _NEW_ operator to specify duplicate keys.

See Also

- "Non-Unique Key and Data Pairs" in *SAS DS2 Programmer's Guide*
- "Providing Initialization Data for a Hash Package" in *SAS DS2 Programmer's Guide*

Methods:

- "MULTIDATA Method" on page 1550

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

FIND Method

Determines whether the specified key is stored in the hash package.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**FIND**( );

Form 2: *package*.**FIND**(\[*keys*\], \[*data*\]);

Form 3: *package*.**FIND**(\[*keys*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*keys*]
specifies the key values by using a variable list.

Restriction If you specify only keys, the FIND method works only for key-only hash packages.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

[*data*]
specifies the variables into which to copy the hash data.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

You use the key variable values to determine whether a key exists in the hash table. If the key exists, the data values are copied into the data variables. The FIND method returns a zero value if the key is found in the hash table and a nonzero value if the key is not found.

There are two ways to pass key and data variables to the FIND method:

- implicit variable method (Form 1)

  The key and data variables are implied in the FIND method invocation and do not have to be specified.

- variable list method (Forms 2 and 3)

  The specified key and data variables are passed explicitly to the FIND method. If the hash package contains only keys, use Form 3.

Comparisons

The FIND method returns a value that indicates whether the key is in the hash package. If the key is in the hash package, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call. The CHECK method returns only a value that indicates whether the key is in the hash package. The data variable is not updated.

Example

See "Example 3: Using the ADD and FIND Methods" on page 1506.

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*
- "Non-Unique Key and Data Pairs" in *SAS DS2 Programmer's Guide*
- "Storing and Retrieving Data" in *SAS DS2 Programmer's Guide*

Methods:

- "CHECK Method" on page 1507
- "KEYS Method" on page 1547
- "DEFINEKEY Method" on page 1528
- "FIND_NEXT Method" on page 1534
- "FIND_PREV Method" on page 1536

FIND_NEXT Method

Sets the current list item to the next item in the current key's multiple item list and sets the data for the corresponding data variables.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**FIND_NEXT**( );

Form 2: *package*.**FIND_NEXT**(\[*data*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*data*]
specifies the variables into which to copy the data associated with the current key.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The FIND method determines whether the key exists in the hash package.

The HAS_NEXT method determines whether multiple data items are associated with the key. When you have determined that the key has another data item, that data item can be retrieved by using the FIND_NEXT method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_NEXT and FIND_NEXT methods to traverse the list. If there is another item, the method returns a zero. Otherwise, it returns a nonzero value.

There are two ways to pass data variables to the FIND_NEXT method:

- implicit variable method (Form 1)

  The data variables are implied in the FIND_NEXT method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified data variables are passed explicitly to the FIND_NEXT method.

Example

This example uses the FIND_NEXT method to iterate through a table where several keys have multiple data items.

```
data testcases;
   dcl double k;
   dcl double expected;
   method init();
      k=0; expected=14; output; /* magic number */

      k=1; expected=1; output;
      k=2; expected=2; output;
      k=3; expected=1; output;
      k=4; expected=3; output;
      k=5; expected=2; output;
      k=6; expected=1; output;
      k=7; expected=1; output;
      k=8; expected=1; output;
      k=9; expected=1; output;
   end;
enddata;
run;

data inp;
   dcl double k v;
   method init();
      do k = 1 to 10; v = k * k; output; end;
      k = 2; v = 3;     output; /* newval < oldval */
      k = 4; v = 4242;  output; /* newval > oldval */
      k = 4; v = 0;     output; /* newval < oldval */
      k = 5; v = 25;    output; /* newval = oldval */
   end;
enddata;
run;

data _null_;
   dcl double k v;
   dcl package hash h(8, 'inp', 'ascending', '', '', 'multidata');
   method init();
      h.defineKey('k');
      h.defineData('k');
      h.defineData('v');
      h.defineDone();
   end;
   method run();
      dcl double actual;
      /*********************************************/
      set testcases;

      rc = h.find();
      if (rc ~= 0) then
         actual = h.get_num_items();
      else do;
         put k= rc=;
         actual = 0;
```

```
         do while (rc = 0);
            actual+1;
            rc = h.find_next();
            put k= rc=;
         end;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
k=1 rc=0
k=1 rc=4
k=2 rc=0
k=2 rc=0
k=2 rc=4
k=3 rc=0
k=3 rc=4
k=4 rc=0
k=4 rc=0
k=4 rc=0
k=4 rc=4
k=5 rc=0
k=5 rc=0
k=5 rc=4
k=6 rc=0
k=6 rc=4
k=7 rc=0
k=7 rc=4
k=8 rc=0
k=8 rc=4
k=9 rc=0
k=9 rc=4
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*
- "Non-Unique Key and Data Pairs" in *SAS DS2 Programmer's Guide*
- "Storing and Retrieving Data" in *SAS DS2 Programmer's Guide*

Methods:

- "FIND Method" on page 1532
- "FIND_PREV Method" on page 1536
- "HAS_NEXT Method" on page 1542

FIND_PREV Method

Sets the current list item to the previous item in the current key's multiple item list and sets the data for the corresponding data variables.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**FIND_PREV**( );

Form 2: *package*.**FIND_PREV**(\[*data*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*data*]
specifies the variables into which to copy the data associated with the current key.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The FIND method determines whether the key exists in the hash package.

The HAS_PREV method determines whether multiple data items are associated with the key. When you have determined that the key has a previous data item, that data item can be retrieved by using the FIND_PREV method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list. If there is another item, the method returns a zero. Otherwise, it returns a nonzero value.

There are two ways to pass data variables to the FIND_PREVmethod:

- implicit variable method (Form 1)

  The data variables are implied in the FIND_PREV method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified data variables are passed explicitly to the FIND_PREV method.

Example

See "Example: Retrieving a Summary Value" on page 1583.

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*
- "Non-Unique Key and Data Pairs" in *SAS DS2 Programmer's Guide*
- "Storing and Retrieving Data" in *SAS DS2 Programmer's Guide*

Methods:

- "FIND Method" on page 1532
- "FIND_NEXT Method" on page 1534
- "HAS_PREV Method" on page 1544

FIRST Method

Returns the first value in the underlying hash package.

Applies to: Hash iterator package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**FIRST**( );

Form 2: *package*.**FIRST**(\[*data*\]);

Required Arguments

package
specifies an instance of the hash iterator package variable.

[*data*]
specifies the variables into which to copy the data associated with the first hash item.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The FIRST method returns the first data item in the hash package. If you specified **YES** or **ASCENDING** in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'least' key (smallest numeric value or first

alphabetic character). This occurs because the data items are sorted in ascending key-value order in the hash package. Repeated calls to the NEXT method iteratively traverse the hash package and return the data items in ascending key order.

Conversely, if you specified **DESCENDING** in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character). This occurs because the data items are sorted in descending key-value order in the hash package. Repeated calls to the NEXT method iteratively traverse the hash package and return the data items in descending key order.

Use the LAST method to return the last data item in the hash package.

Note: The FIRST method sets the data variable to the value of the data item so that it is available for use after the method call.

There are two ways to pass data variables to the FIRST method:

- implicit variable method (Form 1)

  The data variables are implied in the FIRST method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified data variables are passed explicitly to the FIRST method.

Example

The following example uses the FIRST, NEXT, PREV, and LAST methods when starting a new iteration at a different location within the hash package:

```
data _null_;
   dcl double x y rc;
   dcl package hash h([x], [y]);
   dcl package hiter hi('h');
   method init();
      do x = 1 to 10;
         y = 2*x;
         rc = h.add([x], [y]);
      end;
      do while (hi.next([y]) = 0);
         put y=;
      end;
      put;
      hi.first([y]);
      put y=;
      do while (hi.next([y]) = 0);
         put y=;
      end;
      put;
      do while (hi.prev([y]) = 0);
```

```
            put y=;
         end;
         put;
         hi.last([y]);
         put y=;
         do while (hi.prev([y]) = 0);
            put y=;
         end;
      end;
   enddata;
   run;
```

The following lines are written to the SAS log.

```
y=18
y=10
y=14
y=20
y=2
y=4
y=6
y=8
y=12
y=16

y=18
y=10
y=14
y=20
y=2
y=4
y=6
y=8
y=12
y=16

y=16
y=12
y=8
y=6
y=4
y=2
y=20
y=14
y=10
y=18

y=16
y=12
y=8
y=6
y=4
y=2
y=20
y=14
y=10
y=18
```

See Also

Methods:

- “LAST Method” on page 1548
- “ORDERED Method” on page 1560

Operators:

- “_NEW_ Operator: Hash Package” on page 1551

Statements:

- “DECLARE PACKAGE Statement: Hash Package” on page 1514

HASHEXP Method

Defines the hash package’s internal table size. The size of the hash table is 2^n .

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**HASHEXP**(*exponent*);

Required Arguments

package
specifies an instance of the hash package variable.

exponent
specifies the power-of-2 for the internal table size.

Default 8

Data type INTEGER

Details

The value specified for the HASHEXP method is used as a power-of-two exponent to create the hash table size. For example, a value of 4 equates to a hash table size of 2^4, or 16. The maximum value for *exponent* is 16, which equates to a hash table size of 2^{16}.

The hash table size is not equal to the number of items that can be stored. Think of the hash table as an array of containers. A hash table size of 16 would have 16 containers. Each container can hold an infinite number of items. The efficiency of the hash tables lies in the ability of the hash function to map items to and retrieve items from the containers.

In order to maximize the efficiency of the hash package lookup routines, you should set the hash table size according to the amount of data in the hash package. Try different *exponent* values until you get the best result. For example, if the hash package contains one million items, a hash table size of 16 (hashexp = 4) would not be very efficient. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in better performance.

Note: Alternatively, you can use the *hashexp* parameter in the DECLARE PACKAGE statement or the _NEW_ operator to specify the hash table size.

See Also

- "Providing Initialization Data for a Hash Package" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

HAS_NEXT Method

Determines whether there is a next item in the current key's multiple data item list.

Applies to: Hash package

Syntax

package.**HAS_NEXT**( );

Required Argument

package
specifies an instance of the hash package variable.

Details

If a key has multiple data items, you can use the HAS_NEXT method to determine whether there is a next item in the current key's multiple data item list. If there is another item, the method returns a zero value. Otherwise, it returns a nonzero value.

The FIND method determines whether the key exists in the hash package. The HAS_NEXT method determines whether multiple data items are associated with the key. When you have determined that the key has another data item, that data item can be retrieved by using the FIND_NEXT method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list.

Example: Finding Data Items

This example creates a hash package with one key having multiple data items. It uses the FIND and HAS_NEXT methods to search keys 1, 5, and 11 for the multiple data item.

```
data testdata (overwrite=yes);
   dcl double k v;
   method init();
      k=1; v=1; output;
      k=2; v=4; output;
      k=3; v=9; output;
      k=4; v=16; output;
      k=5; v=25; output;
      k=6; v=36; output;
      k=7; v=49; output;
      k=8; v=64; output;
      k=9; v=81; output;
      k=10; v=100; output;
      k=2; v=3; output;
      k=4; v=4242; output;
      k=4; v=0; output;
      k=5; v=25; output;
   end;
enddata;
run;


data _null_;
   dcl double k v rc;
   dcl package hash h(8, 'testdata', 'ascending', '', '', 'multidata');
   method init();
      h.defineKey('k');
      h.defineData('k');
      h.defineData('v');
      h.defineDone();
   end;
```

```
      method term();
         do k = 1, 5, 11;
            put k=;
            rc = h.find();
            put '..find=' rc;
            /*************************/
            do while (rc=0);
                put '    found:' k= v=;
                rc = h.has_next();
                put '..has_next=' rc;
                if (rc = 0) then do;
                   h.find_next();
                   /* ignore return code */
                   /* already know it from 'has_next', above */
                end;
            end;
            /*************************/
       end;
     end;
   enddata;
   run;
```

The following lines are written to the SAS log.

```
k=1
..find= 0
    found: k=1 v=1
..has_next= 4
k=5
..find= 0
    found: k=5 v=25
..has_next= 0
    found: k=5 v=25
..has_next= 4
k=11
..find= 4
```

See Also

- "Non-Unique Key and Data Pairs" in *SAS DS2 Programmer's Guide*

Methods:

- "FIND Method" on page 1532
- "FIND_NEXT Method" on page 1534
- "HAS_PREV Method" on page 1544

HAS_PREV Method

Determines whether there is a previous item in the current key's multiple data item list.

Applies to: Hash package

Syntax

package.**HAS_PREV**( );

Required Argument

package
specifies an instance of the hash package variable.

Details

If a key has multiple data items, you can use the HAS_PREV method to determine whether there is a previous item in the current key's multiple data item list. If there is a previous item, the method returns a zero. Otherwise, it returns a nonzero value.

The FIND method determines whether the key exists in the hash package. The HAS_NEXT method determines whether multiple data items are associated with the key. When you have determined that the key has a previous data item, that data item can be retrieved by using the FIND_PREV method, which sets the data variable to the value of the data item so that it is available for use after the method call. Once you are in the data item list, you can use the HAS_PREV and FIND_PREV methods in addition to the HAS_NEXT and FIND_NEXT methods to traverse the list.

Example

See "Example: Retrieving a Summary Value" on page 1583.

See Also

- "Non-Unique Key and Data Pairs" in *SAS DS2 Programmer's Guide*

Methods:

- "FIND Method" on page 1532
- "FIND_PREV Method" on page 1536
- "HAS_NEXT Method" on page 1542

ITEM_SIZE Attribute

Returns the size (in bytes) for an item in a hash package.

Applies to: Hash package

Syntax

variable-name=*package*.**ITEM_SIZE**;

Required Arguments

variable-name
specifies the name of the variable that contains the size of the item in the hash package after the method is complete.

package
specifies an instance of the hash package variable.

Details

The ITEM_SIZE attribute returns the size (in bytes) of an item, as well as the key and data variables and some internal information. To set an estimate of how much memory the hash package is using, specify the ITEM_SIZE and NUM_ITEMS attributes. The ITEM_SIZE attribute does not reflect the initial overhead that the hash package requires, nor does it take into account any necessary internal alignments. Therefore, using ITEM_SIZE does not provide exact memory usage, but it does return a good approximation.

Example

For an example, see the "NUM_ITEMS Attribute" on page 1558.

See Also

Attributes:

- "NUM_ITEMS Attribute" on page 1558

KEYS Method

Defines the key variables for the hash package using a variable list.

Applies to: Hash package

Notes: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

All variables that are passed to a hash instance must be global variables.

Syntax

package.**KEYS**(\[*keys*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*keys*]
specifies the names of the key variables using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The hash package uses unique lookup keys to store and retrieve data. The keys and data are variables, which you use to initialize the hash package by using dot notation method calls.

You can use a variable list to specify the key variables for a hash package using the KEYS method. When you have defined all key and data variables, you must call the DEFINEDONE method to complete initialization of the hash package.

Note: Alternatively, you can use the DEFINEKEYS method or constructors in the DECLARE PACKAGE statement to specify key variables.

Note: You can have a hash package that contains only key variables and no data variables. This is a keys-only hash package.

Keys and data consist of any number of character or numeric variables.

Example

The following example creates a hash package and defines the key and data variables:

```
declare hash h();
rc = h.keys([k]);
rc = h.data([d]);
rc = h.definedata();
end;
```

See Also

- "Defining Key and Data Variables" in *SAS DS2 Programmer's Guide*
- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "DEFINEKEY Method" on page 1528
- "DEFINEDONE Method" on page 1527

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

LAST Method

Returns the last value in the underlying hash package.

Applies to: Hash iterator package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**LAST**( );

Form 2: *package*.**LAST**(\[*data*\]);

Required Arguments

package
specifies an instance of the hash iterator package variable.

[*data*]
specifies the variables into which to copy the data associated with the first hash item.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The LAST method returns the last data item in the hash package. If you specified **YES** or **ASCENDING** in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character), because the data items are sorted in ascending key-value order in the hash package.

Conversely, if you specified **DESCENDING** in the DECLARE PACKAGE statement, the _NEW_ operator, or the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character), because the data items are sorted in descending key-value order in the hash package.

Use the FIRST method to return the first data item in the hash package.

There are two ways to pass data variables to the LAST method:

- implicit variable method (Form 1)

  The data variables are implied in the LAST method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified data variables are passed explicitly to the LAST method.

Example

For an example, see the "FIRST Method" on page 1538.

See Also

- "Variable Lists" in *SAS DS2 Programmer's Guide*

Methods:

- "FIRST Method" on page 1538
- "ORDERED Method" on page 1560

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

MULTIDATA Method

Specifies whether multiple data items are allowed for each key.

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**MULTIDATA**(['*option*']);

Required Argument

package
specifies an instance of the hash package variable.

Optional Argument

'option'
option can be one of the following values:

'Y'
allows multiple data items for each key.

Alias 'Y' or 'MULTIDATA'

'N'
reports an error to the log if a duplicate key is found.

Alias 'N' or 'SINGLEDATA'

Default NO

Details

By default, all of the keys in a hash package are unique. This means one set of data variables exists for each key. In some situations, you might want to have duplicate keys in the hash package, that is, associate more than one set of data variables with a key.

If the table contains duplicate keys, by default, the first instance is stored in the hash package and subsequent instances are ignored. To store the last instance in the hash package, use the DUPLICATE method. The DUPLICATE method also writes an error to the SAS log if there is a duplicate key.

However, the hash package allows storage of multiple values for each key if you use the MULTIDATA parameter or method. The hash package keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS_NEXT or FIND_NEXT.

Note: Alternatively, you can use the *multidata* parameter in the DECLARE PACKAGE statement or the _NEW_ operator to specify whether multiple data items are allowed for each key.

See Also

- "Non-Unique Key and Data Pairs" in *SAS DS2 Programmer's Guide*
- "Providing Initialization Data for a Hash Package" in *SAS DS2 Programmer's Guide*

Methods:

- "DUPLICATE Method" on page 1530

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

NEW Operator: Hash Package

Constructs an instance of a hash package.

Notes:

Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. *sql-text* must be enclosed in braces ({ }).

The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

All variables that are passed to a hash instance must be global variables.

Syntax

Form 1: *package-variable*=**_NEW_ HASH**();

Form 2: *package-variable*=**_NEW_ HASH**
(*hashexp*, '*datasource*', '*ordered*', '*duplicate* ', '*suminc*', '*multidata*');

Form 3: *package-variable*=**_NEW_ HASH**
(\[*keys*\], \[*data*\], [*hashexp*, '*datasource*', '*ordered*', '*duplicate* ', '*suminc*', '*multidata*']);

Form 4: *package-variable*=**_NEW_ HASH**
(\[*keys*\], [*hashexp*, {'*datasource*' | \{*sql-text*\} }, '*ordered*', '*duplicate* ', '*suminc*', '*multidata*']);

Required Argument

package-variable
specifies a name that can reference an instance of the package.

Optional Arguments

[*keys*]
specifies the key values by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

[*data*]
specifies the data variables by using a variable list and associates them with the specified keys.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

{sql-text}
is any valid FedSQL SELECT statement that resolves to a set of table rows.

Restriction This argument is not supported on the CAS server.

Requirement The FedSQL query must be enclosed in braces ({ }).

Note The FedSQL query is specified in the following form: {`SELECT <select-list> FROM <table-specification>;`}. For more information, see the SELECT statement in the *SAS FedSQL Language Reference*.

Arguments

hashexp
is the hash package's internal table size, where the size of the hash table is 2^n.

The value of hashexp is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for hashexp equates to a hash table size of 2^4, or 16. The maximum value for hashexp is 16, which equates to a hash table size of 2^{16}.

The hash table size is not equal to the number of items that can be stored. Think of the hash table as an array of containers. A hash table size of 16 would have 16 containers. Each container can hold an infinite number of items. The efficiency of the hash tables lies in the ability of the hash function to map items to and retrieve items from the containers.

In order to maximize the efficiency of the hash package lookup routines, you should set the hash table size according to the amount of data in the hash package. Try different hashexp values until you get the best result. For example, if the hash package contains one million items, a hash table size of 16 (hashexp = 4) would not be very efficient. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in better performance.

Requirement Either a placeholder of –1 or a value for *hashexp* must be entered. If the place holder is entered, then a default value of 8, which equates to a hash table size of 2^8 or 256, is used.

Data type INTEGER

'datasource'
is the name of a table to load into the hash package.

The name of the table can be a literal or a character variable. The table name must be enclosed in single quotation marks.

Restriction If the datasource argument is not null, then the _NEW_ operator is not supported on the CAS server.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *datasource* must be entered. If the place holder is entered, then the hash is not loaded from a data source.

'ordered'
specifies whether or how the data is returned in key-value order if you use the hash package with a hash iterator package or if you use the hash package OUTPUT method. Here are the valid values:

'ASCENDING'
Data is returned in ascending key-value order. Specifying '**ASCENDING**' is the same as specifying '**YES**'.

Alias 'A'

'DESCENDING'
Data is returned in descending key-value order.

Alias 'D'

'YES'
Data is returned in ascending key-value order. Specifying '**YES**' is the same as specifying '**ASCENDING**'.

'NO'
Data is returned in an undefined order.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *ordered* must be entered. If the place holder is entered, then a default ordering of 'NO' is used.

'duplicate'
determines whether to ignore duplicate keys when loading a table into the hash package. The default is to store the first key and ignore all subsequent duplicates. Here are the valid values:

'REPLACE'
stores the last duplicate key record.

'ERROR'
reports an error to the log if a duplicate key is found.

'ADD'
stores the first key record found and not any of the duplicates.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *duplicate* must be entered. If the place holder is entered, then a default of 'ADD' is used.

See "Example 3: Using the Duplicate Parameter with a Hash Package" on page 1520

'suminc'
specifies a variable that maintains a summary count of hash package keys.

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for *suminc* must be entered.

Note This variable holds the sum increment–that is, how much to add to the key summary for each reference to the key. The *suminc* value treats a missing or null value as zero, like the SUM function. For example, a key summary changes using the current value of the variable.

'multidata'
specifies whether multiple data items are allowed for each key. Here are the valid values:

'YES'
allows multiple data items for each key

Alias 'Y' or 'MULTIDATA'

'NO'
allows only one data item for each key

Alias 'N' or 'SINGLEDATA'

Requirement Either a placeholder of an empty string in quotation marks ('') or a value for ordered must be entered. If the place holder is entered, then a default ordering of 'NO' is used.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a hash package variable using the DECLARE PACKAGE statement. After you declare a hash package variable, use the _NEW_ operator to instantiate an instance of the hash package and assign the new package instance to the hash package variable.

```
declare package hash myhash();
myhash = _new_ hash();
```

A particular hash package instance is defined by a set of key variables, a set of data variables, and optional initialization data. A hash package instance can be defined either fully at construction or at construction and through a subsequent series of method calls. If key variables and data variables are specified when the hash package instance is constructed (Forms 3 and 4), then the hash instance is constructed as fully defined. If key variables and data variables are not provided when the hash package instance is constructed (Form 2), then additional definition of the hash instance can be specified with a subsequent series of method calls followed by a single DEFINEDONE method call. The DEFINEDONE method indicates that specification of key variables, data variables, and other initialization data is complete.

For example, you can provide initialization data by using parameters in the constructor syntax for the hash package

```
declare package hash h;
h = _new_ hash(0, 'mytable', 'yes', 'replace', 'sumnum', 'y');
```

Note: You can use the DECLARE PACKAGE statement constructor to declare and instantiate a hash or hash iterator package in one step. For more information, see "Defining a Hash Instance By Using Constructors" in *SAS DS2 Programmer's Guide* and the "DECLARE PACKAGE Statement: Hash Package" on page 1514.

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the Hash Package" in *SAS DS2 Programmer's Guide*
- "Using the Hash Iterator Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: Hash Iterator Package" on page 1556

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

NEW Operator: Hash Iterator Package

Creates an instance of a hash iterator package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable = **_NEW_ HITER** (*'hash-name'*);

Required Arguments

package-variable
specifies a name that can reference an instance of the package.

'hash-name'
specifies the hash package that is associated with the hash iterator package.

Requirement You must declare and instantiate a hash package before you create a hash iterator package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a hash iterator package variable using the DECLARE PACKAGE statement. After you declare a hash iterator package variable, use the _NEW_ operator to instantiate an instance of the hash iterator package and assign the new package instance to the hash iterator package variable.

```
declare package hiter myiter;
myiter = _new_ hiter('myhash');
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a hash iterator package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package hiter myiter('myhash');
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the Hash Iterator Package" in *SAS DS2 Programmer's Guide*
- "Using the Hash Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Iterator Package" on page 1523

NEXT Method

Returns the next value in the underlying hash package.

Applies to: Hash iterator package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**NEXT**( );

Form 2: *package*.**NEXT**(\[*data*\]);

Required Arguments

package
specifies an instance of the hash iterator package variable.

[*data*]
specifies the variables into which to copy the data associated with the next hash item.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

Use the NEXT method iteratively to traverse the hash package and return the data items in key order. The FIRST method returns the first data item in the hash package. You can use the PREV method to return the previous data item in the hash package.

There are two ways to pass data variables to the NEXT method:

- implicit variable method (Form 1)

  The data variables are implied in the NEXT method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified data variables are passed explicitly to the NEXT method.

Example

For an example, see the "FIRST Method" on page 1538.

See Also

Methods:

- "FIRST Method" on page 1538
- "PREV Method" on page 1564

NUM_ITEMS Attribute

Returns the number of items in the hash package.

Applies to: Hash package

Syntax

variable-name=*package*.**NUM_ITEMS**;

Required Arguments

variable-name
specifies the name of a variable that contains the number of items in the hash package after the method is complete.

package
specifies an instance of the hash package variable.

Details

The NUM_ITEMS attribute returns the number of key/data pairs stored in the hash table.

Example

The following example uses the NUM_ITEMS attribute to count the number of items within the hash package and the ITEM_SIZE attribute to report the size of an item in the hash package.

```
data _null_;
   dcl int item_size num_items;
   dcl double x;
   dcl timestamp t;
   dcl package hash h(8, '', 'yes');
   dcl package hiter hi('h');
   method init();
      rc = h.defineKey('t');
      rc = h.defineData('t');
      rc = h.defineData('x');
      rc = h.defineDone();

      item_size = h.item_size;
      num_items = h.num_items;
      put item_size=;
      put num_items=;
      put;

      t = timestamp '1927-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1928-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:37.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1930-09-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1930-10-25 13:52:37.01'; x = 1; h.add();

      num_items = h.num_items;
      put num_items=;

   do while (hi.next() = 0);
      put t= x=;
   end;
   put;
```

```
        t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.remove();
        t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.remove();

        num_items = h.num_items;
        put num_items=;

        do while (hi.next() = 0);
           put t= x=;
        end;
     end;
  enddata;
  run;
```

The following lines are written to the SAS log:

```
item_size=72
num_items=0

num_items=11
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:51:36 x=1
t=1929-09-25 13:52:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1

num_items=9
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
```

See Also

Attributes:

- “ITEM_SIZE Attribute” on page 1546

ORDERED Method

Specifies whether or how the data is returned in key-value order if you use the hash package with a hash iterator package or if you use the hash package OUTPUT method.

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**ORDERED**('*option*');

Required Arguments

package
specifies an instance of the hash package variable.

option
option can be one of the following values:

'ASCENDING'
Data is returned in ascending key-value order. Specifying '**ASCENDING**' is the same as specifying '**YES**'.

Alias 'A'

'DESCENDING'
Data is returned in descending key-value order.

Alias 'D'

'YES'
Data is returned in ascending key-value order. Specifying '**YES**' is the same as specifying '**ASCENDING**'.

'NO'
Data is returned in an undefined order.

Default NO

Details

If you specify **YES** or **ASCENDING** in the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'least' key (smallest numeric value or first alphabetic character). This occurs because the data items are sorted in ascending key-value order in the hash package. Repeated calls to the NEXT method iteratively traverse the hash package and return the data items in ascending key order.

Conversely, if you specify **DESCENDING** parameter in the ORDERED method when you instantiate the hash package, then the data item that is returned is the one with the 'highest' key (largest numeric value or last alphabetic character). This occurs because the data items are sorted in descending key-value order in the hash

package. Repeated calls to the NEXT method iteratively traverse the hash package and return the data items in descending key order.

Use the FIRST method returns the first data item in the hash package. Use the LAST method to return the last data item in the hash package.

Note: Alternatively, you can use the *ordered* parameter in the DECLARE PACKAGE statement or the _NEW_ operator to specify whether the data is returned in key-value order .

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "FIRST Method" on page 1538
- "LAST Method" on page 1548

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

OUTPUT Method

Creates a table that contains the data in the hash package.

Applies to: Hash package

Restriction: This method is not supported on the CAS server.

Syntax

package.**OUTPUT** (['*output-table*']);

Required Arguments

package
specifies an instance of the hash iterator package variable.

['*output-table***']**
specifies the name of the output table.

The name of the hash table can be a literal or character variable.

Tip The name of the table can be a literal or a character variable. If a literal is used, the table name must be enclosed in single quotation marks.

Details

Hash package keys are not automatically stored as part of the output table. The keys must be defined as data items by using the DEFINEDATA method, the DATA method, or the DECLARE PACKAGE statement to be included in the output table.

Example

```
data a(overwrite=yes);
   dcl double x;
      method init();
         do x = 1 to 10;
            output;
         end;
      end;
enddata;
run;
data _null_;
   method init();
      dcl package hash h(4, 'a');
      rc = h.defineData('x');
      rc = h.defineKey('x');
      rc = h.defineDone();
      x = 11;
      h.add();
      x = 12;
      h.add();
      x = 13;
      h.add();
      x = 14;
      h.add();
      rc = h.output('out');
   end;
enddata;
run;
```

See Also

Methods:

- “DATA Method” on page 1510
- “DEFINEDATA Method” on page 1525

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

PREV Method

Returns the previous value in the underlying hash package.

Applies to: Hash iterator package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**PREV**( );

Form 2: *package*.**PREV**(\[*data*\]);

Required Arguments

package
specifies an instance of the hash iterator package variable.

[*data*]
specifies the variables into which to copy the data associated with the previous hash item.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

Use the PREV method iteratively to traverse the hash package and return the data items in reverse key order. The FIRST method returns the first data item in the hash package. The LAST method returns the last data item in the hash package. You can use the NEXT method to return the next data item in the hash package.

There are two ways to pass data variables to the PREV method:

- implicit variable method (Form 1)

  The data variables are implied in the PREV method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified data variables are passed explicitly to the PREV method.

Example

For an example, see the "FIRST Method" on page 1538.

See Also

Methods:

- "FIRST Method" on page 1538
- "LAST Method" on page 1548
- "NEXT Method" on page 1557

REF Method

Consolidates a FIND and ADD methods into a single method call.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**REF**( );

Form 2: *package*.**REF**(\[*keys*\], \[*data*\]);

Form 3: *package*.**REF**(\[*keys*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*keys*]
specifies the key variables by using a variable list.

Restriction If you specify only keys, the FIND method works only for key-only hash packages.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

[*data*]
specifies the variables into which to add the hash data.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

You can consolidate FIND and ADD methods into a single REF method.

There are two ways to pass key and data variables to the REF method:

- implicit variable method (Form 1)

  The key and data variables are implied in the REF method invocation and do not have to be specified.

- variable list method (Forms 2 and 3)

  The specified key and data variables are passed explicitly to the REF method. If the hash package contains only keys, use Form 3.

Example

```
data _null_;
   dcl double x y;
   dcl int rc;
   dcl package hash h([x], [x y]);
   dcl package hiter hi('h');
   method init();
      x = 7; y = 13;
      rc = h.add([x], [x y]);
      put x= rc=;
      do x = 5 to 10;
         y = 2*x;
         rc = h.ref([x], [x y]);
         put x= rc=;
      end;
      do while (hi.next([x y]) = 0);
         put x= y=;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
x=7 rc=0
x=5 rc=0
x=6 rc=0
x=7 rc=0
x=8 rc=0
x=9 rc=0
x=10 rc=0
x=9 y=18
x=5 y=10
x=7 y=13
x=10 y=20
x=6 y=12
x=8 y=16
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*
- "Storing and Retrieving Data" in *SAS DS2 Programmer's Guide*

Methods:

- "ADD Method: Hash Package" on page 1504
- "CHECK Method" on page 1507
- "FIND Method" on page 1532

REMOVE Method

Removes the data that is associated with the specified key from the hash package.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**REMOVE**( );

Form 2: *package*.**REMOVE**(\[*keys*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*keys*]
specifies the key values by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The REMOVE method uses the values in the key variables to find and remove an existing key in a hash table.

You specify the key and then use the REMOVE method to remove the key and data in a hash package.

There are two ways to pass key variables to the REMOVE method:

- implicit variable method (Form 1)

  The key variables are implied in the REMOVE method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified key variables are passed explicitly to the REMOVE method.

Note: The REMOVE method does not modify the value of data variables. It removes only the value in the hash package.

Note: If you specify **YES** the DECLARE PACKAGE statement, the _NEW_ operator, or the MULTIDATA method when you instantiate the hash package, the REMOVE method removes all data items for the specified key.

Example

```
data _null_;
   dcl double x rc;
   dcl timestamp t;
   dcl package hash h(0, '', 'yes');
   dcl package hiter hi('h');
   method init();
      rc = h.Keys([t]);
      rc = h.Data([t x]);
      rc = h.defineDone();
      t = timestamp '1927-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1928-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:37.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1930-09-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1930-10-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      do while (hi.next([t x]) = 0);
         put t= x=;
         t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      end;
      put '************************************';
      t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.remove();
      t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.remove();
      t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      do while (hi.next([t x]) = 0);
         put t= x=;
         t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      end;
   end;
```

```
enddata;
run;
```

The following lines are written to the SAS log.

```
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:51:36 x=1
t=1929-09-25 13:52:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
************************************
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*
- "Replacing and Removing Data" in *SAS DS2 Programmer's Guide*

Methods:

- "ADD Method: Hash Package" on page 1504
- "DEFINEKEY Method" on page 1528
- "KEYS Method" on page 1547

REMOVEALL Method

Removes the data that is associated with all keys from the hash package.

Applies to: Hash package

Syntax

Form 1: *package*.**REMOVEALL**();

Form 2: *package*.**REMOVEALL**(\[*keys*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*keys*]
specifies the key values by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The REMOVEALL method deletes both the keys and the data from the hash package.

There are two ways to pass key variables to the REMOVEALL method:

- implicit variable method (Form 1)

  The key variables are implied in the REMOVEALL method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified key variables are passed explicitly to the REMOVEALL method.

Note: The REMOVEALL method does not modify the value of data variables. It removes only the value in the hash package.

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*
- "Replacing and Removing Data" in *SAS DS2 Programmer's Guide*

Methods:

- "MULTIDATA Method" on page 1550
- "REMOVE Method" on page 1567
- "REMOVEDUP Method" on page 1571

Operators:

- "_NEW_ Operator: Hash Package" on page 1551

Statements:

- "DECLARE PACKAGE Statement: Hash Package" on page 1514

REMOVEDUP Method

Removes the data that is associated with the current key's current data item from the hash package.

Applies to: Hash package

Syntax

package.**REMOVEDUP**( );

Required Argument

package
specifies an instance of the hash package variable.

Details

The REMOVEDUP method deletes the current data item from the hash package for keys that have multiple data items.

Note: The REMOVEDUP method does not modify the value of data variables. It removes only the value in the hash package.

Note: If only one data item is in the key's data item list, the key and data are removed from the hash package.

Comparisons

The REMOVEDUP method removes the data that is associated with the current key's current data item from the hash package. The REMOVE method removes the data that is associated with the specified key from the hash package.

Example

This example creates a hash package where several keys have multiple data items. Duplicate data items in the key are removed.

```
data testdup;
```

```
   length key data 8;
   input key data;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;

proc ds2;
data _null_;
   dcl double key "data" k d;
   method init();
      dcl package hash  h([key], [key "data"], 8, 'testdup', 'yes', '',
         '', 'yes');
      dcl package hiter i(h);
      dcl int rc;

      do k = 1 to 5;
         do while (h.find([k], [k d]) = 0 and h.has_next() = 0);
            h.find_next([k d]);
            h.removedup ();
         end;
      end;

      rc = i.first([k d]);
      do while (rc = 0);
         put k= d=;
         rc = i.next([k d]);
      end;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
key=1 data=10
key=2 data=11
key=3 data=20
key=4 data=6
key=5 data=5
```

See Also

- "Replacing and Removing Data" in *SAS DS2 Programmer's Guide*

Methods:

- "REMOVE Method" on page 1567
- "REMOVEALL Method" on page 1569

REPLACE Method

Replaces the data that is associated with the specified key with new data.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**REPLACE**( );

Form 2: *package*.**REPLACE**(\[*keys*\], \[*data*\]);

Form 3: *package*.**REPLACE**(\[*keys*\]);

Required Arguments

package
specifies an instance of the hash package variable.

[*keys*]
specifies the key values by using a variable list.

Restriction If you specify only keys, the REPLACE method works only for key-only hash packages.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

[*data*]
specifies the variables for which the data is replaced.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The REPLACE method uses the values in the key variables to find a key/data pair in the hash table. If a pair is found, the data is replaced with the current value in the data variables (Forms 1 and 2).

For hash packages that have only keys (Form 3), the only effect that the REPLACE method has is that the summary statistics for the keys is updated.

Example

```
data _null_;
   dcl double x rc;
   dcl timestamp t;
   dcl package hash h(0, '', 'yes');
   dcl package hiter hi('h');
   method init();
      rc = h.defineKey('t');
      rc = h.defineData('t');
      rc = h.defineData('x');
      rc = h.defineDone();
      t = timestamp '1927-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1928-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-08-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-24 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 12:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:51:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:36.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:37.00'; x = 1; h.add();
      t = timestamp '1929-09-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1930-09-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1930-10-25 13:52:37.01'; x = 1; h.add();
      t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      do while (hi.next() = 0);
         put t= x=;
         t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      end;
      put '************************************';
      t = timestamp '1929-09-25 13:51:36.00'; x = 64; h.replace();
      t = timestamp '1929-09-25 13:52:36.00'; x = 64; h.replace();
      t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      do while (hi.next() = 0);
         put t= x=;
         t = timestamp '1999-12-01 12:00:00.00'; x = 999;
      end;
   end;
enddata;
```

The following lines are written to the SAS log.

```
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:51:36 x=1
t=1929-09-25 13:52:36 x=1
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
************************************
t=1927-08-24 12:51:36 x=1
t=1928-08-24 12:51:36 x=1
t=1929-08-24 12:51:36 x=1
t=1929-09-24 12:51:36 x=1
t=1929-09-25 12:51:36 x=1
t=1929-09-25 13:51:36 x=64
t=1929-09-25 13:52:36 x=64
t=1929-09-25 13:52:37 x=1
t=1929-09-25 13:52:37.010000000 x=1
t=1930-09-25 13:52:37.010000000 x=1
t=1930-10-25 13:52:37.010000000 x=1
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*
- "Replacing and Removing Data" in *SAS DS2 Programmer's Guide*

Methods:

- "REPLACEDUP Method" on page 1575

REPLACEDUP Method

Replaces the data that is associated with the current key's current data item with new data.

Applies to: Hash package

Syntax

package.**REPLACEDUP**( );

Required Argument

package
specifies an instance of the hash package variable.

Details

The REPLACEDUP method replaces the current data item from the hash package for keys that have multiple data items.

Note: The REPLACEDUP method does not replace the value of the data variable with the value of the data item. It replaces only the value in the hash package.

Note: If you call the REPLACEDUP method and the key is not found, then the key and data are added to the hash package.

Comparisons

The REPLACEDUP method replaces the data that is associated with the current key's current data item with new data. The REPLACE method replaces the data that is associated with the specified key with new data.

Example

This example creates a hash package where several keys have multiple data items. When a duplicate data item is found, 300 is added to the value of the data item.

```
data testdup;
   length key data 8;
   input key data;
   datalines;
   1 10
   2 11
   1 15
   3 20
   2 16
   2 9
   3 100
   5 5
   1 5
   4 6
   5 99
;
proc ds2;
data _null_;
   dcl double key "data" k d;
   method init();
      dcl package hash h([key], [key "data"], 8, 'testdup','yes', '', '', 'yes');
      dcl package hiter i(h);
      dcl int rc;

      do k = 1 to 5;
```

```
         do while (h.find([k], [k d]) = 0);
            put k= d=;
                  do while (h.has_next() = 0);
               h.find_next([k d]);
               put 'dup ' k= d=;
               d = d + 300;
               h.replacedup ();
                     h.has_next();
            end;
         end;
      end;

      put 'iterating...';
      rc = i.first([k d]);
      do while (rc = 0);
         put k= d=;
         rc = i.next([k d]);
      end;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
key=1 data=10
dup key=1 15
dup key=1 5
key=2 data=11
dup key=2 16
dup key=2 9
key=3 data=20
dup key=3 100
key=4 data=6
key=5 data=5
dup key=5 99
iterating...
key=1 data=10
key=1 data=315
key=1 data=305
key=2 data=11
key=2 data=316
key=2 data=309
key=3 data=20
key=3 data=400
key=4 data=6
key=5 data=5
key=5 data=399
```

See Also

- “Replacing and Removing Data” in *SAS DS2 Programmer’s Guide*

Methods:

- “REPLACE Method” on page 1573

SETCUR Method

Specifies a starting key item for iteration.

Applies to: Hash iterator package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package*.**SETCUR**( );

Form 2: *package*.**SETCUR**(\[*keys*\], \[*data*\]);

Required Arguments

package
specifies the name of the hash iterator package variable.

[*keys*]
specifies the key variables by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

[*data*]
specifies the variables into which to store the data item.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

The hash iterator enables you to start iteration on any item in the hash package. The SETCUR method sets the starting key for iteration. You reference the starting item with the specified key variables. If the item exists, the data associated with the item is stored in the data variables.

You can use the FIRST or LAST methods to start iteration on the first or last item, respectively.

There are two ways to pass key and data variables to the SETCUR method:

- implicit variable method (Form 1)

  The key and data variables are implied in the SETCUR method invocation and do not have to be specified.

- variable list method (Form 2)

The specified key and data variables are passed explicitly to the SETCUR method.

Example

The following example uses the SETCUR method to start iteration at RA= 18 31.6 instead of the first or last items:

```
declare hiter iter('myhash');
myhash.defineKey('ra');
myhash.defineData('obj', 'ra');
myhash.defineDone();
ra='18 31.6';
rc = iter.setcur();
   do while (rc = 0);
      put obj= ra=;
      rc = iter.next();
   end;
```

See Also

- "Variable Lists" in *SAS DS2 Programmer's Guide*

Methods:

- "FIRST Method" on page 1538
- "LAST Method" on page 1548

SUM Method

Retrieves the summary value for a given key from the hash table and stores the value in a variable.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *summary-variable*=*package*.**SUM**();

Form 2: *summary-variable*= *package*.**SUM**(\[*keys*\]);

Required Arguments

summary-variable
specifies a variable that holds the current summary value of the current key.

Note A return code that specifies success or failure is not returned by the method.

package
specifies an instance of the hash package variable.

[*keys*]
specifies the key values by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

You use the SUM method to retrieve key summaries from the hash package. The SUM method retrieves the summary value for a given key when only one data item exists per key. For more information, see "Maintaining Key Summaries" in *SAS DS2 Programmer's Guide*.

There are two ways to pass key variables to the SUM method:

- implicit variable method (Form 1)

  The key variables are implied in the SUM method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified key variables are passed explicitly to the SUM method.

Comparisons

The SUMDUP method retrieves the summary value for the current data item of the current key when more than one data item exists for a key.

Example: Retrieving the Key Summary for a Given Key

The following example uses the SUM method to retrieve the key summary for a given key, 99.

```
data _null_;
   declare double k count total;
   declare package hash myhash(0, '', 'a', '', 'count');
   method init();
```

```
        myhash.defineKey('k');
        myhash.defineDone();

        k = 99;
        count = 1;
        myhash.add();

        /* COUNT is given the value 2.5 and the */
        /* FIND sets the summary to 3.5*/
        count = 2.5;
        myhash.find();

        /* The COUNT of 3 is added to the FIND and */
        /* sets the summary to 6.5. */
        count = 3;
        myhash.find();


        /* The COUNT of -1 sets the summary to 5.5. */
        count = -1;
        myhash.find();

        /* The SUM method gives the current value of */
        /* the key summary to the variable TOTAL. */
        total = myhash.sum();

        /* The PUT statement prints total=5.5 in the log. */
        put total=;

    end;
enddata;
run;
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "SUMDUP Method" on page 1581

SUMDUP Method

Retrieves the summary value for the current data item of the current key and stores the value in a variable.

Applies to: Hash package

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *summary-variable*=*package*.**SUMDUP**();

Form 2: *summary-variable*= *package*.**SUMDUP**(\[*keys*\]);

Required Arguments

summary-variable
specifies a variable that holds the current summary value for the current data item of the current key.

Note A return code that specifies success or failure is not returned by the method.

package
specifies an instance of the hash package variable.

[*keys*]
specifies the key values by using a variable list.

See "Variable Lists" in *SAS DS2 Programmer's Guide*

Details

You use the SUMDUP method to retrieve key summaries from the hash package when a key has multiple data items. For more information, see "Maintaining Key Summaries" in *SAS DS2 Programmer's Guide*.

There are two ways to pass key variables to the SUMDUP method:

- implicit variable method (Form 1)

  The key variables are implied in the SUMDUP method invocation and do not have to be specified.

- variable list method (Form 2)

  The specified key variables are passed explicitly to the SUMDUP method.

Comparisons

The SUMDUP method retrieves the summary value for the current data item of the current key when more than one data item exists for a key. The SUM method retrieves the summary value for a given key when only one data item exists per key.

Example: Retrieving a Summary Value

The following example uses the SUMDUP method to retrieve the summary value for the current data item.

```
data hashinp;
   dcl double k v;
   method init();
      k=1; v=2; output;
      k=1; v=4; output;
      k=1; v=8; output;
      k=2; v=2; output;
      k=3; v=4; output;
      k=2; v=8; output;
    end;
enddata;
run;

data results(keep=(k v sumres));
   dcl double k v si sumres;
   dcl package hash h(8, 'hashinp', 'ascending', '', 'si', 'multidata');

   method testsuminc(double kval, double suminc);
      dcl double rc;
      si = suminc;
      put 'input:' kval= suminc=;
      k=kval; rc=h.find();
      if (rc=0) then do;
         rc=h.has_next();
         do while(rc=0);
            put 'next:' k= v=;
            h.find_next();
            rc=h.has_next();
         end;
         rc=h.has_prev();
          do while(rc=0);
             put 'prev:' k= v=;
             h.find_prev();
             rc=h.has_prev();
          end;
          si=0;
         k=kval; rc=h.find();
          do while(rc=0);
             sumres = h.sumdup();
            output;
             rc=h.find_next();
         end;
      end;
   end;

   method init();
      h.defineKey('k');
      h.defineData('k');
      h.defineData('v');
      h.defineDone();
         testsuminc(1.0, 2.0);
```

```
            testsuminc(2.0, 20.0);
            testsuminc(3.0, 4.0);
      end;
      method term();
         dcl package hiter hi('h');
         rc=hi.first();
         do while (rc=0);
            put 'final:' k= v=;
            rc=hi.next();
         end;
       end;
enddata;
run;

data _null_;
   method run();
      set results;
      put k= v= sumres=;
    end;
enddata;
run;
```

The following lines are written to the SAS log.

```
k=1 v=2 sumres=4
k=1 v=4 sumres=4
k=1 v=8 sumres=2
k=2 v=2 sumres=40
k=2 v=8 sumres=20
k=3 v=4 sumres=4
```

See Also

- "Implicit Variable and Variable List Methods" in *SAS DS2 Programmer's Guide*

Methods:

- "SUM Method" on page 1579

SUMINC Method

Specifies a variable that maintains a summary count of hash package keys.

Applies to: Hash package

Note: All variables that are passed to a hash instance must be global variables.

Syntax

package.**SUMINC**(*suminc-variable*);

Required Arguments

package
specifies an instance of the hash package variable.

suminc-variable
specifies a variable that maintains a summary count of hash package keys.

Details

You can maintain a summary count for a hash package key by using the SUMINC parameter or method. SUMINC instructs the hash package to allocate internal storage in each record to store a summary value in the record each time that the record is used by a FIND, CHECK, or REF method. The SUMINC value is also used to maintain a summary count of hash parameter keys after a FIND, CHECK, or REF method. SUMINC is given a variable, which holds the sum increment, that is, how much to add to the key summary for each reference to the key. The SUMINC value can be greater than, less than, or equal to 0.

The SUMINC value is also used to initialize the summary on an ADD method. Each time the ADD method occurs, the key to the SUMINC value is initialized.

The SUMINC variable treats a missing or null value as zero, like the SUM function. For example, a key summary changes using the current value of the variable.

For more information, see "Maintaining Key Summaries" in *SAS DS2 Programmer's Guide*.

Note: Alternatively, you can use the *suminc* parameter in the DECLARE PACKAGE statement or the _NEW_ operator to retrieve the summary value for the current data item of the current key.

Example

See the example in the "SUMDUP Method" on page 1581.

See Also

- "Providing Initialization Data for a Hash Package" in *SAS DS2 Programmer's Guide*

Methods:

- “CHECK Method” on page 1507
- “FIND Method” on page 1532
- “REF Method” on page 1565

Operators:

- “_NEW_ Operator: Hash Package” on page 1551

Statements:

- “DECLARE PACKAGE Statement: Hash Package” on page 1514

16

DS2 HTTP Package Methods, Operators, and Statements

Method Naming Convention 1588

Dictionary 1588
- ABORT Method 1588
- ADDREQUESTHEADER Method 1589
- ADDSASOAUTHTOKEN Method 1590
- CREATEGETMETHOD Method 1591
- CREATEHEADMETHOD Method 1596
- CREATEPOSTMETHOD Method 1598
- DECLARE PACKAGE Statement: HTTP Package 1599
- DELETE Method: HTTP Package 1600
- EXECUTEMETHOD Method 1601
- EXECUTEMETHODSTREAM Method 1602
- GETRESPONSEBODYASBINARY Method 1603
- GETRESPONSEBODYASSTRING Method 1604
- GETRESPONSECONTENTTYPE Method 1607
- GETRESPONSEHEADERSASSTRING Method 1608
- GETSTATUSCODE Method 1610
- _NEW_ Operator: HTTP Package 1611
- SETOAUTHTOKEN Method 1612
- SETPASSWORD Method 1614
- SETPROXYPASSWORD Method 1615
- SETPROXYURL Method 1616
- SETPROXYUSERNAME Method 1617
- SETREQUESTBODYASBINARY Method 1618
- SETREQUESTBODYASSTRING Method 1619
- SETREQUESTBODYCHARACTERSET Method 1620
- SETREQUESTCONTENTTYPE Method 1621
- SETRESPONSEBODYCHARACTERSET Method 1622
- SETSOCKETTIMEOUT Method 1623
- SETURL Method 1624
- SETUSERNAME Method 1626
- STREAMRESPONSEBODYASBINARY Method 1627
- STREAMRESPONSEBODYASSTRING Method 1628

Method Naming Convention

When discussing a similar set of methods, this document uses a short name and an asterisk (*) to designate the set of methods as a whole.

For example, when the discussion involves both the SETREQUESTBODYASBINARY and SETREQUESTBODYASSTRING methods, the documentation reads “the SETREQUESTBODY* methods”.

Dictionary

ABORT Method

Stops the execution of the HTTP method.

Syntax

package.**ABORT**( );

Required Argument

package
specifies an instance of the HTTP package variable.

Details

Call the ABORT method to stop the HTTP method that is currently running. The ABORT method is useful when you are streaming data and decide that you do not need to see the entire response body.

See Also

- “Using the HTTP Package” in *SAS DS2 Programmer’s Guide*

Methods:

- "EXECUTEMETHOD Method" on page 1601
- "EXECUTEMETHODSTREAM Method" on page 1602

ADDREQUESTHEADER Method

Adds a header to the HTTP method request.

Syntax

package.**ADDREQUESTHEADER**(*'name'*, *'value'*);

Required Arguments

package
specifies an instance of the HTTP package variable.

'name'
specifies the name of the header field.

'value'
specifies the value of the header field.

Details

The header is appended to the end of the list of headers.

Example

```
h.addRequestHeader('Cookie', ' SSID=3lk3q84095jk79lgjf');
```

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEGETMETHOD Method" on page 1591
- "CREATEHEADMETHOD Method" on page 1596
- "CREATEPOSTMETHOD Method" on page 1598

ADDSASOAUTHTOKEN Method

Searches the SAS environment for an OAuth access token. If a token is found, it is set as the OAuth token for the request.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**ADDSASOAUTHTOKEN**( );

Required Argument

package
specifies an instance of the HTTP package variable.

Details

Open Authorization (OAuth) is an open standard for token-based authentication and authorization on the internet. OAuth is designed to work with HTTP and allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The third-party client then uses the access token to access the protected resources that are hosted by the resource server.

For the HTTP package, use the ADDSASOAUTHTOKEN method to allow SAS to search the environment for an OAuth access token that will be added to the request header as a Bearer value of an Authorization header field.

For more information about OAuth, see OAuth 2.0.

Note: A return code value of 0 indicates success, a value of 1 indicates failure, a value of 3 indicates a token is not found, and a value of 4 indicates that the token is expired.

Comparisons

You allow SAS to search the environment for an OAuth access token with the ADDSASOAUTHTOKEN method. You provide the OAuth access token with the SETOAUTHTOKEN method.

See Also

Methods:

- "SETOAUTHTOKEN Method" on page 1612

CREATEGETMETHOD Method

Creates an HTTP GET method to retrieve a resource from a web server.

Syntax

package.**CREATEGETMETHOD** ( ) | (*'url'*);

Required Arguments

package
specifies an instance of the HTTP package variable.

( ) | (*'url'*)
either generates the HTTP GET method without a URL or specifies the URL of the resource.

Restriction The version of the CREATEGETMETHOD method that supports being called without arguments, CREATEGETMETHOD( ), is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Tip The URL can include query strings (name/value pairs).

Details

Use the CREATEGETMETHOD method to create an HTTP GET method. After you create the GET method, call one of the EXECUTEMETHOD* methods to request the specified resource from the web server.

The URL must be set either as an argument in the CREATEGETMETHOD method or by using the SETURL method. Here is an example:

```
h.createGetMethod('http://httpbin.org/get');
```

or

```
h.createGetMethod();
h.setUrl('http://httpbin.org/get');
```

Examples

Example 1: Create a GET Method and Retrieve an HTTP Resource

The following example requests an HTTP resource and displays the headers and body of the response from the web server.

```
data _null_;
  method init();
    declare package http h();
    declare varchar(1024) character set utf8 body headers;
    declare int rc status;

    /* Build and send a GET method for the 'FR' resource */
    h.createGetMethod('http://api.worldbank.org/countries/FR');
    h.executeMethod();

    /* If the resource was returned by the server, show the response */
    status = h.getStatusCode();   /* get the HTTP status code */
    put 'Country code: FR, executeMethod() status:' status;
    if status eq 200 then do;     /* 200 = OK */
      /* retrieve the headers from the response that came from the server */
      h.getResponseHeadersAsString(headers, rc);
      put 'Headers:';
      put headers;
      /* retrieve the body from the response that came from the server */
      h.getResponseBodyAsString(body, rc);
      put 'Body:';
      put body;
    end;
  end;
enddata; run;
```

The following lines are written to the log:

```
Country code: FR, executeMethod() status: 200
Headers:
HTTP/1.1 200 OK
Content-Length: 627
Content-Type: text/xml; charset=UTF-8
Server: WorldBank
Web Server2
Date: Sat, 22 Mar 2014 00:43:30 GMT
X-Cache: MISS from transproxy
Via: 1.1
transproxy (squid)
Connection: keep-alive

Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:countries page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:country id="FRA">
<wb:iso2Code>FR</wb:iso2Code>
    <wb:name>France</wb:name>
    <wb:region id="ECS">Europe
&amp; Central Asia (all income levels)</wb:region>
    <wb:adminregion id="" />
<wb:incomeLevel id="OEC">High income: OECD</wb:incomeLevel>
    <wb:lendingType id="LNX">Not
classified</wb:lendingType>
    <wb:capitalCity>Paris</wb:capitalCity>
<wb:longitude>2.35097</wb:longitude>
    <wb:latitude>48.8566</wb:latitude>
</wb:country>
</wb:countries>
```

Example 2: Create GET Methods Using Expressions in the Resource URL

The following example generates a separate GET request for each country code in the input data set. Each code is concatenated into the expression that forms the URL of the resource. Note that the final code, ZZ, is not a valid country code.

```
proc ds2;
data country_codes /overwrite=yes;
dcl char(2) code;
method init();
  code='ES'; output;
  code='FR'; output;
  code='GB'; output;
  code='ZZ'; output; /* unknown country code */
end;
enddata; run; quit;

proc ds2;
data _null_;
  method run();
    declare package http h();
    declare varchar(1024) character set utf8 body;
    declare int rc status;

    /* Build and send a GET method for each country code */
    set country_codes;
    h.createGetMethod('http://api.worldbank.org/countries/'
        || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1990');
```

```
    h.executeMethod();

    /* If the resource was returned by the server, show the response */
    status = h.getStatusCode();     /* get the HTTP status code */
    put 'Country code:' code 'executeMethod() status code:' status;
    if status eq 200 then do;       /* 200 = OK */
      /* retrieve the body from the response that came from the server */
      h.getResponseBodyAsString(body, rc);
      put 'Body:';
      put body;
    end;
    put;
  end;
enddata; run; quit;
```

The following lines are written to the log:

```
Country code: ES executeMethod() status code: 200
Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per
capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="ES">Spain</wb:country>
  <wb:date>1990</wb:date>
    <wb:value>11880</wb:value>
    <wb:decimal>0</wb:decimal>
</wb:data>
</wb:data>

Country code= FR executeMethod() status code = 200
Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per
capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="FR">France</wb:country>
   <wb:date>1990</wb:date>
    <wb:value>20050</wb:value>
    <wb:decimal>0</wb:decimal>
</wb:data>
</wb:data>

Country code= GB executeMethod() status code = 200
Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="1"
xmlns:wb="http://www.worldbank.org">
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per
capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="GB">United
Kingdom</wb:country>
    <wb:date>1990</wb:date>
    <wb:value>16620</wb:value>
<wb:decimal>0</wb:decimal>
  </wb:data>
</wb:data>

Country code= ZZ executeMethod() status code = 200
Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:error xmlns:wb="http://www.worldbank.org">
<wb:message id="120" key="Parameter 'country' has an
invalid value">The provided parameter value
is not valid</wb:message>
</wb:error>
```

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "EXECUTEMETHOD Method" on page 1601

- "EXECUTEMETHODSTREAM Method" on page 1602
- "SETURL Method" on page 1624

CREATEHEADMETHOD Method

Creates an HTTP HEAD method to test whether a web resource exists and to retrieve its headers.

Syntax

package.**CREATEHEADMETHOD**( ) | (*'url'*);

Required Arguments

package
specifies an instance of the HTTP package variable.

( ) | (*'url'*)
either generates the HTTP HEAD method without a URL or specifies the URL of the resource.

Restriction The version of the CREATEHEADMETHOD method that supports being called without arguments, CREATEHEADMETHOD( ), is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Tip The URL can include query strings (name/value pairs).

Details

Use the CREATEHEADMETHOD method to create an HTTP HEAD method. After you create the HEAD method, call the EXECUTEMETHOD method to send the request to the web server.

The URL must be set either as an argument in the CREATEHEADMETHOD method or by using the SETURL method. Here is an example:

```
h.createHeadMethod('http://httpbin.org/head');
```

or

```
h.createHeadMethod();
h.setUrl('http://httpbin.org/head');
```

Example

The following example creates a HEAD method, sends the request to the web server, and displays the headers of the response.

```
data _null_;
  method init();
    declare package http h();
    declare varchar(1024) character set utf8 headers;
    declare int rc status;
    declare char(2) code;

    /* Build and send a HEAD method for a resource */
    code = 'GB';
    h.createHeadMethod('http://api.worldbank.org/countries/'
      || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1990');
    h.executeMethod();

    /* If the resource headers were returned by the server, show them */
    status = h.getStatusCode();   /* get the HTTP status code */
    put 'Country code:' code 'executeMethod() status:' status;
    if status eq 200 then do;     /* 200 = OK */
      /* retrieve the headers from the response that came from the server */
      h.getResponseHeadersAsString(headers, rc);
      put 'Headers:';
      put headers;
    end;
  end;
enddata; run;
```

The following lines are written to the log:

```
Country code: GB executeMethod() status: 200
Headers:
HTTP/1.1 200 OK
Content-Length: 0
Content-Type: text/xml; charset=UTF-8
Server: WorldBank Web
Server1
Date: Tue, 25 Mar 2014 21:05:48 GMT
X-Cache: MISS from transproxy
Via: 1.1 transproxy
(squid)
Connection: keep-alive
```

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "EXECUTEMETHOD Method" on page 1601
- "SETURL Method" on page 1624

CREATEPOSTMETHOD Method

Creates an HTTP POST method to request the web server to accept data for a resource.

Syntax

package.**CREATEPOSTMETHOD** ( ) | (*'url'*);

Required Arguments

package
specifies an instance of the HTTP package variable.

( ) | (*'url'*)
either generates the HTTP POST method without a URL or specifies the URL of the resource.

Restriction The version of the CREATEPOSTMETHOD method that supports being called without arguments, CREATEPOSTMETHOD( ), is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Tip The URL can include query strings (name/value pairs).

Details

Use the CREATEPOSTMETHOD method to create an HTTP POST method.

The URL must be set either as an argument in the CREATEPOSTMETHOD method or by using the SETURL method. Here is an example:

```
h.createPostMethod('http://httpbin.org/post');
```

or

```
h.createPostMethod();
h.setUrl('http://httpbin.org/post');
```

Because the POST method requires a body, complete the POST method by doing the following:

- Add the body content by calling one of the SETREQUESTBODY* methods, depending on the content type.
- Indicate the content type of the body by calling the SETREQUESTCONTENTTYPE method, which sets the `Content-Type:` header.

Then, call the EXECUTEMETHOD method to send the request to the web server.

Note: The content length is computed by the DS2 HTTP client after transcoding the data.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "ADDREQUESTHEADER Method" on page 1589
- "EXECUTEMETHOD Method" on page 1601
- "SETREQUESTBODYASBINARY Method" on page 1618
- "SETREQUESTBODYASSTRING Method" on page 1619
- "SETREQUESTCONTENTTYPE Method" on page 1621
- "SETURL Method" on page 1624

DECLARE PACKAGE Statement: HTTP Package

Creates a package variable and enables you to create an instance of the HTTP package.

Category: Local

Tip: The PACKAGE statement is not required for an HTTP package.

Syntax

DECLARE PACKAGE HTTP *variable*( );

Arguments

variable
specifies a variable that can reference an instance of the HTTP package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use an HTTP package to construct an HTTP client to access HTTP web servers. The HTTP package is predefined for DS2 programs.

You declare an HTTP package by using the DECLARE PACKAGE statement. When a package is declared, a variable is created that can reference an instance of the package. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of an HTTP package:

- Use the DECLARE PACKAGE statement along with the _NEW_ operator:

```
declare package http httpclt;
httpclt = _new_ http();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package http httpclt();
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: HTTP Package" on page 1611

DELETE Method: HTTP Package

Deletes an HTTP package instance and frees its resources.

Note: The DELETE method is not required. When no HTTP package variables reference an HTTP package instance, the package instance is automatically deleted.

Syntax

package.**DELETE**( );

Required Argument

package
specifies the name of the HTTP package variable.

Details

When you no longer need the HTTP package, delete it by using the DELETE method. If you attempt to use an HTTP package instance after you delete it, an error is written to the log.

See Also

"Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

EXECUTEMETHOD Method

Executes the HTTP method and enables retrieval of the response body as a complete entity.

Syntax

package.**EXECUTEMETHOD**( );

Required Argument

package
specifies an instance of the HTTP package variable.

Details

You must create a GET, HEAD, or POST method before you call the EXECUTEMETHOD method to send the request to the web server. The EXECUTEMETHOD method sends the last method that was created.

If the response has a body, the response body must be retrieved as an entire entity by using one of the GETRESPONSEBODYAS* methods.

Note: The EXECUTEMETHOD method does not support streaming of the response body. If you use one of the STREAMRESPONSEBODYAS* methods to retrieve the response body after sending the request with the EXECUTEMETHOD method, a run-time error occurs. Use the EXECUTEMETHODSTREAM method instead.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "EXECUTEMETHODSTREAM Method" on page 1602
- "GETSTATUSCODE Method" on page 1610
- "GETRESPONSECONTENTTYPE Method" on page 1607
- "GETRESPONSEBODYASBINARY Method" on page 1603
- "GETRESPONSEBODYASSTRING Method" on page 1604
- "SETSOCKETTIMEOUT Method" on page 1623

EXECUTEMETHODSTREAM Method

Executes the HTTP method and enables streaming of the response body from the HTTP server.

Syntax

package.**EXECUTEMETHODSTREAM**( );

Required Argument

package
specifies an instance of the HTTP package variable.

Details

Use the EXECUTEMETHODSTREAM method when you want to stream the response body from the web server in chunks, using either the STREAMRESPONSEBODYASBINARY or STREAMRESPONSEBODYASSTRING method. This is useful when you want to process the response without waiting for all of the data to arrive from the server.

You must create a GET method before you call the EXECUTEMETHODSTREAM method to send the request to the web server. The EXECUTEMETHODSTREAM method sends the last method that was created.

Note: The EXECUTEMETHODSTREAM method does not support retrieval of the response body as an entire entity. If you use one of the GETRESPONSEBODYAS* methods to retrieve the response body after sending the request with the

EXECUTEMETHODSTREAM method, a run-time error occurs. Use the EXECUTEMETHOD method instead.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "EXECUTEMETHOD Method" on page 1601
- "GETSTATUSCODE Method" on page 1610
- "GETRESPONSECONTENTTYPE Method" on page 1607
- "SETSOCKETTIMEOUT Method" on page 1623
- "STREAMRESPONSEBODYASBINARY Method" on page 1627
- "STREAMRESPONSEBODYASSTRING Method" on page 1628

GETRESPONSEBODYASBINARY Method

Returns the entire body from the HTTP response in binary format.

Restriction: If you use the GETRESPONSEBODYASBINARY method to return the entire body from the HTTP response in binary format, you cannot use the SETRESPONSEBODYCHARACTERSET method.

Syntax

package.**GETRESPONSEBODYASBINARY**(*variable*, *rc*);

Required Arguments

package
specifies an instance of the HTTP package variable.

variable
specifies the binary variable to hold the entire response body.

rc
specifies the variable to hold the return code value.

Note: A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the GETRESPONSEBODYASBINARY method to retrieve the response body in binary format. The response body is returned as one entity.

You can call the GETRESPONSEBODYASBINARY method after using the EXECUTEMETHOD method to retrieve the response body returned by the web server.

Note: The GETREAPONSEBODYASBINARY method does not return until the web client has received all the data from the web server.

Note: The response body is not transcoded.

Note: The EXECUTEMETHOD method does not support streaming of the response body.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEGETMETHOD Method" on page 1591
- "EXECUTEMETHOD Method" on page 1601
- "GETRESPONSEBODYASSTRING Method" on page 1604
- "SETRESPONSEBODYCHARACTERSET Method" on page 1622

GETRESPONSEBODYASSTRING Method

Returns the entire body from the HTTP response in character string format.

Requirement: A character set for the response body must not be set after getting the response body. A character set for the response body must be specified with the SETRESPONSEBODYCHARACTERSET method before getting the response body with the GETRESPONSEBODYASSTRING method.

Syntax

package.**GETRESPONSEBODYASSTRING**(*variable*, *rc*);

Required Arguments

package
specifies an instance of the HTTP package variable.

variable
specifies the variable to hold the entire response body.

rc
specifies the variable to hold the return code value.

Note: A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the GETRESPONSEBODYASSTRING method to retrieve the response body in character string format. The response body is returned as one entity.

If the content type is not set, the HTTP client deduces a character set from the response content type. If the HTTP client is unable to deduce a character set from the response content type, the character set defaults to UTF-8. However, you can use the SETRESPONSEBODYCHARACTERSET method to specify the character set to use when retrieving the response body.

You can call the GETRESPONSEBODYASSTRING method after using the EXECUTEMETHOD method to retrieve the response body returned by the web server.

Note: The GETREAPONSEBODYASSTRING method does not return until the web client has received all the data from the web server.

Note: The response body is transcoded to the encoding of the string if the encodings are different.

Note: The EXECUTEMETHOD method does not support streaming of the response body.

Example

```
data _null_;
  method init();
    declare package http h();
    declare varchar(1024) character set utf8 body;
    declare int rc status;
    declare char(2) code;
```

```
    /* Build and send a GET method for a resource */
    code = 'FR';
    h.createGetMethod('http://api.worldbank.org/countries/'
      || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1991');
    h.executeMethod();

    /* If the resource was returned by the server, show the response */
    status = h.getStatusCode();   /* get the HTTP status code */
    put 'Requested resource for country code:' code 'executeMethod() status:' status;
    if status eq 200 then do;     /* 200 = OK */
      /* retrieve the body from the response that came from the server */
      h.getResponseBodyAsString(body, rc);
      put 'Body:';
      put body;
    end;
  end;
enddata; run;
```

The following lines are written to the log.

```
Requested resource for country code: FR executeMethod() status: 200
Body:
<?xml version="1.0" encoding="utf-8"?>
<wb:data page="1" pages="1" per_page="50" total="2"
xmlns:wb="http://www.worldbank.org">
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per
capita, Atlas method (current US$)</wb:indicator>
    <wb:country id="FR">France</wb:country>
   <wb:date>1991</wb:date>
    <wb:value>20880</wb:value>
    <wb:decimal>0</wb:decimal>
</wb:data>
  <wb:data>
    <wb:indicator id="NY.GNP.PCAP.CD">GNI per capita, Atlas method
(current US$)</wb:indicator>
    <wb:country id="FR">France</wb:country>
<wb:date>1990</wb:date>
    <wb:value>20050</wb:value>
    <wb:decimal>0</wb:decimal>
</wb:data>
</wb:data>
```

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEGETMETHOD Method" on page 1591
- "EXECUTEMETHOD Method" on page 1601
- "GETRESPONSEBODYASBINARY Method" on page 1603
- "SETRESPONSEBODYCHARACTERSET Method" on page 1622

GETRESPONSECONTENTTYPE Method

Returns the content type from the HTTP response.

Syntax

content-type-variable=*package*.**GETRESPONSECONTENTTYPE**( );

Required Arguments

content-type-variable
specifies the variable to hold the content type value of the HTTP response.

Note: Consult an HTTP reference for a list of possible content types.

package
specifies an instance of the HTTP package.

Details

Use the GETRESPONSECONTENTTYPE method to retrieve the content type from the latest response message.

Example

```
data _null_;
  method init();
    declare package http h();
    declare varchar(1024) character set utf8 body contentType headers;
    declare int rc status;

    /* Build and send a HEAD method for the 'ES' resource */
    h.createHeadMethod('http://api.worldbank.org/countries/ES');
    h.executeMethod();

    /* If the resource was returned by the server, show the response */
    status = h.getStatusCode();   /* get the HTTP status code */
    put 'Country code: ES, executeMethod() status:' status;
    if status eq 200 then do;     /* 200 = OK */
      /* retrieve the content type from the response that came from the server */
      contentType = h.getResponseContentType();
      put 'Content type:' contentType;
```

```
    end;
  end;
enddata; run;
```

The following lines are written to the log.

```
Country code: ES, executeMethod() status: 200
Content type: text/xml; charset=UTF-8
```

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "EXECUTEMETHOD Method" on page 1601
- "EXECUTEMETHODSTREAM Method" on page 1602

GETRESPONSEHEADERSASSTRING Method

Returns the response headers from the HTTP method in character string format.

Syntax

package.**GETRESPONSEHEADERSASSTRING**(*variable*, *rc*);

Required Arguments

package
specifies an instance of the HTTP package variable.

variable
specifies the variable to hold the response headers.

rc
specifies the variable to hold the return code value.

Note: A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the GETRESPONSEHEADERSASSTRING method to retrieve all headers from the HTTP response. You can use the GETRESPONSEHEADERSASSTRING method

after using either the EXECUTEMETHOD method or the EXECUTEMETHODSTREAM method to send any HTTP request to the server.

Example

The following example creates and sends a HEAD method and uses the GETRESPONSEHEADERSASSTRING method to display the headers of the response from the web server.

```
data _null_;
  method init();
    declare package http h();
    declare varchar(1024) character set utf8 headers;
    declare int rc status;
    declare char(2) code;

    /* Build and send a HEAD method for a resource */
    code = 'GB';
    h.createHeadMethod('http://api.worldbank.org/countries/'
      || code || '/indicators/NY.GNP.PCAP.CD/?date=1990:1990');
    h.executeMethod();

    /* If the resource headers were returned by the server, show them */
    status = h.getStatusCode();   /* get the HTTP status code */
    put 'Country code:' code 'executeMethod() status:' status;
    if status eq 200 then do;     /* 200 = OK */
      /* retrieve the headers from the response that came from the server */
      h.getResponseHeadersAsString(headers, rc);
      put 'Headers:';
      put headers;
    end;
  end;
enddata; run;
```

The following lines are written to the log:

```
Country code: GB executeMethod() status: 200
Headers:
HTTP/1.1 200 OK
Content-Length: 0
Content-Type: text/xml; charset=UTF-8
Server: WorldBank Web
Server1
Date: Tue, 25 Mar 2014 21:05:48 GMT
X-Cache: MISS from transproxy
Via: 1.1 transproxy
(squid)
Connection: keep-alive
```

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEGETMETHOD Method" on page 1591
- "CREATEHEADMETHOD Method" on page 1596
- "CREATEPOSTMETHOD Method" on page 1598
- "EXECUTEMETHOD Method" on page 1601
- "EXECUTEMETHODSTREAM Method" on page 1602

GETSTATUSCODE Method

Returns the HTTP status code from the most recently executed HTTP method.

Syntax

status-code-variable=*package*.**GETSTATUSCODE**( );

Required Arguments

status-code-variable
specifies the variable to hold the HTTP status code value.

Note: Consult an HTTP reference for possible status codes.

package
specifies an instance of the HTTP package variable.

Details

Use the GETSTATUSCODE method to retrieve the HTTP status code from the most recently executed HTTP method.

Example

```
data _null_;
  method init();
    declare package http h();
    declare int status;

    /* Build and send a HEAD method for a resource */
    h.createHeadMethod('http://support.sas.com/documentation/');
    h.executeMethod();
```

```
    /* If the resource was returned by the server, show the response */
    status = h.getStatusCode();   /* get the HTTP status code */
    put 'HEAD method created for resource:';
    put 'http://support.sas.com/documentation/';
    put 'executeMethod() status:' status;
  end;
enddata; run;
```

The following lines are written to the log.

```
HEAD method created for resource:
http://support.sas.com/documentation/
executeMethod() status: 200
```

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "EXECUTEMETHOD Method" on page 1601
- "EXECUTEMETHODSTREAM Method" on page 1602

NEW Operator: HTTP Package

Constructs an instance of an HTTP package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable=**_NEW_ HTTP**( );

Required Argument

package-variable
specifies a name that can reference an instance of the package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a HTTP package variable using the DECLARE PACKAGE statement. After you declare a HTTP package variable, use the _NEW_ operator to instantiate an instance of the HTTP package and assign the new package instance to the HTTP package variable.

```
declare package http h;
h = _new_ http();
```

As an alternative to the two-step process of using the DECLARE PACKAGE statement and the _NEW_ operator to declare and instantiate an HTTP package, you can declare and instantiate a package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package http h();
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: HTTP Package" on page 1599
- "PACKAGE Statement" on page 1286

SETOAUTHTOKEN Method

Specifies a string that contains the OAuth access token for the request.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**SETOAUTHTOKEN**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the string variable that contains the OAuth access token.

Details

Open Authorization (OAuth) is an open standard for token-based authentication and authorization on the internet. OAuth is designed to work with HTTP and allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The third-party client then uses the access token to access the protected resources that are hosted by the resource server.

For the HTTP package, use the SETOAUTHTOKEN method to provide the OAuth access token that are added to the request header as a Bearer value of an Authorization header field.

For more information about OAuth, see OAuth 2.0.

Comparisons

You provide the OAuth access token with the SETOAUTHTOKEN method. You allow SAS to search the environment for an OAuth access token with the ADDSASOAUTHTOKEN method.

Example

Here is an example of GET a method that sets a bearer token to access an OAuth-protected resource.

```
proc ds2;
data _null_;
  method init();
    dcl package http h();
    dcl int statusCode;
    dcl varchar(32767) headers body;
    dcl int rc;

    h.createGetMethod();
    h.setURL('http://httpbin.org/bearer');
    h.setOAuthToken('mF_9.B5f-4.1JqM');
    h.executeMethod();
    statusCode = h.getStatusCode();
    put statusCode=;
    if (statusCode = 200) then do;
      h.getResponseHeadersAsString(headers, rc);
      h.setResponseBodyCharacterSet('utf8');
      h.getResponseBodyAsString(body, rc);
      put;
      put 'RESPONSE HEADERS';
      put headers;
```

```
        put;
        put 'RESPONSE BODY';
        put body;
      end;
    end;
  enddata;
  run; quit;
```

See Also

Methods:

- “ADDSASOAUTHTOKEN Method” on page 1590

SETPASSWORD Method

Specifies the string that represents the password associated with the specified user name.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**SETPASSWORD**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the string variable that contains the password.

Details

If the URL that you specify with the SETURL method requires a password, specify it with the SETPASSWORD method. Likewise, if the URL requires a user name, specify it with the SETUSERNAME method.

Comparisons

The SETPASSWORD method specifies the password of the non-proxy user name. The SETPROXYPASSWORD method specifies the password of the proxy user name.

See Also

Methods:

- “SETPROXYPASSWORD Method” on page 1615
- “SETURL Method” on page 1624
- “SETUSERNAME Method” on page 1626

SETPROXYPASSWORD Method

Specifies the string that represents the password associated with the specified proxy user name.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**SETPROXYPASSWORD**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the string variable that contains the password.

Details

If the proxy URL that you specify with the SETPROXYURL method requires a password, specify it with the SETPROXYPASSWORD method. Likewise, if the proxy URL requires a user name, specify it with the SETPROXYUSERNAME method.

Comparisons

The SETPROXYPASSWORD method specifies the password of the proxy user name. The SETPASSWORD method specifies the password of the non-proxy user name.

See Also

Methods:

- “SETPASSWORD Method” on page 1614
- “SETPROXYURL Method” on page 1616
- “SETPROXYUSERNAME Method” on page 1617

SETPROXYURL Method

Specifies the proxy URL to which the HTTP client should connect.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**SETPROXYURL**(*'variable'*);

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the variable that contains the proxy URL to which the HTTP client should connect.

Details

The SETPROXYURL method enables you to specify a proxy URL to which the HTTP client should connect. A proxy URL is a web address to access a proxy server through a web browser.

Comparisons

The SETPROXYURL specifies a proxy URL. The SETURL specifies a URL.

See Also

Methods:

- “SETPROXYPASSWORD Method” on page 1615
- “SETPROXYUSERNAME Method” on page 1617
- “SETURL Method” on page 1624

SETPROXYUSERNAME Method

Specifies the string that represents the domain-qualified user name of the user who is attempting to connect to the specified proxy URL.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**SETPROXYUSERNAME**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the variable that contains the domain-qualified user name.

Details

If the proxy URL that you specify with the SETPROXYURL method requires a user name, specify it with the SETPROXYUSERNAME method. Likewise, if the proxy URL requires a password, specify it with the SETPROXYPASSWORD method.

Comparisons

The SETPROXYUSERNAME specifies the domain-qualified user name of the user who attempts to connect to the specified proxy URL. The SETUSERNAME specifies the domain-qualified user name of the user who attempts to connect to the specified URL.

See Also

Methods:

- “SETPROXYPASSWORD Method” on page 1615
- “SETURL Method” on page 1624
- “SETUSERNAME Method” on page 1626

SETREQUESTBODYASBINARY Method

Adds the specified body to the HTTP method request in binary format.

Restriction: If you use the SETREQUESTBODYASBINARY method to return the entire body in binary format, you cannot use the SETREQUESTBODYCHARACTERSET method.

Syntax

package.**SETREQUESTBODYASBINARY**(*variable*);

Required Arguments

package
specifies an instance of the HTTP package variable.

variable
specifies the binary variable that contains the request body data.

Details

Use the SETREQUESTBODYASBINARY method to add the request body, in binary format, to the HTTP method.

To complete the HTTP method, indicate the content type of the body by calling the SETREQUESTCONTENTTYPE method, which sets the `Content-Type:` header. Then, call the EXECUTEMETHOD method to send the request to the web server.

Note: The content length is computed by the DS2 HTTP client after transcoding the data.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "ADDREQUESTHEADER Method" on page 1589
- "CREATEPOSTMETHOD Method" on page 1598
- "EXECUTEMETHOD Method" on page 1601
- "SETREQUESTBODYASSTRING Method" on page 1619
- "SETREQUESTBODYCHARACTERSET Method" on page 1620
- "SETREQUESTCONTENTTYPE Method" on page 1621

SETREQUESTBODYASSTRING Method

Adds the specified body to the HTTP method request in character string format.

Requirement: A character set for the request body must not be specified after setting the request body. A character set for encoding the request body must be specified with the SETREQUESTBODYCHARACTERSET method before setting the request body with the SETREQUESTBODYASSTRING method.

Syntax

package.**SETREQUESTBODYASSTRING**(*variable*);

Required Arguments

package
specifies an instance of the HTTP package variable.

variable
specifies the string variable that contains the request body data.

Details

Use the SETREQUESTBODYASSTRING method to add the request body, in character string format, to the HTTP method.

To complete the HTTP method, indicate the content type of the body by calling the SETREQUESTCONTENTTYPE method, which sets the `Content-Type:` header. Then, call the EXECUTEMETHOD method to send the request to the web server.

If the content type is not set, the HTTP client deduces a character set from the request content type. If the HTTP client is unable to deduce a character set from the request content type, the character set defaults to UTF-8. However, you can use the SETREQUESTBODYCHARACTERSET method to specify the character set to use when encoding the request body.

Note: The content length is computed by the DS2 HTTP client after transcoding the data.

Note: If the encoding of the provided request data differs from the specified content type, the data is transcoded to the encoding that is specified by content type.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "ADDREQUESTHEADER Method" on page 1589
- "CREATEPOSTMETHOD Method" on page 1598
- "EXECUTEMETHOD Method" on page 1601
- "SETREQUESTBODYASBINARY Method" on page 1618
- "SETREQUESTBODYCHARACTERSET Method" on page 1620
- "SETREQUESTCONTENTTYPE Method" on page 1621

SETREQUESTBODYCHARACTERSET Method

Specifies the character set to use when encoding the request body.

Restrictions: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

If you use the SETREQUESTBODYASBINARY method to return the entire body in binary format, you cannot use the SETREQUESTBODYCHARACTERSET method.

Requirement: A character set for the request body must not be specified after setting the request body. A character set for encoding the request body must be specified with the SETREQUESTBODYCHARACTERSET method before setting the request body with the SETREQUESTBODYASSTRING method.

Syntax

package.**SETREQUESTBODYCHARACTERSET**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the string variable that contains the character set specification.

See For a complete list of character set encoding values, see "Encoding Values in SAS Language Elements" in the *SAS National Language Support (NLS): Reference Guide*.

Details

If you do not specify the character set for encoding the request body, the HTTP client deduces a character set from the request content type. If the HTTP client is unable to deduce a character set from the request content type, the character set defaults to UTF-8.

See Also

Methods:

- "SETREQUESTBODYASBINARY Method" on page 1618
- "SETREQUESTBODYASSTRING Method" on page 1619
- "SETRESPONSEBODYCHARACTERSET Method" on page 1622

SETREQUESTCONTENTTYPE Method

Specifies the content type of the body of the HTTP method request.

Syntax

package.**SETREQUESTCONTENTTYPE**(*content-type*);

Required Arguments

content-type
specifies the variable that contains the content type value.

Note: Consult an HTTP reference for possible content types.

package
specifies an instance of the HTTP package variable.

Details

Use the SETREQUESTCONTENTTYPE method to specify the content type of the body of the HTTP method.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEPOSTMETHOD Method" on page 1598
- "EXECUTEMETHOD Method" on page 1601
- "SETREQUESTBODYASBINARY Method" on page 1618
- "SETREQUESTBODYASSTRING Method" on page 1619

SETRESPONSEBODYCHARACTERSET Method

Specifies the character set to use when decoding the response body.

Restrictions: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.
If you use the GETRESPONSEBODYASBINARY method to return the entire body from the HTTP response in binary format, you cannot use the SETRESPONSEBODYCHARACTERSET method.

Requirement: A character set for the response body must not be set after getting the response body. A character set for the response body must be specified with the SETRESPONSEBODYCHARACTERSET method before getting the request body with the GETRESPONSEBODYASSTRING method.

Syntax

package.**SETRESPONSEBODYCHARACTERSET**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the string variable that contains the character set specification.

See For a complete list of character set encoding values, see "Encoding Values in SAS Language Elements" in the *SAS National Language Support (NLS): Reference Guide*.

Details

If you do not specify the character set for encoding the response body, the HTTP client deduces a character set from the response content type. If the HTTP client is unable to deduce a character set from the response content type, the character set defaults to UTF-8.

See Also

Methods:

- "GETRESPONSEBODYASBINARY Method" on page 1603
- "GETRESPONSEBODYASSTRING Method" on page 1604
- "SETREQUESTBODYCHARACTERSET Method" on page 1620

SETSOCKETTIMEOUT Method

Specifies the socket time-out value to wait for a response from an HTTP web server.

Syntax

package.**SETSOCKETTIMEOUT**(*time-out-value*);

Required Arguments

package
specifies an instance of the HTTP package variable.

time-out-value
specifies the default socket time-out, in milliseconds, to wait for a response from the web server.

Details

Use the SETSOCKETTIMEOUT method to specify how long to wait for a response from the web server.

The SETSOCKETTIMEOUT method can be set for each new CREATE method when the default time-out is too long or too short.

Example

The following example sets the socket time-out value.

```
data _null_;
   method init();
      declare package http h();
      declare int status_code rc;

      /* get method call with 10 second delay */
      rc = h.createGetMethod('http://httpbin.org/delay/10');
      if (rc ne 0) then put 'TEST ERROR:' rc=;

      /* set 5 second timeout -- execute should timeout */
      rc = h.setSocketTimeout(5000);
      if (rc ne 0) then put 'TEST ERROR:' rc=;
      rc = h.executeMethod();
      if (rc eq 0) then put 'TEST ERROR:' rc=;
   end;
enddata;
run;
```

See Also

"Using the HTTP Package" in *SAS DS2 Programmer's Guide*

SETURL Method

Specifies the URL to which the HTTP client should connect.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**SETURL**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the variable that contains the URL to which the HTTP client should connect.

Details

The SETURL method enables you to set a URL to which the HTTP client should connect.

In order to use a URL that is set in the client, you must use the version of the CREATEGETMETHOD, CREATEHEADMETHOD, and CREATEPOSTMETHOD methods that accepts no arguments.

Here is an example of how the SETURL method is used:

```
h.createGetMethod();
h.setURL('http://httpbin.org/get');
```

Comparisons

The SETURL specifies a URL. The SETPROXYURL specifies a proxy URL.

See Also

Methods:

- "CREATEGETMETHOD Method" on page 1591
- "CREATEHEADMETHOD Method" on page 1596
- "CREATEPOSTMETHOD Method" on page 1598
- "SETPROXYURL Method" on page 1616
- "SETUSERNAME Method" on page 1626
- "SETPASSWORD Method" on page 1614

SETUSERNAME Method

Specifies the string that represents the domain-qualified user name of the user who is attempting to connect to the specified URL.

Restriction: This method is available beginning with SAS 9.4M6 and SAS Viya 3.4.

Syntax

package.**SETUSERNAME**('*variable*');

Required Arguments

package
specifies an instance of the HTTP package variable.

'variable'
specifies the string variable that contains the domain-qualified user name.

Details

If the URL that you specify with the SETURL method requires a user name, specify it with the SETUSERNAME method. Likewise, if the URL requires a password, specify it with the SETPASSWORD method.

Comparisons

The SETUSERNAME method specifies the domain-qualified user name of the user who attempts to connect to the specified URL. The SETPROXYUSERNAME method specifies the domain-qualified user name of the user who attempts to connect to the specified proxy URL.

See Also

Methods:

- "SETPROXYUSERNAME Method" on page 1617
- "SETPASSWORD Method" on page 1614
- "SETURL Method" on page 1624

STREAMRESPONSEBODYASBINARY Method

Streams the body, in chunks, from the HTTP response in binary format.

Syntax

package.**STREAMRESPONSEBODYASBINARY**(*variable*, *rc*);

Required Arguments

package
specifies an instance of the HTTP package variable.

variable
specifies the variable that will contain the response data chunk.

rc
specifies the variable to hold the return code value.

Note: A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the STREAMRESPONSEBODYASBINARY method to retrieve the response body in binary format. The response body is streamed, in chunks.

You can call the STREAMRESPONSEBODYASBINARY method after using the EXECUTEMETHODSTREAM method.

If you do not want to complete the streaming of the body data, call the ABORT method to stop the execution of the method.

Note: The EXECUTEMETHODSTREAM method does not support retrieval of the response body as one entity.

See Also

- "Using the HTTP Package" in *SAS DS2 Programmer's Guide*

Methods:

- "ABORT Method" on page 1588

- "EXECUTEMETHODSTREAM Method" on page 1602
- "STREAMRESPONSEBODYASSTRING Method" on page 1628

STREAMRESPONSEBODYASSTRING Method

Streams the body, in chunks, from the HTTP response in character string format.

Applies to: HTTP package

Syntax

package.**STREAMRESPONSEBODYASSTRING**(*variable*, *rc*);

Required Arguments

package
specifies an instance of the HTTP package variable.

variable
specifies the variable that will contain the response data chunk.

rc
specifies the variable to hold the return code value.

Note: A return code value of 0 indicates success; a value of 1 indicates failure.

Details

Use the STREAMRESPONSEBODYASSTRING method to retrieve the response body in character string format. The response body is streamed, in chunks.

You can call the STREAMRESPONSEBODYASBINARY method after using the EXECUTEMETHODSTREAM method.

If you do not want to complete the streaming of the body data, call the ABORT method to stop the execution of the method.

Note: The EXECUTEMETHODSTREAM method does not support retrieval of the response body as one entity.

See Also

- “Using the HTTP Package” in *SAS DS2 Programmer’s Guide*

Methods:

- “ABORT Method” on page 1588
- “EXECUTEMETHODSTREAM Method” on page 1602
- “STREAMRESPONSEBODYASBINARY Method” on page 1627

17

DS2 JSON Package Methods, Operators, and Statements

Dictionary **1632**
CREATEPARSER Method 1632
CREATEWRITER Method 1633
DECLARE PACKAGE Statement: JSON Package 1634
DELETE Method: JSON Package 1635
DESTROYPARSER Method 1636
DESTROYWRITER Method 1637
GETNEXTTOKEN Method 1638
ISBOOLEANFALSE Method 1640
ISBOOLEANTRUE Method 1641
ISFLOAT Method 1642
ISINTEGER Method 1643
ISLABEL Method 1644
ISLEFTBRACE Method 1644
ISLEFTBRACKET Method 1645
ISNULL Method 1646
ISNUMERIC Method 1647
ISPARTIAL Method 1648
ISRIGHTBRACE Method 1649
ISRIGHTBRACKET Method 1649
ISSTRING Method 1650
NEW Operator: JSON Package 1651
SETPARSERINPUT Method 1652
WRITEARRAYOPEN Method 1653
WRITEBOOLEANFALSE Method 1654
WRITEBOOLEANTRUE Method 1655
WRITECLOSE Method 1655
WRITEDOUBLE Method 1656
WRITENULL Method 1659
WRITEOBJOPEN Method 1660
WRITERGETTEXT Method 1660
WRITESTRING Method 1662

Dictionary

CREATEPARSER Method

Creates a JSON Parser instance.

Syntax

Form 1: *package*.**CREATEPARSER** ( );

Form 2: *package*.**CREATEPARSER** (*json-text*, *tipping-size*);

Form 3: *package*.**CREATEPARSER** (*json-text*);

Form 4: *package*.**CREATEPARSER** (*tipping-size*);

Required Arguments

package
specifies an instance of the JSON package.

json-text
specifies the input JSON text to be parsed.

Data type NCHAR, NVARCHAR

tipping-size
specifies the minimum number of characters of output JSON text to accumulate before calling the string call-back routine for strings longer than *tipping-size*.

Default 0, which indicates that only complete strings are returned to the string callback, regardless of length.

Restriction The maximum number of characters that can be returned is (*tipping-size* + 4) when *tipping-size* is set.

Data type INTEGER

Details

If you use Form 1 or Form 4 of the CREATEPARSER method syntax, you should subsequently call the SETPARSERINPUT method to provide the JSON text to be parsed.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "DESTROYPARSER Method" on page 1636
- "SETPARSERINPUT Method" on page 1652

CREATEWRITER Method

Creates a JSON writer instance.

Syntax

Form 1: *package*.**CREATEWRITER** ([PRETTY]);

Form 2:

Required Argument

package
specifies an instance of the JSON package.

Optional Argument

PRETTY
creates a more human-readable format that uses indention to illustrate the JSON container structure. Otherwise, the output is written in a single line.

Data type NVARCHAR

Note More flags might be available in future releases.

Details

The DS2 JSON package's writer currently does not support streaming. Instead, the Write instances are gathered in memory until retrieved by calling the WRITERGETTEXT method.

Example

For an example, see "WRITEARRAYOPEN Method" on page 1653.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "DESTROYWRITER Method" on page 1637
- "WRITERGETTEXT Method" on page 1660

DECLARE PACKAGE Statement: JSON Package

Creates a package variable and enables you to create an instance of the JSON package.

Category: Local

Syntax

DECLARE PACKAGE JSON *variable* ( );

Arguments

variable
specifies a name that can reference an instance of the JSON package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use a JSON package to create and parse JSON text. The JSON package is predefined for DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of a JSON package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package json j;
j = _new_ json();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package json j();
```

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: JSON Package" on page 1651

DELETE Method: JSON Package

Deletes a JSON package.

Syntax

package.**DELETE**( );

Required Argument

package
specifies an instance of the JSON package variable.

Details

When you no longer need the JSON package, delete it by using the DELETE method. The DELETE method is not required. When a JSON package's reference count drops to zero, the package is automatically deleted.

Note: It is possible to write DS2 code to create circular references such that the reference count might never drop to zero. In that case, the instance lives until the end of the RUN block unless it is explicitly deleted using the DELETE method. Here is an example of how to create a circular reference in DS2:

```
proc ds2;
package e / overwrite=yes;
   dcl package e next;
   dcl package e prev;

   method setnext(package e e);
      next = e;
   end;

   method setprev(package e e);
      prev = e;
   end;
endpackage;

data _null_;
   dcl package e e1();
   dcl package e e2();

   method init();
      e1.setnext(e2);
      e2.setprev(e1);
   end;
enddata;
run;
quit;
```

See Also

"Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

DESTROYPARSER Method

Destroys a JSON Parser instance.

Syntax

package.**DESTROYPARSER** ( );

Required Argument

package
specifies an instance of the JSON package.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEPARSER Method" on page 1632

DESTROYWRITER Method

Destroys a JSON writer instance.

Syntax

package.**DESTROYWRITER** ( );

Required Argument

package
specifies an instance of the JSON package.

Example

For an example, see "WRITEARRAYOPEN Method" on page 1653.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

■ "CREATEWRITER Method" on page 1633

GETNEXTTOKEN Method

Returns the next validated JSON language item or element from the JSON text.

Syntax

Form 1: *package*.**GETNEXTTOKEN** (*rc*, *token-type*, *parse-flags*,);

Form 2: *package*.**GETNEXTTOKEN** (*rc*, *token*, *token-type*, *parse-flags*,);

Form 3: *package*.**GETNEXTTOKEN** (*rc*, *token*, *token-type*, *parse-flags*, *line-number*, *column-number*);

Required Arguments

package
specifies an instance of the JSON package.

rc
specifies the variable to hold the return code value. Possible return code values are as follows:

0	Success
100	The output token argument's maximum length was not large enough and truncation occurred.
101	Done. Depending on the use case, this might or might not be expected.
300	End of text. Depending on the use case, this might or might not be expected.
301	An error occurred while parsing the text.

Data type INTEGER

token-type
token-type can be one of the following values:

4	Boolean true
8	Boolean false
16	Left bracket ([ )
32	Right bracket ([ )
64	Left brace ({)
128	Right brace (})

256 String

512 Numeric

1024 Null

Data type INTEGER

parse-flags

parse-flags output value can be an integer flag set consisting of one or more of the following flags:

0x00000001	token is a label in an object
0x00000002	token is not complete
0x00000003	token is an integral numeric
0x00000004	token is a floating point number

token

is the next token or string.

Data type NVARCHAR

line-number

Updates the given integer variable argument with the line number within the text where the token is located.

Data type BIGINT

Tip You can use the line number to help determine the location of the token within the text.

column-number

Updates the given integer variable argument with the column number within the text where the token is located.

Data type BIGINT

Tip You can use the column number to help determine the location of the token within the text.

Details

All of the arguments to the GETNEXTTOKEN method are passed by reference.

You can use the IS* methods to test the token type.

See Also

Methods:

- "ISBOOLEANFALSE Method" on page 1640
- "ISBOOLEANTRUE Method" on page 1641
- "ISFLOAT Method" on page 1642
- "ISINTEGER Method" on page 1643
- "ISLABEL Method" on page 1644
- "ISLEFTBRACE Method" on page 1644
- "ISLEFTBRACKET Method" on page 1645
- "ISNULL Method" on page 1646
- "ISNUMERIC Method" on page 1647
- "ISPARTIAL Method" on page 1648
- "ISRIGHTBRACE Method" on page 1649
- "ISRIGHTBRACKET Method" on page 1649
- "ISSTRING Method" on page 1650

ISBOOLEANFALSE Method

Returns true if the token is false.

Syntax

package.**ISBOOLEANFALSE** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISBOOLEANFALSE method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISBOOLEANTRUE Method" on page 1641
- "WRITEBOOLEANFALSE Method" on page 1654

ISBOOLEANTRUE Method

Returns true if the token is true.

Syntax

package.**ISBOOLEANTRUE** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISBOOLEANTRUE method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISBOOLEANFALSE Method" on page 1640

- "WRITEBOOLEANTRUE Method" on page 1655

ISFLOAT Method

Returns true if the token is a floating point number in text form.

Syntax

package.**ISFLOAT** (*token-type*, *parse-flags*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

parse-flags
specifies the parse flags that were obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISFLOAT method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISINTEGER Method" on page 1643
- "ISNUMERIC Method" on page 1647

ISINTEGER Method

Returns true if the token is an integer in text form.

Syntax

package.**ISINTEGER** (*token-type*, *parse-flags*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

parse-flags
specifies the parse flags that were obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISINTEGER method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISFLOAT Method" on page 1642
- "ISNUMERIC Method" on page 1647

ISLABEL Method

Returns true if the token is an object label.

Syntax

package.**ISLABEL** (*token-type*, *parse-flags*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

parse-flags
specifies the parse flags that were obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISLABEL method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638

ISLEFTBRACE Method

Returns true if the token is a left brace ({).

Syntax

package.**ISLEFTBRACE** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISLEFTBRACE method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISRIGHTBRACE Method" on page 1649

ISLEFTBRACKET Method

Returns true if the token is a left bracket ([ ).

Syntax

package.**ISLEFTBRACKET** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
specifies the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISLEFTBRACKET method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISRIGHTBRACKET Method" on page 1649

ISNULL Method

Returns true if the token is null.

Syntax

package.**ISNULL** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
receives the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISNULL method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "WRITENULL Method" on page 1659

ISNUMERIC Method

Returns true if the token is numeric in text form.

Syntax

package.**ISNUMERIC** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
receives the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISNUMERIC method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISFLOAT Method" on page 1642
- "ISINTEGER Method" on page 1643

ISPARTIAL Method

Returns true if the token is incomplete because of tipping.

Syntax

package.**ISPARTIAL** (*parse-flags*);

Required Arguments

package
specifies an instance of the JSON package.

parse-flags
receives the parse flags that were obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISPARTIAL method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638

ISRIGHTBRACE Method

Returns true if the token is a right brace (}).

Syntax

package.**ISRIGHTBRACE** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
receives the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISRIGHTBRACE method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISLEFTBRACE Method" on page 1644

ISRIGHTBRACKET Method

Returns true if the token is a right bracket ( ]).

Syntax

package.**ISRIGHTBRACKET** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
receives the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISRIGHTBRACKET method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Parsing JSON Text" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638
- "ISLEFTBRACKET Method" on page 1645

ISSTRING Method

Returns true if the token is a string.

Syntax

package.**ISSTRING** (*token-type*);

Required Arguments

package
specifies an instance of the JSON package.

token-type
receives the token type that was obtained from the GETNEXTTOKEN method.

Data type INTEGER

Details

You can use the ISSTRING method to test the token type that was obtained from the GETNEXTTOKEN method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNEXTTOKEN Method" on page 1638

NEW Operator: JSON Package

Constructs an instance of a JSON package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable = **_NEW_ JSON**( );

Required Argument

package-variable
specifies a name that can reference an instance of the package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a JSON package variable using the DECLARE PACKAGE statement. After you declare a JSON package variable, use the _NEW_ operator to instantiate

an instance of the JSON package and assign the new package instance to the JSON package variable.

```
declare package json jsontxt;
jsontxt = _new_ json( );
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a JSON package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package json jsontxt( );
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: JSON Package" on page 1634

SETPARSERINPUT Method

Provides JSON text to the parser when it needs more text.

Restriction: This method is valid only if the parser does not have any text; an error is returned if it does.

Syntax

package.**SETPARSERINPUT** ([*json-text*,]);

Required Argument

package
specifies an instance of the JSON package.

Optional Argument

json-text
specifies the JSON text for the parser.

Data type CHAR, VARCHAR, NCHAR, NVARCHAR

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEPARSER Method" on page 1632

WRITEARRAYOPEN Method

Writes the open bracket ([ ) signifying the beginning of an array.

Syntax

package.**WRITEARRAYOPEN** ( );

Required Argument

package
specifies an instance of the JSON package.

Details

The WRITEARRAYOPEN method explicitly opens an array container, which you must explicitly close with the WRITECLOSE method.

Example

The following example creates a writer instance and writes a numeric value to a JSON array container.

```
data _null_;
    method init();
        dcl package json j();
```

```
        dcl double dblVal;
        dcl int rc;
        dcl nvarchar(15) jsontxt;

        rc = j.createWriter();
        rc = j.writeArrayOpen();
        dblVal = 12345678.1234;
        rc = j.writeDouble( dblVal,13, 5 );
        rc = j.writeClose();
        j.writerGetText( rc, jsontxt);
        put jsontxt=;
        rc = j.destroywriter();
    end;
enddata;
  run;
```

The following line is written to the SAS log:

```
jsontxt=[1.2346e+07   ]
```

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "WRITECLOSE Method" on page 1655

WRITEBOOLEANFALSE Method

Writes a Boolean false value to the text.

Syntax

package.**WRITEBOOLEANFALSE** ( );

Required Argument

package
specifies an instance of the JSON package.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- “ISBOOLEANFALSE Method” on page 1640
- “WRITEBOOLEANTRUE Method” on page 1655

WRITEBOOLEANTRUE Method

Writes a Boolean true value to the text.

Syntax

package.**WRITEBOOLEANTRUE** ( );

Required Argument

package
specifies an instance of the JSON package.

See Also

- “Using the JSON Package” in *SAS DS2 Programmer’s Guide*

Methods:

- “ISBOOLEANTRUE Method” on page 1641
- “WRITEBOOLEANFALSE Method” on page 1654

WRITECLOSE Method

Closes the corresponding object (}) or array ( ]).

Syntax

package.**WRITECLOSE** ( );

Required Argument

package
specifies an instance of the JSON package.

Details

The WRITECLOSE method closes the most recently opened container of either type that was explicitly opened with the WRITEARRAYOPEN or WRITEOBJOPEN method. You should call the WRITECLOSE method for containers only if you explicitly opened the container with a WRITE*OPEN method.

Example

For an example, see "WRITEARRAYOPEN Method" on page 1653.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "WRITEOBJOPEN Method" on page 1660
- "WRITEARRAYOPEN Method" on page 1653

WRITEDOUBLE Method

Writes a DOUBLE value in text form.

Syntax

Form 1: *package*.**WRITEDOUBLE** (*double-value*);

Form 2: *package*.**WRITEDOUBLE** (*double-value*, *width*);

Form 3: *package*.**WRITEDOUBLE** (*double-value*, *width*, *precision*);

Form 4: *package*.**WRITEDOUBLE** (*double-value*, *width*, *precision*, *options*);

Required Arguments

package
specifies an instance of the JSON package.

double-value
specifies the value to be written.

Data type DOUBLE

width
specifies the width of the formatted value.

Default 0

Data type DOUBLE

precision
indicates the number of digits that appear after the radix character.

Default 15

Data type DOUBLE

options
specifies a flag for formatting. *options* can be one of the following values:

BESTFIT
formats the value using decimal notation in the form of [–]*dddd.ddd* or scientific notation in the form of [–]*d.ddd*e±*dd*. The formatting style depends on the value.

Notes Valid width is 1–32 characters.

Precision is ignored.

BESTFITBIG
formats the value using decimal notation in the form of [–]*dddd.ddd* or scientific notation in the form of [–]*d.ddd*E±*dd*. The formatting style depends on the value.

Notes Valid width is 1–32 characters.

Precision is ignored.

SASBEST
Conforms to the BESTw. format rules. The value is formatted within the specified width. Decimal notation is produced if possible. Otherwise, scientific notation is produced in the style [–]*ddd*.dddE[–]*dd*

Note Trailing zeros after the radix character are suppressed.

SASEW
Conforms to the E*w*. format rules. The value is formatted within the specified width. Scientific notation is always produced in the style [-]*ddd.ddd*E±*dd*.

Notes Valid width is 7–32 characters.

Precision is ignored.

SASEWD
Conforms to the SAS XP Services %*w.d*e rules. The value is formatted within the specified *width*. Scientific notation is always produced in the style [–]*ddd.ddd*E±*dd*.

Notes Valid width is 7–32 digits.

Valid precision is 0–31 digits.

SASWD
Conforms to the *w.d* format rules.

Note The value is formatted within the specified *width*. If *width* is too small, it reverts to the behavior indicated by SASBEST.

DECIMAL
Formats the value using decimal notation in the style [–]*dddd.dd*, using *precision* to determine the number of digits after the radix.

Note The radix character does not appear if there are no digits to display after it or if *precision* is set to zero.

FRACTION
Formats the fractional part of the value in the style of [–]0.*dddd*.

Note The digit before the radix character is always zero.

INTEGER
Formats the fractional part of the value in the style of [–]*dddd*.

Note The fractional part of the value is ignored and no radix character is added to the result.

SNOTE
formats the value using scientific notation in the form of [–]*d.ddd*e±*dd*, using the lowercase 'e' to precede the exponent.

SNOTEBIG
formats the value using scientific notation in the form of [–]*d.ddd*E±*dd*, using the uppercase 'E' to precede the exponent.

Default BESTFIT

Data type CHAR

Example

The following example writes DOUBLE values to an array.

```
data _null_;
    dcl package logger lgr( 'App.tk.D2PKG.JSON' );
    dcl package json j();
    dcl nvarchar(256) matrixA;
    dcl int rc;

    method init();
        rc = j.createWriter();
        rc = j.writeArrayOpen();
```

```
        rc = j.writeArrayOpen();
        rc = j.writeDouble(1.1);
        rc = j.writeDouble(1.2);
        rc = j.writeClose();
        rc = j.writeArrayOpen();
        rc = j.writeDouble(2.1);
        rc = j.writeDouble(2.2);
        rc = j.writeClose();
        rc = j.writeClose();
        j.writerGetText( rc, matrixA );
        lgr.log( 4, 'matrix A = $s', matrixA );
        rc = j.destroyWriter();
    end;
enddata;
run;
```

The following lines are written to the SAS log:

```
NOTE: matrix A = [[1.1,1.2],[2.1,2.2]]
```

See Also

"Using the JSON Package" in *SAS DS2 Programmer's Guide*

WRITENULL Method

Writes a null value to the text.

Syntax

package.**WRITENULL** ( );

Required Argument

package
specifies an instance of the JSON package.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "ISNULL Method" on page 1646

WRITEOBJOPEN Method

Writes the open brace ({) signifying the beginning of an object.

Syntax

package.**WRITEOBJOPEN** ( );

Required Argument

package
specifies an instance of the JSON package.

Details

The WRITEOBJOPEN method explicitly opens an object container, which you must explicitly close with the WRITECLOSE method.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "WRITECLOSE Method" on page 1655

WRITERGETTEXT Method

Obtains the JSON text that is produced by the writer.

Syntax

package.**WRITERGETTEXT** (*rc*, *json-text*);

Required Arguments

package
specifies an instance of the JSON package.

rc
specifies the variable to hold the return code value. Possible return code values are as follows:

0	Success
100	The output token argument's maximum length was not large enough and truncation occurred.
101	Done. Depending on the use case, this might or might not be expected.
300	End of text. Depending on the use case, this might or might not be expected.
301	A status condition was returned.

Data type INTEGER

json-text
specifies a variable that receives the JSON text.

Data type NVARCHAR

Details

The DS2 JSON package's writer currently does not support streaming. Instead, the Write instances are gathered in memory until retrieved by calling the WRITERGETTEXT method.

Example

For an example, see "WRITEARRAYOPEN Method" on page 1653.

See Also

- "Using the JSON Package" in *SAS DS2 Programmer's Guide*

Methods:

- "CREATEWRITER Method" on page 1633

WRITESTRING Method

Writes a string to the JSON text.

Note: Braces in the syntax convention indicate a syntax grouping.

Syntax

package.**WRITESTRING** ({[' | "]}*string*{[' | "]}, *flags*);

Required Arguments

package
specifies an instance of the JSON package.

{['"] }*string*{['"] }
specifies the string to write.

Data type NVARCHAR

Tip The string can be a string literal in single quotation marks, a normal identifier without quotation marks, or a delimited identifier in double quotation marks.

flags
specifies options for special handling of the string. The following values are possible:

0
0 indicates no flags.

16
16 (JSN_SkipScan) indicates that the normal scanning and JSON encoding of the input string should be skipped. This means that either the string is known to contain no invalid or JSON escape characters, or the caller has already performed JSON scanning or encoding on the string. In the latter case, only quotation marks and separators would be inserted with the given string.

For normal scanning, omit the flag so that normal scanning and JSON encoding can proceed.

32
32 (JSN_TrimBlanks) causes the writer to trim trailing blanks from the string.

48
48 causes both the writer to skip scanning and trim blanks.

Data type INTEGER

Example

The following example illustrates different types of string values.

```
data _null_;
    dcl package logger lgr( 'App.tk.D2PKG.JSON' );
    dcl package json j();
    dcl nvarchar(2000) txt;
    dcl varchar(20) "a**b";
    dcl varchar(20) myStr;
    dcl int rc;

    method init();
        rc = j.createWriter();
        rc = j.writeArrayOpen();
        rc = j.writeString( 'aaaAAA', 0 );
        "a**b" = 'bbbBBB';
        rc = j.writeString( "a**b", 0 );
        myStr = 'cccCCC';
        rc = j.writeString( myStr, 0 );
        rc = j.writeClose();
        j.writerGetText( rc, txt );
        lgr.log( 3, 'txt = $s', txt );
        rc = j.destroyWriter();
    end;
enddata;
run;
```

See Also

"Using the JSON Package" in *SAS DS2 Programmer's Guide*

18

DS2 Logger Package Methods, Operators, and Statements

Dictionary 1665
DECLARE PACKAGE Statement: Logger Package 1665
DELETE Method: Logger Package 1667
ISLEVELACTIVE Method 1668
LOG Method: Logger Package 1669
NEW Operator: Logger Package 1672

Dictionary

DECLARE PACKAGE Statement: Logger Package

Creates a package variable and gives you the option of creating an instance of the logger package.

Category: Local

Tip: The PACKAGE statement is not required for a logger package.

Syntax

DECLARE PACKAGE LOGGER *variable* ([*logger-name*]);

Arguments

variable
specifies a name that can reference an instance of the logger package.

logger-name
specifies the name of the logger that is defined in the SAS logging facility.

Default SAS root logger

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use a logger package to interface with the SAS logging facility. The logger package is predefined for DS2 programs. For more information about the logging facility, see *SAS Logging: Configuration and Programming Reference*.

You declare a logger package by using the DECLARE PACKAGE statement. This associates a logger package with a logger name. After you declare the new logger package, you can send messages to the logger at a specified logging level.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of a logger package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package logger logpkg;
logpkg = _new_ logger();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package logger logpkg();
```

For more information about the logger package, see "Using the Logger Package" in *SAS DS2 Programmer's Guide*.

Example

This example creates an instance of a logger package.

```
data _null_;
   dcl package logger l();
   method init();
      l.log('i', 'Hello World!');
   end;
```

```
enddata;
```

See Also

- "Using the Logger Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: Logger Package" on page 1672

DELETE Method: Logger Package

Deletes a logger package.

Note: The DELETE method is not required. When no logger package variables reference a logger package instance, the package instance is automatically deleted.

Syntax

package.**DELETE**( );

Required Argument

package
specifies an instance of the logger package variable.

Details

When you no longer need the logger package, delete it by using the DELETE method. If you attempt to use a logger package after you delete it, an error will be written to the log.

See Also

"Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

ISLEVELACTIVE Method

Returns a value that indicates whether the logger package that is associated with the logger name suppresses a message at the logger level.

Syntax

package.**ISLEVELACTIVE**(['])*level*(']);

Required Arguments

package
specifies an instance of the logger package variable.

[']*level*[']
a numeric value that specifies the level at which a logging request is applied for the specified logger package.

Requirements *level* must be a string that contains either the severity value or a one-character abbreviation for the severity value. Valid values are listed as follows.

- 2 or 'T' for TRACE
- 3 or 'D' for DEBUG
- 4 or 'I' for INFO
- 5 or 'W' for WARN
- 6 or 'E' for ERROR
- 7 or 'F' for FATAL

If a character is used for *level*, the character must be enclosed in single quotation marks. Numeric values can be quoted but do not need to be.

Example

These are examples of the ISLEVELACTIVE method.

```
mylog.islevelactive(5);

testlog.islevelactive('F');
```

See Also

- *SAS Logging: Configuration and Programming Reference*
- "Using the Logger Package" in *SAS DS2 Programmer's Guide*

LOG Method: Logger Package

Send the specified message to the logger at the specified level.

Note:

Syntax

Form 1: *package*.**LOG**(['])*level*['], '*raw-message*');

Form 2: *package*.**LOG**(['])*level*['], [*message-format*]'*argument-1*' [..., *argument-9*]);

Required Arguments

package
specifies an instance of the logger package variable.

['])*level*['])
a numeric value that specifies the level at which a logging request is applied for the specified logger package.

Requirements *level* must be a string that contains either the severity value or a one-character abbreviation for the severity value. Valid values are listed as follows. Note that any part of the word that indicates the level is valid. For example, *i*, *in*, *inf*, or *info* is valid.

- 2 or 'T' for TRACE
- 3 or 'D' for DEBUG
- 4 or 'I' for INFO
- 5 or 'W' for WARN
- 6 or 'E' for ERROR
- 7 or 'F' for FATAL

If a character is used for *level*, the character must be enclosed in single quotation marks. Numeric values can be quoted but do not need to be.

'raw-message'
specifies the message to write at the level.

Range 1–65535 characters

Requirement The message is a string and must be enclosed in single quotation marks.

Tip The message can be any character type expression. Here is an example, `x.log(5, 'Error while processing function' || trimn(FNAME));`. However, using a character expression causes a conversion from a CHAR data type to an NCHAR data type. It is faster to use a character string.

argument
specifies a value that replaces the $s format marker in the *message-format*.

Interaction If the LOG method has more than two parameters and the level is valid, each $s format marker in the *message-format* is replaced by the content of the corresponding *argument*.

Tip Extra arguments are ignored when using formatted output.

Optional Argument

message-format
specifies a message format that is used to produce the log message. The message format contains at least one $s format marker.

Requirement The number of $s format markers must be less than or equal to the number of arguments. Otherwise, an error occurs.

Interaction If the LOG method has more than two parameters and the level is valid, each $s format marker is replaced by the content of the corresponding *argument*.

Tip To display a dollar sign ($) in your message, use $$ in the $s format marker.

Details

Unformatted Messages (Form 1)

A LOG method call with exactly two parameters, an active logger, and a valid level, sends the specified message to the associated logger in its raw format.

Formatted Messages (Form 2)

A LOG method call with more than two parameters, an active logger, and a valid level, sends a formatted message to the associated logger. Each $s format marker in the *message-format* is replaced by the content of the corresponding *argument*.

Examples

Example 1: Unformatted Messages

These are examples of unformatted messages.

```
mylog.log('T', 'The output was written to the log');

testlog.log(7, 'The output could not be written');
```

Example 2: Formatted Messages

The following example shows several combinations of $s format markers.

```
data _NULL_;
   dcl package logger root();
   method init();
      /* $$ is not evaluated here - one parameter */
      root.log(n'note', 'one-parm dollar pair: $$');
      /* $s is not evaluated here - one parameter */
      root.log(n'note', 'one-parm dollar-s: $s');
      /* $$ is evaluated here */
      root.log(n'note', 'two-parm dollar pair: $$', n'');
      /* $$ is evaluated here and "mine" is substituted for $s */
      root.log(n'note', 'dollar pair: $$; me:$s', n'mine');
      /* "mine" is substituted for $s */
      root.log(n'note', 'me:$s', n'mine');
      /* "mine" is substituted for the first $s */
      /* "thine" is substituted for the second $s */
      root.log(n'note', 'me:$s thee:$s', n'mine', n'thine');
      /* there are five arguments */
      root.log(n'note', 'one:$s two:$s three:$s four:$s five:$s', 1N, 2, 3.0, n'4', '5');
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
NOTE: hello world
NOTE: one-parm dollar pair: $$
NOTE: one-parm dollar-s: $s
NOTE: two-parm dollar pair: $
NOTE: dollar pair: $; me:mine
NOTE: me:mine
NOTE: me:mine thee:thine
NOTE: one:1 two:2 three:3 four:4 five:5
```

See Also

- *SAS Logging: Configuration and Programming Reference*
- "Using the Logger Package" in *SAS DS2 Programmer's Guide*

NEW Operator: Logger Package

Constructs an instance of a logger package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable = **_NEW_ LOGGER**([*logger-name*]);

Required Argument

package-variable
specifies a name that can reference an instance of the package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a logger package variable using the DECLARE PACKAGE statement. After you declare a logger package variable, use the _NEW_ operator to instantiate an instance of the logger package and assign the new package instance to the logger package variable.

```
declare package logger mylog;
mylog = _new_ logger( );
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a logger package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package logger mylog( );
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the Logger Package" in *SAS DS2 Programmer's Guide*

- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: Logger Package" on page 1665

19

DS2 Matrix Package Methods, Operators, and Statements

Dictionary **1676**
ABS Method 1676
ADD Method: Matrix Package 1677
ALL_AND Method 1679
ALL_EQ Method 1680
ALL_GE Method 1682
ALL_GT Method 1683
ALL_LE Method 1684
ALL_LT Method 1685
ALL_NE Method 1687
ALL_OR Method 1688
AND Method 1689
ANY_AND Method 1690
ANY_EQ Method 1692
ANY_GE Method 1693
ANY_GT Method 1694
ANY_LE Method 1695
ANY_LT Method 1697
ANY_NE Method 1698
ANY_OR Method 1700
COLS Method 1701
COPY Method 1702
DECLARE PACKAGE Statement: Matrix Package 1703
DELETE Method: Matrix Package 1705
DET Method 1705
EDIV Method 1706
EMAX Method 1708
EMIN Method 1710
EMOD Method 1712
EMULT Method 1714
EPOW Method 1715
EQ Method 1717
EXP Method 1719
FLOOR Method 1720

GE Method ... 1721
GT Method ... 1723
IN Method ... 1724
INVERSE Method ... 1727
LE Method ... 1729
LOG Method: Matrix Package ... 1731
LT Method ... 1732
NEW Operator: Matrix Package ... 1734
MULT Method ... 1736
NE Method ... 1739
OR Method ... 1741
OUT Method ... 1742
ROWS Method ... 1744
SQRT Method ... 1745
SUB Method ... 1746
TOARRAY Method ... 1748
TOVARARRAY Method ... 1749
TRANS Method ... 1751

Dictionary

ABS Method

Returns a matrix that contains the absolute value of each value in the input matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package*.**ABS**( );

Required Arguments

r
specifies a matrix that contains the absolute value of each value in the input matrix.

package
specifies an instance of the matrix package variable.

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

ADD Method: Matrix Package

Adds one matrix to another.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**ADD**(*package-2*);

Required Arguments

r
specifies the matrix that is automatically created by the ADD method.

package-1
specifies an instance of the first matrix package variable that is used in the addition operation.

package-2
specifies an instance of the second matrix package variable that is used in the addition operation.

Details

The matrix dimensions for the ADD method must be the same size in order for the matrix addition to take place. Each [i, j] element in the first matrix is added to its corresponding [i, j] element in the second matrix.

If you add matrices that have missing values, you do not receive an error.

It is also possible to perform scalar addition by using a 1x1 matrix.

Examples

Example 1: Adding Two Matrices

Here is an example of using the ADD method to add two 3x3 matrices. Each [i, j] element in the first matrix is added to its corresponding [i, j] element in the second matrix.

```
data _null_;
   dcl double a[3,3];
   dcl double b[3,3];
   dcl double c[3,3];

   method run();
      dcl package matrix m;
      dcl package matrix m2;
      dcl package matrix r;
      dcl double i j;

      a := (1,2,3,4,5,6,7,8,9);
      b := (1,5,9,2,6,10,3,7,1);

      m = _new_ matrix(a, 3, 3);
      m2 = _new_ matrix(b, 3, 3);

      r = m.add(m2);
      r.toarray(c);

      do i = 1 to 3;
         do j = 1 to 3;
             put c[i,j];
         end;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
2
7
12
6
11
16
10
15
10
```

Example 2: Scalar Addition of Two Matrices

Here is an example of how to perform scalar addition by using a 1x1 matrix.

```
data _null_;
   dcl double c[3,3];
   dcl double d[1,1];
   dcl double f[3,3];
   method run();
      dcl package matrix m3;
      dcl package matrix m4;
      dcl package matrix r;
      dcl double i j;

      c := (-0, 0, -1, 1, -2.2, 2.2, -3.3, 4.4, 5.5);
      d := (1);
```

```
      m3 = _new_ matrix(c, 3, 3);
      m4 = _new_ matrix(d, 1, 1);

      r = m3.add(m4);
      r.toarray(f);

      do i = 1 to 3;
         do j = 1 to 3;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

In this example, 1 was added to each entry in matrix *m3*. The scalar addition produces a 3x3 matrix that has the following values:

```
1       1      0
2       -1.2    3.2
-2.3    5.4    6.5
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "SUB Method" on page 1746

ALL_AND Method

Produces a scalar result in an ALL_AND comparison between all elements in one matrix and all elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_AND**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL AND comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL AND comparison.

Details

The ALL_AND logical operation produces a result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the logical [i, j]operations must be true. Otherwise, the result is 0.

The matrix sizes must match or you can also use a scalar comparison.

Comparisons

The ANY_AND operation is similar to the ALL_AND operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_and(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_OR Method" on page 1688
- "AND Method" on page 1689
- "ANY_AND Method" on page 1690

ALL_EQ Method

Produces a scalar result in an ALL_EQ (ALL equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_EQ**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL equal-to comparison.

Details

The ALL_EQ relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the [i, j]element comparisons must be true. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_EQ operation is similar to the ALL_EQ operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_eq(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_NE Method" on page 1687
- "ANY_EQ Method" on page 1692

ALL_GE Method

Produces a scalar result in an ALL_GE (ALL greater-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_GE**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL greater-than-or-equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL greater-than-or-equal-to comparison.

Details

The ALL_GE relational operation produces a scalar result that indicates whether an $[i, j]^{th}$ element of the first matrix satisfies the comparison with the $[i, j]^{th}$ element of the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the [i, j] element comparisons must be true. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_GE operation is similar to the ALL_GE operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_ge(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_LE Method" on page 1684
- "ALL_NE Method" on page 1687
- "ANY_GE Method" on page 1693

ALL_GT Method

Produces a scalar result in an ALL_GT (ALL greater-than) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_GT**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL greater-than comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL greater-than comparison.

Details

The ALL_GT relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of

the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the [i, j]element comparisons must be true. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_GT operation is similar to the ALL_GT operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_gt(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_GE Method" on page 1682
- "ALL_LT Method" on page 1685
- "ANY_GT Method" on page 1694

ALL_LE Method

Produces a scalar result in an ALL_LE (ALL less-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_LE**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL less-than comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL less-than comparison.

Details

The ALL_LE relational operation produces a scalar result that indicates whether an $[i, j]^{th}$ element of the first matrix satisfies the comparison with the $[i, j]^{th}$ element of the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the [i, j] element comparisons must be true. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_LE operation is similar to the ALL_LE operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_le(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_GE Method" on page 1682
- "ALL_LT Method" on page 1685
- "ANY_LE Method" on page 1695

ALL_LT Method

Produces a scalar result in an ALL_LT (ALL less-than) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_LT**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL less-than comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL less-than comparison.

Details

The ALL_LT relational operation produces a scalar result that indicates whether an $[i, j]^{th}$ element of the first matrix satisfies the comparison with the $[i, j]^{th}$ element of the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the [i, j] element comparisons must be true. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_LT operation is similar to the ALL_LT operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_lt(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_GT Method" on page 1683
- "ALL_LE Method" on page 1684
- "ANY_LT Method" on page 1697

ALL_NE Method

Produces a scalar result in an ALL_NE (ALL not-equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_NE**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL not-equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL not-equal-to comparison.

Details

The ALL_NE relational operation produces a scalar result that indicates whether an $[i, j]^{th}$ element of the first matrix satisfies the comparison with the $[i, j]^{th}$ element of the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the [i, j] element comparisons must be true. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_NE operation is similar to the ALL_NE operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_ne(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_EQ Method" on page 1680
- "ANY_NE Method" on page 1698

ALL_OR Method

Produces a scalar result in an ALL_OR comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ALL_OR**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ALL OR comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ALL OR comparison.

Details

The ALL_OR logical operation produces a result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. In order for the result to be 1, all of the logical [i, j] operations must be true. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ANY_OR operation is similar to the ALL_OR operation except that if any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

Example

```
x=m1.all_or(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_AND Method" on page 1679
- "ANY_OR Method" on page 1700
- "OR Method" on page 1741

AND Method

Compares two matrices based on the AND logical operation, and returns the resulting matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**AND**(*package-2*);

Required Arguments

r
specifies a matrix that contains the results of an AND comparison between the values of two matrices.

package-1
specifies an instance of the first matrix package variable that is used in the AND comparison.

package-2
specifies an instance of the second matrix package variable that is used in the AND comparison.

Details

The AND logical operator behaves similarly to the binary relational operations (LT, LE, GE, GT, NE, and EQ). In each case, the AND logical operation is applied to the [i, j]th elements of two matrices and placed in the result matrix *r*.

The matrix sizes must match or you can use a scalar comparison.

Example

```
r=m1.and(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_AND Method" on page 1679
- "ANY_AND Method" on page 1690
- "OR Method" on page 1741

ANY_AND Method

Produces a scalar result in an ANY_AND comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_AND**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY AND comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY AND comparison.

Details

The ANY_AND logical operation produces a scalar result that indicates whether an $[i, j]^{th}$ element of the first matrix satisfies the comparison with the $[i, j]^{th}$ element of the second matrix. The scalar result is 0 or 1. If any of the logical [i, j] operations is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_AND operation is similar to the ANY_AND operation except that all of the logical [i, j] operations have to be true for the result to be 1. Otherwise, the result is 0.

Example

```
x=m1.any_and(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_OR Method" on page 1688
- "AND Method" on page 1689
- "ANY_OR Method" on page 1700

ANY_EQ Method

Produces a scalar result in an ANY_EQ (ANY equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_EQ**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY equal-to comparison.

Details

The ANY_EQ relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_EQ operation is similar to the ANY_EQ operation except that all of the logical [i, j] operations have to be true for the result to be 1. Otherwise, the result is 0.

Example

```
x=m1.any_eq(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_EQ Method" on page 1680
- "ANY_NE Method" on page 1698
- "EQ Method" on page 1717

ANY_GE Method

Produces a scalar result in an ANY_GE (ANY greater-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_GE**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY greater-than-or-equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY greater-than-or-equal-to comparison.

Details

The ANY_GE relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element

of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_GE operation is similar to the ANY_GE operation except that all of the logical [i, j] operations have to be true for the result to be 1. Otherwise, the result is 0.

Example

```
x=m1.any_ge(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_GE Method" on page 1682
- "ANY_LE Method" on page 1695
- "ANY_NE Method" on page 1698

ANY_GT Method

Produces a scalar result in an ANY_GT (ANY greater-than) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_GT**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY greater-than comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY greater-than comparison.

Details

The ANY_GT relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_GT operation is similar to the ANY_GT operation except that all of the logical [i, j] operations has to be true for the result to be 1. Otherwise, the result is 0.

Example

```
x=m1.any_gt(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_GT Method" on page 1683
- "ANY_GE Method" on page 1693
- "ANY_LT Method" on page 1697

ANY_LE Method

Produces a scalar result in an ANY_LE (any less-than-or-equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_LE**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY less-than comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY less-than comparison.

Details

The ANY_LE relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_LE operation is similar to the ANY_LE operation except that all of the logical [i, j] operations has to be true for the result to be 1. Otherwise, the result is 0.

Example

```
x=m1.any_le(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_LE Method" on page 1684
- "ANY_GE Method" on page 1693
- "ANY_LT Method" on page 1697

ANY_LT Method

Produces a scalar result in an ANY_LT (ANY less-than) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_LT**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY less-than comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY less-than comparison.

Details

The ANY_LT relational operation produces a scalar result that indicates whether an $[i, j]^{th}$ element of the first matrix satisfies the comparison with the $[i, j]^{th}$ element of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_LT operation is similar to the ANY_LT operation except that all of the logical [i, j] operations has to be true for the result to be 1. Otherwise, the result is 0.

Example

The following example produces a result of 1 because the [1, 2] element of matrix *m* (= =2) is less than the [1, 2] element of matrix *m2* (= =5). All of the other elements do not satisfy the comparison. If the [1, 2] element of matrix *m* was changed to 6, for example, the result would be 0.

```
data _null_;
   dcl double a[3,3];
   dcl double b[3,3];
   dcl double r;

   method run();
      dcl package matrix m;
      dcl package matrix m2;

      a := (1,2,10,4,7,11,15,9,12);
      b := (1,5,9,2,6,10,3,7,11);

      m = _new_ matrix(a, 3, 3);
      m2 = _new_ matrix(b, 3, 3);

      r = m.any_lt(m2);
      put r=;
   end;
enddata;
run;
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_LT Method" on page 1685
- "ANY_GT Method" on page 1694
- "ANY_LE Method" on page 1695

ANY_NE Method

Produces a scalar result in an ANY_NE (ANY not-equal-to) comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_NE**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY not-equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY not-equal-to comparison.

Details

The ANY_NE relational operation produces a scalar result that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_NE operation is similar to the ANY_NE operation except that all of the logical [i, j] operations have to be true for the result to be 1. Otherwise, the result is 0.

Example

```
x=m1.any_ne(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_NE Method" on page 1687
- "ANY_EQ Method" on page 1692

ANY_OR Method

Produces a scalar result in an ANY_OR comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**ANY_OR**(*package-2*);

Required Arguments

x
specifies the scalar result.

package-1
specifies an instance of the first matrix package variable that is used in the ANY OR comparison.

package-2
specifies an instance of the second matrix package variable that is used in the ANY OR comparison.

Details

The ANY_OR logical operation produces a scalar result that indicates whether an $[i, j]^{th}$ element of the first matrix satisfies the comparison with the $[i, j]^{th}$ element of the second matrix. The scalar result is 0 or 1. If any of the [i, j] element comparisons is true, then the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Comparisons

The ALL_OR operation is similar to the ANY_OR operation except that all of the logical [i, j] operations have to be true for the result to be 1. Otherwise, the result is 0.

Example

```
x=m1.any_or(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_OR Method" on page 1688
- "ANY_AND Method" on page 1690
- "OR Method" on page 1741

COLS Method

Returns the number of columns in the specified matrix.

Restriction: This method is not supported on the CAS server.

Syntax

variable-name=*package*.**COLS**( );

Required Arguments

variable-name
specifies the name of a variable that contains the number of columns after the method is complete.

package
specifies an instance of the matrix package variable.

Example

See the example in the ROWS method on page 1745.

See Also

- "Using the MATRIX Package" in *SAS DS2 Programmer's Guide*

Methods:

- "ROWS Method" on page 1744

COPY Method

Copies one matrix to another.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package*.**COPY**( );

Required Arguments

r
specifies the matrix that is automatically created by the COPY method.

package
specifies an instance of the matrix package variable.

Details

The COPY method copies a matrix into a new matrix.

Example

This example creates a new copy of a 3x4 matrix.

```
data _null_;
   dcl double a[3,4];
   dcl double b[3,4];

   method run();
      dcl package matrix m;
      dcl package matrix r;
      dcl double i j;

      a := (1,2,3,4,5,6,7,8,9,10,11,12);

      m = _new_ matrix(a, 3, 4);
      r = m.copy();
      r.toarray(b);
```

```
      do i = 1 to 3;
         do j = 1 to 4;
            put b[i,j];
         end;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
1
2
3
4
5
6
7
8
9
10
11
12
```

See Also

"Using the MATRIX Package" in *SAS DS2 Programmer's Guide*

DECLARE PACKAGE Statement: Matrix Package

Creates an instance of a MATRIX package.

Category: Local

Restriction: This method is not supported on the CAS server.

Syntax

DECLARE PACKAGE MATRIX *variable* ([*row-dimension*, *column-dimension*]);

Arguments

variable
specifies a name that can reference an instance of the matrix package.

row-dimension
specifies the number of rows in the matrix instance.

column-dimension
specifies the number of columns in the matrix instance.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

The matrix package provides a DS2-level implementation of SAS/IML functionality. The matrix package is predefined for DS2 programs.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

A matrix package is created by declaring and instantiating the package using the DECLARE PACKAGE statement.

This example creates an empty 2x2 matrix, and stores the instance in the variable *m*.

```
declare package matrix m(2, 2);
```

A matrix must be initialized before it can be used, and initialization is done in the code stream, not in the declarations. You can use the following actions to load data into a matrix instance.

- _NEW_ operator to load an array
- IN method to load an array
- SET statement to load external data

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Declaring and Instantiating a MATRIX Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Methods:

- "IN Method" on page 1724

Operators:

- "_NEW_ Operator: Matrix Package" on page 1734

Statements:

- "SET Statement" on page 1305

DELETE Method: Matrix Package

Deletes a matrix package.

Restriction: This method is not supported on the CAS server.

Note: The DELETE method is not required. When no matrix package variables reference a matrix package instance, the package instance is automatically deleted.

Syntax

package.**DELETE**( );

Required Argument

package
specifies an instance of the matrix package variable.

Details

When you no longer need the matrix package, delete it by using the DELETE method. If you attempt to use a matrix package after you delete it, an error will be written to the log.

See Also

"Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

DET Method

Computes the determinant of a square matrix.

Restriction: This method is not supported on the CAS server.

Syntax

d=*package*.**DET**( );

Required Arguments

d
specifies the matrix that is automatically created by the DET method.

package
specifies an instance of the matrix package variable.

Details

The input matrix for a determinant must be square. Otherwise, you receive a run-time error. The output from the DET method is a real or complex number that is called the determinant.

Example

The following example computes a determinant for a 3x3 matrix.

```
data _null_;
   dcl double a[3,3];

   method run();
      dcl package matrix m;
      dcl double d;

      a := (1,3,2,5,4,6,9,8,9);
      m = _new_ matrix(a, 3, 3);
      d = m.det();
      put d=;
   end;
enddata;
run;
```

The following line is written to the SAS log.

```
d=23
```

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

EDIV Method

Performs an elementwise scalar division.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**EDIV**(*package-2*);

Required Arguments

x
specifies the matrix that is automatically created by the EDIV method.

package-1
specifies an instance of the first matrix package variable that is used in the elementwise division operation.

package-2
specifies an instance of the second matrix package variable that is used in the elementwise division operation.

Details

The EDIV method enables you to apply the elementwise scalar division of one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EDIV method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

```
x=m1.ediv(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "EMULT Method" on page 1714

EMAX Method

Performs an elementwise comparison of two matrices and returns the largest elements.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**EMAX**(*package-2*);

Required Arguments

x
specifies the matrix that is automatically created by the EMAX method.

package-1
specifies an instance of the first matrix package variable that is used in the elementwise maximum operation.

package-2
specifies an instance of the second matrix package variable that is used in the elementwise maximum operation.

Details

The EMAX method enables you to apply an elementwise maximum value comparison to one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EMAX method produces a result matrix from the element-by-element operations on the two argument matrices.

Examples

Example 1: Comparing Maximum Values Using a 1x1 Matrix

The following example creates a matrix that contains the maximum value from two 2x2 matrices.

```
data _null_;
   dcl double a[2,2];
   dcl double b[2,2];
   dcl double f[2,2];

   method run();
      dcl package matrix m;
      dcl package matrix m1;
        dcl package matrix r;
        dcl double i j;

      a := (2, 2, 3, 4);
        b := (4, 5, 1, 0);

      m = _new_ matrix(a, 2, 2);
        m1 = _new_ matrix(b, 2, 2);

      r = m.emax(m1);
        r.toarray(f);

         do i = 1 to 2;
         do j = 1 to 2;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
4  5
3  4
```

Example 2: Vector Operation on a Matrix

In this example, the maximum operator is applied to all the rows of matrix **m** by using the matrix **m1** as a row.

```
data _null_;
   method init();
      dcl package matrix m;
      dcl package matrix m1;
      dcl package matrix r;
      dcl double i j;
      dcl double a[4];
      dcl double b[2];
      dcl double f[2,2];
```

```
      a := (2, 2, 3, 4);
      b := (1, 5);
      m = _new_ matrix(a, 2, 2);
      m1 = _new_ matrix(b, 1, 2);
      r = m.emax(m1);
      r.toarray(f);
      do i = 1 to 2;
         do j = 1 to 2;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
2
5
3
5
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "EMIN Method" on page 1710

EMIN Method

Performs an elementwise comparison of two matrices and returns the smallest elements.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**EMIN**(*package-2*);

Required Arguments

x
specifies the matrix that is automatically created by the EMIN method.

package-1
specifies an instance of the first matrix package variable that is used in the elementwise minimum operation.

package-2
specifies an instance of the second matrix package variable that is used in the elementwise minimum operation.

Details

The EMIN method enables you to apply an elementwise minimum value comparison of one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EMIN method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example creates a matrix that contains the maximum value from two 2x2 matrices.

```
data _null_;
   dcl double a[2,2];
   dcl double b[2,2];
   dcl double f[2,2];

   method run();
      dcl package matrix m;
      dcl package matrix m1;
         dcl package matrix r;
         dcl double i j;

      a := (2, 2, 3, 4);
        b := (4, 5, 1, 0);

      m = _new_ matrix(a, 2, 2);
        m1 = _new_ matrix(b, 2, 2);

      r = m.emin(m1);
         r.toarray(f);

        do i = 1 to 2;
         do j = 1 to 2;
            put f[i,j];
         end;
      end;
   end;
```

```
enddata;
run;
```

The resulting matrix has the following values.

```
2  2
1  0
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "EMAX Method" on page 1708

EMOD Method

Returns the remainder of the division of elements of the first matrix by elements of the second matrix in an elementwise scalar operation.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**EMOD**(*package-2*);

Required Arguments

x
specifies the matrix that is automatically created by the EMOD method.

package-1
specifies an instance of the first matrix package variable that is used in the elementwise MOD comparison.

package-2
specifies an instance of the second matrix package variable that is used in the elementwise MOD comparison.

Details

The EMOD elementwise operation enables you to find the remainder of a division operation on one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EMOD method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example divides the elements in matrix, **m**, by the elements in matrix, **m2**. The EMOD method is used to return the matrix of remainders, **f**.

```
data _null_;
   dcl double a[2,2];
   dcl double b[2,2];
   dcl double f[2,2];

   method run();
      dcl package matrix m;
      dcl package matrix m1;
        dcl package matrix r;
        dcl double i j;

      a := (125, 17, 39, 40);
        b := (40, 5, 12, 8);

      m = _new_ matrix(a, 2, 2);
        m1 = _new_ matrix(b, 2, 2);

      r = m.emod(m1);
        r.toarray(f);

        do i = 1 to 2;
         do j = 1 to 2;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
5  2
3  0
```

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

EMULT Method

Performs an elementwise scalar multiplication.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**EMULT**(*package-2*);

Required Arguments

r
specifies the matrix that is automatically created by the EMULT method.

package-1
specifies an instance of the first matrix package variable that is used in the elementwise multiplication operation.

package-2
specifies an instance of the second matrix package variable that is used in the elementwise multiplication operation.

Details

The EMULT method enables you to apply the elementwise scalar multiplication on one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EMULT method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example shows how to perform an elementwise scalar multiplication. Each element of matrix *c* is multiplied by 2.

```
data _null_;
```

```
   dcl double c[3,3];
   dcl double d[1,1];
   dcl double f[3,3];

   method run();
      dcl package matrix m3;
      dcl package matrix m4;
      dcl package matrix r;
        dcl double i j;

      c := (-0, 0, -1, 1, -2.2, 2.2, -3.3, 4.4, 5.5);
      d := (2);

      m3 = _new_ matrix(c, 3, 3);
      m4 = _new_ matrix(d, 1, 1);

      r = m3.emult(m4);
      r.toarray(f);

      do i = 1 to 3;
         do j = 1 to 3;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
0     0     -2
2    -4.4   4.4
-6.6  8.8   11
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "EDIV Method" on page 1706

EPOW Method

Raises a number to a specified power in an elementwise operation.

Restriction: This method is not supported on the CAS server.

Syntax

x=*package-1*.**EPOW**(*package-2*);

Required Arguments

x
specifies the matrix that is automatically created by the EPOW method.

package-1
specifies an instance of the first matrix package variable that is used in the elementwise operation.

package-2
specifies an instance of the second matrix package variable that is used in the elementwise operation.

Details

The EPOW elementwise operation enables you to raise a value exponentially in one matrix using another matrix. The second matrix can be any of the following:

- a matrix with the same dimensions as
- a vector whose row dimension matches the row dimension of the first matrix
- is a vector whose column dimension matches the column dimension of the first matrix
- a 1x1 matrix effectively allowing a scalar operation on each [i,j] element

The EPOW method produces a result matrix from the element-by-element operations on the two argument matrices.

Example

The following example raises each element of matrix *c* to a power of 2.

```
data _null_;
   dcl double c[3,3];
   dcl double d[1,1];
   dcl double f[3,3];

   method run();
      dcl package matrix m3;
      dcl package matrix m4;
      dcl package matrix r;
        dcl double i j;

      c := (0, 15, 1.7, 13, -2.2, 10, -3.3, 7, 2);
      d := (2);
```

```
      m3 = _new_ matrix(c, 3, 3);
      m4 = _new_ matrix(d, 1, 1);

      r = m3.epow(m4);
      r.toarray(f);

      do i = 1 to 3;
         do j = 1 to 3;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
0     225    2.89
169   4.84   100
10.89 49   4
```

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

EQ Method

Produces a scalar result in an equal-to comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**EQ**(*package-2*);

Required Arguments

r
specifies a matrix that contains the results of an equal-to comparison between the values of two matrices.

package-1
specifies an instance of the first matrix package variable that is used in the equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the equal-to comparison.

Details

The EQ relational operation produces a matrix that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The result is 0 or 1. If the [i, j] elements are equal, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Example

The following example compares the elements in two matrices for equality. Note that missing values are compared.

```
data _null_;
   dcl double a[2,2];
   dcl double b[2,2];
   dcl double f[2,2];

   method run();
      dcl package matrix m;
      dcl package matrix m1;
        dcl package matrix r;
        dcl double i j;

      a := (2, 2, 3, .);
        b := (4, 5, 1, .);

      m = _new_ matrix(a, 2, 2);
        m1 = _new_ matrix(b, 2, 2);

      r = m.eq(m1);
        r.toarray(f);

        do i = 1 to 2;
         do j = 1 to 2;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
0   0
0   1
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "GE Method" on page 1721
- "LE Method" on page 1729
- "NE Method" on page 1739

EXP Method

Returns a matrix that contains an exponential value for each value in the input matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package*.**EXP**( );

Required Arguments

r
specifies the matrix that is automatically created by the EXP method.

package
specifies an instance of matrix package variable.

Details

The EXP method creates a matrix that contains each element of the input matrix raised to the eth power.

Example

The following example raises each element of a matrix to the eth power.

```
data _null_;
   dcl double a[3, 3];
   dcl double c[3, 3];

   method run();
      dcl package matrix m;
```

```
      dcl package matrix r;
        dcl double i j;

      a := (1, 2, 3, 1, 2, 3, 1, 2, 3);

      m=_new_ matrix(a, 3, 3);
      r=m.exp();
      r.toarray(c);

      do i=1 to 3;
         do j=1 to 3;
            put c[i, j];
         end;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
2.71828182845904
7.38905609893065
20.0855369231876
2.71828182845904
7.38905609893065
20.0855369231876
2.71828182845904
7.38905609893065
20.0855369231876
```

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

FLOOR Method

Returns a matrix that contains the integer part of each value in the input matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*matrix-package*.**FLOOR**( );

Required Arguments

r

specifies a matrix that contains the integer part of each value in the input matrix.

matrix-package
specifies an instance of matrix package variable.

Example

The following example creates a matrix that contains the integer part of input matrix, **m**.

```
data _null_;
   dcl double a[2, 2];
   dcl double c[2, 2];

   method run();
      dcl package matrix m;
      dcl package matrix r;
      dcl double i j;

      a := (1323.43, -.72, 3.38, 45);

      m=_new_ matrix(a, 2, 2);
      r=m.floor();
      r.toarray(c);

      do i=1 to 2;
         do j=1 to 2;
            put c[i, j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
1323    0
3       45
```

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

GE Method

Produces a scalar result in a greater-than-or-equal-to comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**GE**(*package-2*);

Required Arguments

r
specifies a matrix that contains the results of a greater-than-or-equal-to comparison between the values of two matrices.

package-1
specifies an instance of the first matrix package variable that is used in the greater-than-or-equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the greater-than-or-equal-to comparison.

Details

The GE relational operation produces a matrix that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The result is 0 or 1. If the [i, j] element greater-than-or-equal-to comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Example

```
r=m1.ge(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "GT Method" on page 1723
- "LE Method" on page 1729

GT Method

Produces a scalar result in a greater-than comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**GT**(*package-2*);

Required Arguments

r
specifies a matrix that contains the results of a greater-than comparison between the values in two matrices.

package-1
specifies an instance of the first matrix package variable that is used in the greater-than comparison.

package-2
specifies an instance of the first matrix package variable that is used in the greater-than comparison.

Details

The GT relational operation produces a matrix that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The result is 0 or 1. If the [i, j] element greater-than comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Example

```
r=m1.gt(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "GE Method" on page 1721
- "LT Method" on page 1732

IN Method

Loads an array into a matrix.

Alias: LOAD

Restriction: This method is not supported on the CAS server.

Syntax

package.**IN**(*array-name*);

Required Arguments

package
specifies an instance of the matrix package variable.

array-name
specifies an array that is used in loading the matrix.

Restriction: The array dimensions must match the matrix dimensions. Otherwise, an error occurs.

Tip: You can use variable arrays.

See: "DS2 Arrays" in *SAS DS2 Programmer's Guide*

Details

The IN method loads an array into a matrix. This process might be useful if an array `a` can change as the program executes and you want to repeatedly reset the values of matrix `m`. Here is an example:

```
dcl double a[3, 3];
dcl package matrix m;

m=_new_ matrix(3, 3);
m.in(a);
```

You can use variable arrays to load data into a matrix. Here is an example:

```
vararray double va[3,3];
  dcl package matrix m;
```

```
  m.in(va);
```

You can also use variable arrays to input and output data using the SET and OUT statements.

Examples

Example 1: Loading and Writing Data

This example reads data from a data set in matrix form, finds the matrix inverse, and writes the result matrix to an output table. The example uses the IN and OUT methods. The IN method loads data from a variable array into a matrix. The OUT method writes the data in the matrix to a variable array. The SET and OUPUT statements are used in the program to read data from an array and write the results to an output array.

The IN and OUT methods are overloaded to accept an integer argument that tells which row of the matrix to load. For the IN method, the variable array row data is read into the matrix. For the OUT method, the matrix row data is written to the variable array.

```
/* DATA step to create an array of data */
data x;
   array a[4];

   /* Create a 4x4 matrix. */
   a1 = 1; a2 = 5; a3 = 2; a4 = 3;
   output;

   a1 = 3; a2 = 3; a3 = 1; a4 = 7;
   output;

   a1 = 2; a2 = 3; a3 = 8; a4 = 9;
   output;

   a1 = 3; a2 = 6; a3 = 7; a4 = 4;
   output;
run;

proc ds2;
data inv/overwrite=yes;

   /* global declarations */
   vararray double v[4];
   vararray double a[4];
   keep v1 v2 v3;

   dcl package matrix m;
   dcl package matrix r;
   dcl package matrix inv;
   dcl double c[4, 4];
   dcl double i j;
```

```
   /* Create an empty matrix to hold the input values. */
   method init();
      m=_new_ matrix(4, 4);
      i=1;
   end;

   /* Read and initialize each row of the matrix from VARARRAY a. */
   method run();
      set x;
      m.in(a, i);
      i + 1;
   end;

   method term();
      /* Find the inverse of the matrix. */
      inv=m.inverse();

      /* Check whether it gives an identity matrix. */
      r=m.mult(inv);
      /* Write each row of inverse to the output table */
      /* using VARARRAY v.                              */
      do i=1 to 4;
         inv.out(v, i);
         output;
      end;

      /* Print the result to see if it's the identity. */
      r.toarray(c);
      do i=1 to 4;
         do j=1 to 4;
            put c[i, j];
         end;
      end;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log.

```
1
-2.7755575615628E-17
-1.1102230246251E-16
0
1.1102230246251E-16
1
0
0
1.1102230246251E-16
5.5511151231257E-17
1
0
1.6653345369377E-16
1.6653345369377E-16
-1.6653345369377E-16
1
```

These are the key calls for the program:

```
/*  Input  */
method run();
   set x;
   m.in(a, i);
   i + 1;
end;

/* Output */
   do i=1 to 4;
      inv.out(v, i);
      output;
   end;
```

Example 2: Using the IN and OUT Methods

For another example of using the IN and OUT methods, see "Example 2: Multiply Two Matrices That Are Read from External Data" on page 1738.

See Also

- "Matrix Data Input" in *SAS DS2 Programmer's Guide*

Methods:

- "OUT Method" on page 1742

Statements:

- "OUTPUT Statement" on page 1280
- "SET Statement" on page 1305

INVERSE Method

Computes the inverse of a matrix.

Restriction: This method is not supported on the CAS server.

Syntax

im=*matrix-package*.**INVERSE**( );

Required Arguments

im
specifies the matrix that is automatically created by the INVERSE method.

matrix-package
specifies an instance of matrix package variable.

Details

It is possible to perform basic matrix operations on a single matrix. The INVERSE matrix operation computes the inverse of a matrix. If the matrix is not square or is singular (not invertible), you receive a run-time error.

Example

The following example computes the inverse of a 3x3 matrix, and checks, using matrix multiplication, whether the resulting inverse produces the identity matrix.

```
data _null_;
   dcl double a[3,3];
   dcl double b[3,3];

   method run();
      dcl package matrix m im r;
      dcl double i j;

      a := (1, 2, -1, 2, 1, 0, -1, 1, 2);
      m = _new_ matrix(a, 3, 3);

      im = m.inverse();
      r = m.mult(im);
      r.toarray(b);

      do i = 1 to 3;
         do j = 1 to 3;
            put b[i,j];
         end;
      end;
   end;
enddata;
run;
```

Some values of the resulting matrix are not exactly zero. The values might be very small numbers, such as 1.1102230246251E-16. These small numbers are the exact results that the SAS/IML subsystem provides.

```
1                    5.5511151231257E-17   0
0                    1                    0
1.1102230246251E-16   -1.1102230246251E-16   1
```

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

LE Method

Produces a scalar result in a less-than-or-equal-to comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**LE**(*package-2*);

Required Arguments

r
specifies a matrix that contains the results of a less-than-or-equal-to comparison between the values of two matrices.

package-1
specifies an instance of the first matrix package variable that is used in the less-than-or-equal-to comparison.

package-2
specifies an instance of the second matrix package variable that is used in the less-than-or-equal-to comparison.

Details

The LE relational operation produces a matrix that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The result is 0 or 1. If the [i, j] element less-than-or-equal-to comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Examples

Example 1: Comparing Two Matrices Using the LE Method

The following example uses the LE method. The [i, j]th element of matrix **m** is compared with the [i, j] element of matrix **m2**, using 0 or 1 for the result entries.

```
data _null_;
   dcl double a[3,3];
   dcl double b[3,3];
```

```
   dcl double c[3,3];

   method run();
      dcl package matrix m;
      dcl package matrix m2;
      dcl package matrix r;
      dcl double i j;

      a := (1,2,3,4,5,6,7,8,9);
      b := (1,5,9,2,6,10,3,7,11);

      m = _new_ matrix(a, 3, 3);
      m2 = _new_ matrix(b, 3, 3);

      r = m.le(m2);
      r.toarray(c);

      do i = 1 to 3;
         do j = 1 to 3;
            put c[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
1   1   1
0   1   1
0   0   1
```

Example 2: Using a Scalar Matrix

The following example uses a scalar matrix with the LE method.

```
data _null_;
   dcl double a[3,3];
   dcl double b[1,1];
   dcl double c[3,3];

   method run();
      dcl package matrix m;
      dcl package matrix m2;
      dcl package matrix r;
      dcl double i j;

      a := (1,3,3,4,5,6,7,8,9);
      b := (4);

      m = _new_ matrix(a, 3, 3);
      m2 = _new_ matrix(b, 1, 1);

      r = m.le(m2);
      r.toarray(c);

      do i = 1 to 3;
```

```
            do j = 1 to 3;
                put c[i,j];
            end;
        end;
    end;
enddata;
run;
```

The resulting matrix has the following values.

```
1    1    1
1    0    0
0    0    0
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "GE Method" on page 1721
- "LT Method" on page 1732

LOG Method: Matrix Package

Returns a matrix that contains the natural logarithm for each value in the input matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package*.**LOG**( );

Required Arguments

r
specifies a matrix that is automatically created by the LOG method.

package
specifies an instance of matrix package variable.

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

LT Method

Produces a scalar result in a less-than comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*package-1*.**LT**(*package-2*);

Required Arguments

r
specifies a matrix that contains the results of a less-than comparison between the values in two matrices.

package-1
specifies an instance of the first matrix package variable that is used in the less-than comparison.

package-2
specifies an instance of the second matrix package variable that is used in the less-than comparison.

Details

The LT relational operation produces a matrix that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The result is 0 or 1. If the [i, j] element less-than comparison is true, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Examples

Example 1: Comparing Two Matrices Using the LT Method

The following example uses the LT method. The [i, j]th element of matrix **m** is compared with the [i, j] element of matrix **m2**, using 0 or 1 for the result entries.

```
data _null_;
   dcl double a[3,3];
   dcl double b[3,3];
```

```
   dcl double c[3,3];

   method run();
      dcl package matrix m;
      dcl package matrix m2;
      dcl package matrix r;
      dcl double i j;

      a := (1,2,3,4,5,6,7,8,9);
      b := (1,5,9,2,6,10,3,7,11);

      m = _new_ matrix(a, 3, 3);
      m2 = _new_ matrix(b, 3, 3);

      r = m.lt(m2);
      r.toarray(c);

      do i = 1 to 3;
         do j = 1 to 3;
            put c[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
0   1   1
0   1   1
0   0   1
```

Example 2: Using a Scalar Matrix

The following example uses a scalar matrix with the LT method.

```
data _null_;
   dcl double a[3,3];
   dcl double b[1,1];
   dcl double c[3,3];

   method run();
      dcl package matrix m;
      dcl package matrix m2;
      dcl package matrix r;
      dcl double i j;

      a := (1,3,3,4,5,6,7,8,9);
      b := (4);

      m = _new_ matrix(a, 3, 3);
      m2 = _new_ matrix(b, 1, 1);

      r = m.lt(m2);
      r.toarray(c);

      do i = 1 to 3;
```

```
            do j = 1 to 3;
                put c[i,j];
            end;
        end;
    end;
enddata;
run;
```

The resulting matrix has the following values.

```
1    1    1
0    0    0
0    0    0
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "GT Method" on page 1723
- "LE Method" on page 1729

NEW Operator: Matrix Package

Constructs an instance of a matrix package.

Restriction: This method is not supported on the CAS server.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable=**_NEW_**

MATRIX([*array-name*,] *rows*, *columns*)

Required Arguments

package-variable
specifies a name that can reference an instance of the matrix package.

rows
specifies the number of rows in the matrix.

columns
specifies the number of columns in the matrix.

Optional Argument

array-name
specifies the name of an array to load into the matrix package.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a matrix package variable using the DECLARE PACKAGE statement. After you declare a matrix package variable, use the _NEW_ operator to instantiate an instance of the matrix package and assign the new package instance to the matrix package variable.

A matrix must be initialized before it can be used, and initialization is done in the code stream, not in the declarations. To initialize a matrix, you can use the _NEW_ operator or the DECLARE statement.

A matrix can be initialized with values from a DS2 array. Here is an example:

```
method init();
   dcl double a[3, 3];
   dcl package matrix m;

   a :=(1, 2, -1, 2, 1, 0, -1, 1, 2);
   m=_new_ matrix(a, 3, 3);
end;
```

In this example, a 3x3 array is initialized with values that you specify, and is used to set up the matrix *m*. The values are read in row-major order, and the matrix that is produced has the following values:

```
 1     2    -1
 2     1     0
-1     1     2
```

You can also initialize a matrix by using a variable array, as the following example shows:

```
vararray double a [3, 3];
dcl package matrix m;

method init();
   a := (1, 2, -1, 2, 1, 0, -1, 1, 2);
   m=_new_ matrix(a, 3, 3);
end;
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Matrix Data Input" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: Matrix Package" on page 1703

MULT Method

Multiplies one matrix by another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*matrix-package-1*.**MULT**(*matrix-package-2*);

Required Arguments

r
specifies the matrix that is automatically created by the MULT method.

matrix-package-1
specifies the name of the first matrix package that is used in the multiplication operation.

matrix-package-2
specifies the name of the second matrix package that is used in the multiplication operation.

Details

The MULT method automatically creates a matrix that contains the result of matrix multiplication. The result matrix has the same number of rows as the first matrix and the same number of columns as the second matrix.

The following considerations apply when performing matrix multiplication.

- Array dimensions for the matrices that are used in multiplication operations must be compatible. Multiplication requires that the number of columns in the first matrix be equal to the number of rows in the second matrix. Otherwise, a run-time error is generated.

- If you multiply matrices that have missing values, you will receive a run-time error.

Examples

Example 1: Simple Matrix Multiplication

The following example uses the MULT method to perform a simple matrix multiplication. A 3x4 matrix, **m**, is multiplied by a 4x3 matrix, **m2**, to obtain a 3x3 result. The result is stored in matrix, **r**. The values in matrix **r** can be placed into a 3x3 array, **c**, and written to the SAS log.

```
data _null_;
   dcl double a[3,4];
   dcl double b[4,3];
   dcl double c[3,3];

   method run();
      dcl package matrix m;
      dcl package matrix m2;
      dcl package matrix r;
      dcl double i j;

      a := (1,2,3,4,5,6,7,8,9,10,11,12);
      b := (1,5,9,2,6,10,3,7,11,4,8,12);

      m = _new_ matrix(a, 3, 4);
      m2 = _new_ matrix(b, 4, 3);

      r = m.mult(m2);
      r.toarray(c);

      do i = 1 to 3;
         do j = 1 to 3;
            put c[i,j];
         end;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
30
70
110
70
174
278
110
278
446
```

Example 2: Multiply Two Matrices That Are Read from External Data

This example multiplies two matrices that are read from external data. The IN method reads the matrices and the OUT method writes the output. The IN method is an alias for the LOAD method.For more information, see the "IN Method" on page 1724 and the "OUT Method" on page 1742.

```
proc ds2;
 data x(keep =(a1 a2 a3)) y(keep=(b1 b2 b3 b4))/overwrite=yes;
   vararray double a[3];
   vararray double b[4];

   method init();
      dcl double i j;

      /* Create output for matrix a */
      do i = 1 to 4;
        do j = 1 to 3;
          a[j] = 2 * j + i;
        end;
      output x;
      end;

      /* Create output for matrix b */
      do i = 1 to 3;
        do j = 1 to 4;
          b[j] = 3 * j - 2 * i;
        end;
      output y;
      end;
   end;
enddata;
run;
quit;

proc ds2;
data z(keep =(v1 v2 v3 v4))/overwrite=yes;
   drop i;
   vararray double v[4];
   vararray double a[3];
   vararray double b[4];
   dcl package matrix ma mb;
   dcl package matrix r;

   method init();
      dcl double i;
      ma = _new_ matrix(4,3);
      mb = _new_ matrix(3,4);

      /* Read matrix a (row-by-row) */

      do i = 1 to 4;
         set x;
         ma.in(a, i);
      end;
```

```
      /* Read matrix b (row-by-row) */
      do i = 1 to 3;
         set y;
         mb.in(b, i);
      end;
   end;

   method term();
      dcl double i;
      /* Multiply matrices */
      r = ma.mult(mb);

      /* Output result */
      do i = 1 to 4;
         r.out(v, i);
         output;
      end;
   end;
enddata;
run;
quit;
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "EMULT Method" on page 1714

NE Method

Produces a scalar result in a not-equal-to comparison between elements in one matrix and elements in another matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*matrix-package-1*.**NE**(*matrix-package-2*);

Required Arguments

r

specifies a matrix that contains the results of a not-equal-to comparison between the values in two matrices.

matrix-package-1
specifies the name of the first matrix package that is used in the not-equal-to comparison.

matrix-package-2
specifies the name of the second matrix package that is used in the not-equal-to comparison.

Details

The NE relational operation produces a matrix that indicates whether an [i, j]th element of the first matrix satisfies the comparison with the [i, j]th element of the second matrix. The result is 0 or 1. If the [i, j] elements are not equal, the result is 1. Otherwise, the result is 0.

The matrix sizes must match or you can use a scalar comparison.

Example

The following example compares the elements in two matrices for inequality. Note that missing values are compared.

```
data _null_;
   dcl double a[2,2];
   dcl double b[2,2];
   dcl double f[2,2];

   method run();
      dcl package matrix m;
      dcl package matrix m1;
        dcl package matrix r;
        dcl double i j;

      a := (2, 2, 3, .);
        b := (4, 5, 1, .);

      m = _new_ matrix(a, 2, 2);
        m1 = _new_ matrix(b, 2, 2);

      r = m.ne(m1);
        r.toarray(f);

        do i = 1 to 2;
         do j = 1 to 2;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting matrix has the following values.

```
1   1
1   0
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "EQ Method" on page 1717
- "GE Method" on page 1721
- "LE Method" on page 1729

OR Method

Compares two matrices based on the OR logical operation, and returns the resulting matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*matrix-package-1*.**OR**(*matrix-package-2*);

Required Arguments

r
specifies a matrix that contains the results of an OR comparison between the values of two matrices.

matrix-package-1
specifies the name of the first matrix package that is used in the OR comparison.

matrix-package-2
specifies the name of the second matrix package that is used in the OR comparison.

Details

The OR logical operator behaves similarly to the binary relational operations (LT, LE, GE, GT, NE, and EQ). In each case, the OR logical operation is applied to the [i, j]th elements of two matrices and placed in the result matrix *r*.

Example

```
r=m1.or(m2);
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ALL_OR Method" on page 1688
- "AND Method" on page 1689
- "ANY_OR Method" on page 1700

OUT Method

Writes matrix row data to a variable array.

Restriction: This method is not supported on the CAS server.

Syntax

matrix-package.**OUT**(*array-name*);

Required Arguments

matrix-package
specifies a matrix package to be used with the OUT method.

array-name
specifies a matrix that is used in writing output.

Restriction: The array dimensions must match the matrix dimensions. Otherwise, an error occurs.

Tip: You can use variable arrays.

See: "DS2 Arrays" in *SAS DS2 Programmer's Guide*

Details

The OUT method writes matrix row data to a variable array. Variable arrays can be used to input and output data using an existing DS2 table and output statements. For example, you can read data from a table in matrix form, find the matrix inverse, and write the result matrix to an output table. See the example below.

The OUT method can be overloaded to accept an integer argument that tells which row of a matrix to write to a variable array.

The synonym for the OUT method is the TOVARARRAY method. The following two statements are equivalent:

```
r.tovararray(va, i);
r.out(va, i);
```

The OUT method, which writes data, is often used with the IN method. The IN and OUT methods are overloaded to accept an integer argument that tells which row of a matrix to load. For the IN method, the variable array row data is read into the matrix. For the OUT method, the matrix row data is written to the variable array. In this way, matrices can be loaded and unloaded a row at a time from and to external data storage using variable arrays.

Examples

Example 1

This example writes each row of an inverse to an output table using the variable array *v*.

```
do i=1 to 4;
   inv.out(v, i);
   output;
end;
```

Example 2

Similar to the standard IN method, a complete matrix can also be written to a variable array:

```
vararray double va(3, 3);
dcl package matrix r;

r=_new_ matrix(3, 3);
r.out(va);
```

In this example, a matrix does not need to be written row-by-row. The entire matrix can be written to the variable array. You can use this method in the case where the DS2 output statement (which is row-based) is not being used.

Example 3: Loading and Writing Data

For another example of using the IN and OUT methods, see "Example 2: Multiply Two Matrices That Are Read from External Data" on page 1738.

Example 4: Writing the Entire Matrix to the Array

Similar to the standard LOAD method, a complete matrix can also be written to a variable array:

```
vararray double va[3, 3];
dcl package matrix r;
  r=_new_ matrix(3, 3);
r.out(va);
```

This example shows that matrix data does not need to be written row-by-row. You can write the entire matrix to the array. You could use this technique in a case where the DS2 OUTPUT statement (which is row-based) is not used.

See Also

- "Matrix Data Output" in *SAS DS2 Programmer's Guide*

Methods:

- "IN Method" on page 1724
- "OUT Method" on page 1742

Statements:

- "OUTPUT Statement" on page 1280
- "SET Statement" on page 1305

ROWS Method

Returns the number of rows in the specified matrix.

Restriction: This method is not supported on the CAS server.

Syntax

variable-name=*matrix-package*.**ROWS**( );

Required Arguments

variable-name
specifies the name of a variable that contains the number of rows after the method is complete.

matrix-package
specifies a matrix package to use with the ROWS method.

Example

The following example returns the number of rows and columns in the matrix, **m**.

```
data _null_;
   dcl double a[2, 3];

   method run();
      dcl package matrix m;
      dcl double mr mc;

      a := (1,2,3,4,5,6);

      m=_new_ matrix(a, 2, 3);
      mr=m.rows();
      mc=m.cols();
      put mr=;
      put mc=;

   end;
run;
```

The following lines are written to the SAS log.

```
mr=2
mc=3
```

See Also

- "Using the MATRIX Package" in *SAS DS2 Programmer's Guide*

Methods:

- "COLS Method" on page 1701

SQRT Method

Returns a matrix that contains the square root of each value in the input matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*matrix-package*.**SQRT**( );

Required Arguments

r
specifies a matrix that contains the square root of each value in the input matrix.

matrix-package
specifies a matrix package to use with the SQRT method.

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

SUB Method

Subtracts one matrix from another.

Restriction: This method is not supported on the CAS server.

Syntax

r=*matrix-package-1*.**SUB**(*matrix-package-2*);

Required Arguments

r
specifies the matrix that is automatically created by the SUB method.

matrix-package-1
specifies the name of the first matrix package that is used in the subtraction operation.

matrix-package-2
specifies the name of the second matrix package that is used in the subtraction operation.

Details

The matrix dimensions for the SUB method must be the same size in order for the matrix subtraction to take place. Each [i, j] element in the first matrix is subtracted from its corresponding [i, j] element in the second matrix.

If you subtract matrices that have missing values, you do not receive an error.

It is also possible to perform scalar subtraction by using a 1x1 matrix.

Example

Here is an example of using the SUB method to subtract two matrices.

```
data _null_;
   dcl double c[3,3];
   dcl double d[1,1];
   dcl double f[3,3];
   method run();
      dcl package matrix m3;
      dcl package matrix m4;
      dcl package matrix r;
      dcl double i j;

      c := (-0, 0, -1, 1, -2.2, 2.2, -3.3, 4.4, 5.5);
      d := (1);

      m3 = _new_ matrix(c, 3, 3);
      m4 = _new_ matrix(d, 1, 1);

      r = m3.sub(m4);
      r.toarray(f);

      do i = 1 to 3;
         do j = 1 to 3;
            put f[i,j];
         end;
      end;
   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
-1
-1
-2
0
-3.2
1.2
-4.3
3.4
4.5
```

See Also

- "Matrix Operations" in *SAS DS2 Programmer's Guide*

Methods:

- "ADD Method: Matrix Package" on page 1677

TOARRAY Method

Moves the values from a matrix package into a DS2 array.

Restriction: This method is not supported on the CAS server.

Syntax

array-name.**TOARRAY**(*matrix-package*);

Required Arguments

array-name
specifies the name of an array to which matrix values are moved.

matrix-package
specifies the matrix package from which values are moved into an array.

Details

You use the TOARRAY method to move values from a matrix to a DS2 array. The array can then be used directly in a DS2 program.

Note: You can also move values to a variable array by using the TOVARARRAY method.

Example

This example moves the values from a matrix into a DS2 array.

```
data _null_;
   dcl double a[3,3];
   dcl double c[3,3];

   method init();
```

```
      dcl double i j;
      dcl package matrix m;

      a := (1,2,3,4,5,6,7,8,9);

      m=_new_ matrix(a,3,3);
      m.toarray(c);

      do i=1 to 3;
         do j=1 to 3;
            put c[i,j];
         end;
      end;
   end;
enddata;
run;
```

The resulting array has the following values.

```
1  2  3
4  5  6
7  8  9
```

See Also

- "Matrix Data Output" in *SAS DS2 Programmer's Guide*
- "DS2 Arrays" in *SAS DS2 Programmer's Guide*

Methods:

- "TOVARARRAY Method" on page 1749

TOVARARRAY Method

Moves the values from a matrix package into a variable array.

Restriction: This method is not supported on the CAS server.

Syntax

variable-array-name.**TOVARARRAY**(*matrix-package*);

Required Arguments

variable-array-name
specifies the name of a variable array to which matrix values are moved.

matrix-package
specifies the matrix package from which values are moved into a variable array.

Details

You use the TOVARARRAY method to move values from a matrix to a DS2 variable array. The variable array can then be used directly in a DS2 program.

Note: You can also move values to an array by using the TOARRAY method.

Example

This example shows how to move the values from a matrix into a variable array.

```
data _null_;
   dcl double a[3, 3];
   vararray double c[3, 3];

   method run();
      dcl package matrix m;
      dcl double i j;

      a := (1,2,3,4,5,6,7,8,9);

      m=_new_ matrix(a, 3, 3);
      m.tovararray(c);

      do i=1 to 3;
         do j=1 to 3;
            put c[i, j];
         end;
      end;
   end;
run;
```

The resulting array has the following values.

```
1  2  3
4  5  6
7  8  9
```

See Also

- "Matrix Data Output" in *SAS DS2 Programmer's Guide*
- "Variable Arrays" in *SAS DS2 Programmer's Guide*

Methods:

■ “TOARRAY Method” on page 1748

TRANS Method

Returns a matrix that transposes the rows and columns of the input matrix.

Restriction: This method is not supported on the CAS server.

Syntax

r=*matrix-package*.**TRANS**( );

Required Arguments

r
specifies the matrix that contains the transposition of the input matrix.

matrix-package
specifies a matrix package to use with the TRANS method.

Details

The TRANS method exchanges the rows and columns of a given matrix producing the transpose of matrix. If $\mathbf{v}$ is the value in the ith row and j^{th} column of matrix, then the transpose of matrix contains $\mathbf{v}$ in the jth row and i^{th} column. If matrix contains n rows and *p* columns, the transpose has *p* rows and *n* columns.

Example

The following example transposes a 3x4 matrix to produce a 4x3 result matrix.

```
data _null_;
   dcl double a[3, 4];
   dcl double b[4, 3];

   method run();
      dcl package matrix m;
      dcl package matrix r;
        dcl double i j;

      a := (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);

      m=_new_ matrix(a, 3, 4);
      r=m.trans();
```

```
      r.toarray(b);

      do i=1 to 4;
         do j=1 to 3;
            put b[i, j];
         end;
      end;
   end;
enddata;
run;
```

The input matrix is shown here.

```
1   2  3
4   5  6
7   8  9
10  11  12
```

The transposed, result matrix is shown here.

```
1  5   9
2  6  10
3  7  11
4  8  12
```

See Also

"Matrix Operations" in *SAS DS2 Programmer's Guide*

20

DS2 PCRXFIND Package Methods, Operators, and Statements

Dictionary 1753
DECLARE PACKAGE Statement: PCRXFIND Package 1753
GETGROUP Method 1755
GETGROUPEND Method 1757
GETGROUPLENGTH Method 1758
GETGROUPSTART Method 1759
GETMATCH Method 1760
GETMATCHEND Method 1761
GETMATCHLENGTH Method 1763
GETMATCHSTART Method 1764
MATCH Method 1765
PARSE Method: PCRXFIND Package 1766
NEW Operator: PCRXFIND Package 1768

Dictionary

DECLARE PACKAGE Statement: PCRXFIND Package

Creates a package variable and gives you the option of creating an instance of the PCRXFIND package.

Category: Local

Restriction: This package is not supported on z/OS or 32-bit Windows operating systems.

Tip: The PACKAGE statement is not required for a PCRXFIND package.

Syntax

DECLARE PACKAGE PCRXFIND *variable* ([*regular-expression*]);

Arguments

variable
specifies a name that can reference an instance of the PCRXFIND package.

regular-expression
specifies a regular expression in the form of */expression/flags*. These regular expression *flags* are supported:

i
case insensitive – Allows a case-insensitive expression match within a string.

m
multiple lines – Treats the string as a set of multiple lines. Allows the first character match or last character match to occur next to embedded newline characters.

n
non-capturing – Disables numbered capturing parenthesis.

s
single line – Treats a string as a single long line. Allows the match of a single character (.) to include a newline character.

x
extends your pattern by allowing whitespace characters (tab, newline, carriage return, and form feed characters) and comments in your match.

Data type VARCHAR

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use a PCRXFIND package to search for a particular string of text, a *regular-expression*, in another string of text. The PCRXFIND package is predefined for DS2 programs.

You declare a PCRXFIND package by using the DECLARE PACKAGE statement. After you declare the new PCRXFIND package, you can parse a Perl-compatible regular expression. (Optional) This expression can be passed in when the package

is declared. The loaded expression can then be used to perform match operations by providing the indexes of the beginning and ending of the match, as well as indexes related to each parenthetical capture group within the previously parsed regular expression.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement. Each package instance represents a completely separate copy of the package.

There are three ways to construct an instance of a PCRXFIND package.

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package pcrxfind finder(<'regular-expression'>);
```

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package pcrxfind finder;
finder = _new_ pcrxfind(<'regular-expression'>);
```

- Use the DECLARE PACKAGE statement without an expression:

```
declare package pcrxfind finder;
finder = _new_ pcrxfind();
```

See Also

Operators:

- "_NEW_ Operator: PCRXFIND Package" on page 1768

GETGROUP Method

Stores the text represented by the given capture group based on the previous match into the user-supplied resultString.

Requirement: You must have a successful match (returns an index) using the MATCH method before using the GETGROUP method. Using the GETGROUP method without a successful match causes an error and stops program execution.

Syntax

returnCode=*package*.**GETGROUP** (IN_OUT *resultString*, *groupNumber*);

Required Arguments

package
specifies an instance of the PCRXFIND package variable.

groupNumber
specifies the number of the capture group whose text is stored in the *resultString*.

Data type INTEGER

resultString
specifies the string to store the contents of the group in.

Data type VARCHAR

Returned Values

returnCode
specifies a status indicator.

0
indicates successful execution.

–1
indicates the specified capture group was not involved in the previous match.

Data type INTEGER

Details

The GETGROUP method stores the text of the specified capture group into the given string.

See Also

Methods:

- “GETGROUPEND Method” on page 1757
- “GETGROUPLENGTH Method” on page 1758
- “GETGROUPSTART Method” on page 1759

GETGROUPEND Method

Returns the ending index of the given capture group.

Requirement: You must have a successful match using the MATCH method before using the GETGROUPEND method. Using the GETGROUPEND method without a successful match causes an error and stops program execution.

Syntax

ending-index=*package*.**GETGROUPEND** (*groupNumber*) returns int;

Required Arguments

groupNumber
specifies the number of the capture group.

Data type INTEGER

package
specifies an instance of the PCRXFIND package variable.

Returned Values

ending-index
returns the next position in the string after the given capture group.

1 or greater
specifies the ending position.

-1
indicates the specified capture group was not involved in the previous match.

Data type INTEGER

Details

The GETGROUPEND method returns the position in the string after the match or capture group. It adds the length to the starting index.

```
int endIndex = rx.getGroupStart(n) + rx.getGroupLength(n);
```

See Also

Methods:

- “GETGROUP Method” on page 1755
- “GETGROUPLENGTH Method” on page 1758
- “GETGROUPSTART Method” on page 1759

GETGROUPLENGTH Method

Returns the length of the given capture group's text in characters.

Requirement: You must have a successful match using the MATCH method before using the GETGROUPLENGTH method. Using the GETGROUPLENGTH method without a successful match causes an error and stops program execution.

Syntax

length=*package*.**GETGROUPLENGTH** (*groupNumber*);

Required Arguments

package
specifies an instance of the PCRXFIND package variable.

groupNumber
specifies the number of the capture group.

Data type INTEGER

Returned Values

length
specifies the variable to hold the number of characters of the given capture group's text.

1 or greater
specifies the length of the text for the given capture group.

-1
indicates that the specified capture group was not involved in the previous match.

Data type INTEGER

Example

For an example, see "Using the PCRXFIND Package" in *SAS DS2 Programmer's Guide*.

See Also

Methods:

- "GETGROUP Method" on page 1755
- "GETGROUPEND Method" on page 1757
- "GETGROUPSTART Method" on page 1759

GETGROUPSTART Method

Returns the starting index of the given capture group.

Requirement: You must have a successful match using the MATCH method before using the GETGROUPSTART method. Using the GETGROUPSTART method without a successful match causes an error and stops program execution.

Syntax

starting-index=*package*.**GETGROUPSTART** (*groupNumber*);

Required Arguments

groupNumber
specifies the number of the capture group.

Data type INTEGER

package
specifies an instance of the PCRXFIND package variable.

Returned Values

starting-index
specifies the starting index for the previous match.

1 or greater
specifies the starting position.

-1
indicates that the specified capture group was not involved in the previous match.

Data type INTEGER

See Also

Methods:

- "GETGROUP Method" on page 1755
- "GETGROUPLENGTH Method" on page 1758
- "GETGROUPEND Method" on page 1757

GETMATCH Method

Places the text of the previous match into the result string.

Requirement: You must have a successful match using the MATCH method before using the GETMATCH method. Using the GETMATCH method without a successful match causes an error and stops program execution.

Syntax

returnCode=*package*.**GETMATCH** (IN_OUT *resultString*);

Required Arguments

package
specifies an instance of the PCRXFIND package variable.

returnCode
specifies a status indicator.

0
indicates successful execution.

Data type Integer

resultString
specifies the string to store the contents of the match in.

Data type VARCHAR

Details

The GETMATCH method stores the contents of the previous match into the given string.

See Also

Methods:

- “GETMATCHEND Method” on page 1761
- “GETMATCHLENGTH Method” on page 1763
- “GETMATCHSTART Method” on page 1764

GETMATCHEND Method

Returns the next position after the end of the text from the previous match.

Requirement: You must have a successful match using the MATCH method before using the GETMATCHEND method. Using the GETMATCHEND method without a successful match causes an error and stops program execution.

Syntax

ending-index=*package*.**GETMATCHEND**();

Required Arguments

ending-index
specifies the position after the last character of the matched string.

2 or greater
Two or greater is returned for the first value after the matched string.

-1
indicates that the specified capture group was not involved in the previous match.

Data type INTEGER

package
specifies an instance of the PCRXFIND package variable.

Details

Use the GETMATCHEND method to determine the first position after the matched string within the subject string. This enables you to easily start your next search from the point where the matched string ends.

Example

This example searches a line of text using a regular expression. The GETMATCHEND method returns the index of the next position after the end of the matched string.

```
data regex_test/overwrite=yes;
   dcl char(20) text regex;
   dcl double parse_rc match_rc next_search_start;
   method run();
      dcl package pcrxfind prxf();
      text='quick brown fox';
      regex='/brown/';
      parse_rc=prxf.parse(regex);
      if parse_rc=0 then do;
         match_rc=prxf.match(text);
         if match_rc>0 then do;
            next_search_start=prxf.getmatchend();
            put next_search_start=;
         end;
      end;
   end;
enddata;
run;
```

The following line is written to the SAS log.

```
next_search_start=12
```

See Also

Methods:

- “GETMATCH Method” on page 1760
- “GETMATCHLENGTH Method” on page 1763
- “GETMATCHSTART Method” on page 1764

GETMATCHLENGTH Method

Returns the number of characters of the previous match.

Requirement: You must have a successful match using the MATCH method before using the GETMATCHLENGTH method. Using the GETMATCHLENGTH method without a successful match causes an error and stops program execution.

Syntax

length=*package*.**GETMATCHLENGTH** ();

Required Arguments

package
specifies an instance of the PCRXFIND package variable.

length
specifies the number of characters of the matched string. Length can be one of these values:

1 or greater
the number of characters of the matched string.

-1
indicates that the specified capture group was not involved in the previous match.

Data type Integer

Details

The GETMATCHLENGTH method

See Also

Methods:

- “GETMATCH Method” on page 1760
- “GETMATCHEND Method” on page 1761
- “GETMATCHSTART Method” on page 1764

GETMATCHSTART Method

Returns the starting index of the previous match operation.

Requirement: You must have a successful match using the MATCH method before using the GETMATCHSTART method. Using the GETMATCHSTART method without a successful match causes an error and stops program execution.

Syntax

starting-index=*package*.**GETMATCHSTART**();

Required Arguments

package
specifies an instance of the PCRXFIND package variable.

starting-index
specifies the position of the first character of the matched string. This is the same value as the previous call to the MATCH method.

1 or greater
one or greater is returned for the starting index.

-1
indicates that the specified capture group was not involved in the previous match.

Data type INTEGER

Details

The GETMATCHSTART method is used to determine the beginning position of the matched string within the subject string

See Also

Methods:

- “GETMATCH Method” on page 1760
- “GETMATCHEND Method” on page 1761
- “GETMATCHLENGTH Method” on page 1763

MATCH Method

Performs a match operation using the previously parsed regular expression.

Syntax

returnCode=*package*.**MATCH** (VARCHAR *subjectString*, [*offset-index*]) returns int;

Required Arguments

returnCode
specifies a status indicator.

1 or greater
returns the index where the match begins.

-1
indicates successful execution, but no match found.

Data type INTEGER

package
specifies an instance of the PCRXFIND package variable.

subjectString
specifies the character string to perform the regular expression against.

Data type VARCHAR

Optional Argument

offset-index
specifies an optional index to start the regular expression from.

Data type INTEGER

Details

The MATCH method returns either the index where the match begins or –1 if the regular expression is not found. Calling the MATCH method before parsing a regular expression results in an error. Upon successfully performing a match, methods such as GetGroupStart use the match results in their operation.

Example

This example searches a line of text using a regular expression. The MATCH method returns the index where the match begins.

```
data regex_test/overwrite=yes;
   dcl char(20) text regex;
   dcl double parse_rc match_rc;
   method run();
      dcl package pcrxfind prxf();
      text='quick brown fox';
      regex='/brown/';
      parse_rc=prxf.parse(regex);
      if parse_rc=0 then do;
         match_rc=prxf.match(text);
         put match_rc=;
      end;
    end;
enddata;
run;
```

The following line is written to the SAS log.

```
match_rc=7
```

See Also

Statements

- “DECLARE PACKAGE Statement: PCRXFIND Package” on page 1753

PARSE Method: PCRXFIND Package

Compiles a Perl-compatible regular expression that can be used for pattern matching.

Note: SAS has adopted the Perl Compatible Regular Expression (PCRE) for pattern matching character values. For more information, see PCRE - Perl Compatible Regular Expressions.

Syntax

returnCode=pcrxfind-package.**PARSE** (*pcrxfind-expression*);

Required Arguments

pcrxfind-expression
specifies a regular expression in the form of */expression/flag1...flagn*.

expression
The pattern used to match a substring within a string.

> **TIP** Backslashes within the *expression* can be used to escape special regular expression characters, such as \w or \d.

> **TIP** The substitution portion of the string does not require backslashes to escape any characters.

flag
These flags can be used to interpret the regular expression:

i
case insensitive — allows a case-insensitive pattern match within a string.

m
multiple lines — treats a string as a set of multiple lines. Allows the first character match or last character match to occur next to embedded newline characters.

n
non-capturing — prevents the grouping metacharacters () from capturing.

s
single line — treats a string as a single long line. Allows the match of a single character (.) to include the newline character.

x
match extension — allows whitespace characters (tab, newline, carriage return, and form feed characters) and comments in your match.

Restriction The matching mode modifiers p, o, c, a, and l are not supported.

pcrxfind-package
specifies an instance of the PCRXFIND package.

returnCode
specifies a status indicator.

0
indicates successful execution.

NULL
indicates failure and returns an error message.

Data type Integer

Details

The PARSE method in the PCRXFIND package is used to parse a Perl-compatible regular expression.

Compiling an expression allows the same package to parse using a different expression. A successful parse is required before you can use the MATCH method.

Example

The following example creates a PCRXFIND package and uses PARSE to compile a regular expression that matches phone numbers with the form *(nnn) nnn-nnnn*.

```
dcl package pcrxfind phoneFinder();
method init();
   dcl int rc;
   rc = phoneFinder.parse('/(\(\d{3}\)) \d{3}-\d{4}/');
   if null(rc) then do;
      put 'ERROR: Could not parse the provided expression.';
   end;
end;
```

See Also

Statements

- "DECLARE PACKAGE Statement: PCRXFIND Package" on page 1753

NEW Operator: PCRXFIND Package

Constructs an instance of a PCRXFIND package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable = **_NEW_ PCRXFIND**([*expression*]);

Required Argument

package-variable
specifies a name that can reference an instance of the package.

Optional Argument

expression
specifies a regular expression used to operate on the input text.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a PCRXFIND package variable using the DECLARE PACKAGE statement. After you declare a PCRXFIND package variable, use the _NEW_ operator to instantiate an instance of the PCRXFIND package and assign the new package instance to the PCRXFIND package variable.

```
declare package PCRXFIND finder;
finder = _new_ PCRXFIND();
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a logger package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package PCRXFIND finder();
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the PCRXFIND Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: PCRXFIND Package" on page 1753

21

DS2 PCRXREPLACE Package Methods, Operators, and Statements

Dictionary 1771
DECLARE PACKAGE Statement: PCRXREPLACE Package 1771
APPLY Method 1773
PARSE Method: PCRXREPLACE Package 1774
NEW Operator: PCRXREPLACE Package 1776

Dictionary

DECLARE PACKAGE Statement: PCRXREPLACE Package

Creates a package variable and gives you the option of creating an instance of the PCRXREPLACE package.

Category: Local

Restriction: This package is not supported on z/OS or 32-bit Windows operating systems.

Syntax

DECLARE PACKAGE PCRXREPLACE *variable* ([*regular-expression*]);

Arguments

variable
specifies a name that can reference an instance of the PCRXREPLACE package.

regular-expression
specifies a regular expression in the form of [s]/*expression*/*substitution*/*flags*. *Flags* can consist of one or more of these values:

g
global – Allows an expression match and replacement to occur as many times as possible within a string.

i
case insensitive – Allows a case-insensitive expression match within a string.

m
multiple lines – Treats the string as a set of multiple lines. Allows the first character match or last character match to occur next to embedded newline characters.

n
non-capturing – Disables numbered capturing parenthesis.

s
single line – Treats a string as a single long line. Allows the match of a single character (.) to include a newline character.

x
extends your pattern by allowing whitespace characters (tab, newline, carriage return, and form feed characters) and comments in your match.

Data type VARCHAR

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use a PCRXREPLACE package to replace or substitute text using a regular *expression*. The PCRXREPLACE package is predefined for DS2 programs.

You declare a PCRXREPLACE package by using the DECLARE PACKAGE statement. (Optional) You can include the regular expression to associate a PCRXREPLACE package with an expression and substitution text. After you declare the new PCRXREPLACE package, you can use the package to perform substitution operations on strings.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable

declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are three ways to construct an instance of a PCRXREPLACE package.

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package pcrxreplace swapper('/expression/replacementText/<flags>');
```

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package pcrxreplace swapper;
swapper = _new_ pcrxreplace(<regular-expression>);
```

- Use the DECLARE PACKAGE statement without an expression:

```
declare package pcrxreplace swapper;
swapper = _new_ pcrxreplace();
```

See Also

Operators:

- "_NEW_ Operator: PCRXREPLACE Package" on page 1776

APPLY Method

Applies the substitution expression, provided at instantiation time, to the loaded subject data.

Syntax

returnCode=*package*.**APPLY**(IN_OUT *subjectString*, [*numberOfTimes*]);

Required Arguments

package
specifies an instance of the PCRXREPLACE package variable.

subjectString
specifies the subject data.

Data type VARCHAR

returnCode
specifies the variable in which to place the return code.

0
indicates successful execution.

NULL
indicates failure and returns an error message.

Data type INTEGER

Optional Argument

numberOfTimes
specifies the number of times the change will be applied.

Data type INTEGER

Details

The APPLY method replaces a matched set of characters in a string, for a specified number of times.

See Also

Statements

- "DECLARE PACKAGE Statement: PCRXREPLACE Package" on page 1771

PARSE Method: PCRXREPLACE Package

Compiles a Perl-compatible regular expression that can be used for pattern matching.

Note: SAS has adopted the Perl Compatible Regular Expression (PCRE) for pattern matching character values. For more information, see PCRE - Perl Compatible Regular Expressions.

Syntax

returnCode=pcrxreplace-package.**PARSE** (*pcrxreplace-expression*);

Required Arguments

pcrxreplace-expression
specifies a regular expression in the form of [s]/*expression*/*substitution-text*/*flag1...flagn*.

expression
The pattern used to match a substring within a string.

> **TIP** Backslashes within the *expression* can be used to escape special regular expression characters, such as \w or \d.

> **TIP** The substitution portion of the string does not require backslashes to escape any characters.

flag
These flags can be used to interpret the expression:

g
global — allows a pattern match and replacement substitution to occur as many times as possible within a string.

i
case insensitive — allows a case-insensitive pattern match within a string.

m
multiple lines — treats a string as a set of multiple lines. Allows the first character match or last character match to occur next to embedded newline characters.

n
non-capturing — prevents the grouping metacharacters () from capturing.

s
single line — treats a string as a single long line. Allows the match of a single character (.) to include the newline character.

x
match extension — allows whitespace characters (tab, newline, carriage return, and form feed characters) and comments in your match.

Restriction The matching mode modifiers p, o, c, a, and l are not supported.

substitution-text
The text used to replace the pattern-matched text in the string.

Data type VARCHAR

pcrxreplace-package
specifies an instance of the PCRXREPLACE package.

returnCode
specifies a status indicator.

0
indicates successful execution.

NULL
indicates failure and returns an error message.

Data type Integer

Details

The PARSE method in the PCRXREPLACE package is used to parse a Perl-compatible regular expression.

Compiling an expression allows the same package to parse using a different expression. A successful parse is required before you can use the APPLY method.

Example: Compiling a Perl Regular Expression

The following example creates a PCRXREPLACE package and uses PARSE to compile a regular expression that converts phone numbers to the form *(nnn) nnn-nnnn*.

```
dcl package pcrxreplace normalizer();
method init();
   dcl int rc;
   rc = normalizer.parse(  '/.*(\d{3}).*(\d{3}).*(\d{4})/($1) $2-$3/' );
   if null(rc) then do;
      put 'ERROR: Could not parse the provided expression.';
   end;
end;
```

See Also

Methods

- “APPLY Method” on page 1773

Statements

- “DECLARE PACKAGE Statement: PCRXREPLACE Package” on page 1771

NEW Operator: PCRXREPLACE Package

Constructs an instance of a PCRXREPLACE package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable = **_NEW_ PCRXREPLACE**([*expression*]);

Required Argument

package-variable
specifies a name that can reference an instance of the package.

Optional Argument

expression
specifies a regular expression used to substitute values on the input text.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a PCRXREPLACE package variable using the DECLARE PACKAGE statement. After you declare a PCRXREPLACE package variable, use the _NEW_ operator to instantiate an instance of the PCRXREPLACE package and assign the new package instance to the PCRXREPLACE package variable.

```
declare package pcrxreplace swapper;
swapper = _new_ pcrxreplace();
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a logger package, you can declare and instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package pcrxreplace swapper();
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the PCRXREPLACE Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: PCRXREPLACE Package" on page 1771

22

DS2 SQLSTMT Package Methods, Operators, and Statements

Dictionary **1780**
BINDPARAMETERS Method 1780
BINDRESULTS Method 1781
CLOSERESULTS Method 1783
DECLARE PACKAGE Statement: SQLSTMT Package 1783
DELETE Method: SQLSTMT Package 1786
EXECUTE Method 1787
FETCH Method 1788
GETBIGINT Method 1789
GETBINARY Method 1790
GETCHAR Method 1792
GETCOLUMNCOUNT Method 1793
GETCOLUMNNAME Method 1794
GETCOLUMNTYPENAME Method 1795
GETDATE Method 1796
GETDECIMAL Method 1797
GETDOUBLE Method 1799
GETINTEGER Method 1800
GETNCHAR Method 1802
GETNUMERIC Method 1804
GETNVARCHAR Method 1805
GETREAL Method 1807
GETSMALLINT Method 1808
GETTIME Method 1810
GETTIMESTAMP Method 1811
GETTINYINT Method 1813
GETVARBINARY Method 1815
GETVARCHAR Method 1816
ISPREPARED Method 1818
NEW Operator: SQLSTMT Package 1819
PREPARE Method 1821

REGISTEROUTPARAMETER Method 1823
SETBIGINT Method 1825
SETBINARY Method 1826
SETCHAR Method 1827
SETDATE Method 1829
SETDECIMAL Method 1830
SETDOUBLE Method 1831
SETINTEGER Method 1832
SETNCHAR Method 1834
SETNUMERIC Method 1835
SETNVARCHAR Method 1836
SETREAL Method 1838
SETSMALLINT Method 1839
SETTIME Method 1840
SETTIMESTAMP Method 1841
SETTINYINT Method 1843
SETVARBINARY Method 1844
SETVARCHAR Method 1845

Dictionary

BINDPARAMETERS Method

Binds a list of variables to the parameters in the FedSQL statement.

Restriction: This method is not supported on the CAS server.

Syntax

package.**BINDPARAMETERS** ([*parameter-variable-list*]);

Required Argument

package
specifies an instance of the SQLSTMT package.

Optional Argument

[*parameter-variable-list*]
specifies a variable list or named variable list that contains the variables to bind to the FedSQL statement's parameters.

Requirement: Variables must be in the form of a variable list, which must be enclosed in brackets ([]) or a named variable list.

Tip The number of variables in the variable list must match the number of parameters in the FedSQL statement.

See "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*

Details

If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the SQLSTMT package's SET*type* methods.

The BINDPARAMETERS method binds the variables in the specified variable list to the parameters in the FedSQL statement.

Parameter values must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and a SET*type* method is invoked.

If the type of a bound variable differs from the corresponding parameter's type, the bound variable's value is converted to the parameter's type.

The BINDPARAMETERS method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

A run-time error also results if the BINDPARAMETERS method is called after the FedSQL statement is executed.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "BINDRESULTS Method" on page 1781
- SET "type" methods in this chapter

BINDRESULTS Method

Binds a list of variables to the columns of the result set of the FedSQL statement.

Restriction: This method is not supported on the CAS server.

Syntax

package.**BINDRESULTS** (*parameter-variable-list*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

parameter-variable-list
specifies a variable list or named variable list that contains the variables to bind to the columns of the result set.

Requirement: Variables must be in the form of a variable list, which must be enclosed in brackets ([]) or a named variable list.

Tip: The number of variables in the variable list must match the number of columns in the result set.

See: "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*

Details

The FETCH method returns the next row of data from the result set. If variables are bound to the result set columns with the BINDRESULTS method, then the fetched data for each result set column is placed in the variable bound to that column. If the type of a variable differs from the corresponding column's type, the column data value is converted to the variable's type.

The result data must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

A run-time error results if the BINDRESULTS method is called after the result data is fetched.

The BINDRESULTS method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "BINDPARAMETERS Method" on page 1780
- SET "type" methods in this chapter

CLOSERESULTS Method

Releases the result set from the last execution of the statement.

Restriction: This method is not supported on the CAS server.

Syntax

package.**CLOSERESULTS**( );

Required Argument

package
specifies an instance of the SQLSTMT package.

Details

An SQLSTMT instance maintains only one result set. The result set is automatically released when the FedSQL statement is executed or deleted.

The CLOSERESULTS method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

See Also

"Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

DECLARE PACKAGE Statement: SQLSTMT Package

Creates a package variable and enables you to create an instance of the SQLSTMT package.

Category: Local

Restriction: This statement is not supported on the CAS server.

Note: Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. *sql-text* must be enclosed in braces ({ }).

Syntax

Form 1: **DECLARE PACKAGE SQLSTMT** *variable* [('*sql-text*' [,\[*parameter-variable-list*\]])];

Form 2: **DECLARE PACKAGE SQLSTMT** *variable* [('*sql-text*' [, *connection-string*])];

Form 3: **DECLARE PACKAGE SQLSTMT** *variable* ( );

Form 4: **DECLARE PACKAGE SQLSTMT** *variable* ;

Arguments

variable
specifies a name that can reference an instance of the SQLSTMT package.

'sql-text'
is a valid FedSQL statement or string variable that contains a FedSQL statement that inserts into, updates, selects from, or deletes rows from a table.

CAUTION
Only FedSQL statements can be used with the SQLSTMT package. DBMS-specific SQL cannot be used. For more information, see *SAS FedSQL Language Reference*.

Requirement: The FedSQL statement must be enclosed in single quotation marks (') unless the statement is stored in a string variable.

Notes: The statement is a string literal.

The rules for identifiers for the FedSQL language apply to variables used in the SQLSTMT package, rather than the DS2 rules for identifiers. This occurs because FedSQL parses the string containing the SQL statement rather than DS2.

[*parameter-variable-list*]
specifies variables that are bound to the parameters contained in the FedSQL statement.

Requirements: Variables must be in the form of a variable list that must be enclosed in brackets ([]) or a named variable list.

Parameter data must be specified exclusively with either bound variables or exclusively with the SQLSTMT SET*type* methods.

See: "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*

connection-string
contains the fully specified connection string.

Default: If a connection string is not provided, the SQLSTMT package instance uses the connection string that is generated by the HPDS2 procedure or the DS2 procedure by using the attributes of the currently assigned libref.

Note The connection string is a string literal.

Tip A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

See This parameter is primarily designed for use with the SAS Federation Server. For more information about creating a fully specified connection string, see the *SAS Federation Server: Administrator's Guide*.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

The SQLSTMT package provides a way to pass FedSQL statements to a DBMS for execution. The FedSQL statements could create, modify, or delete tables. If the FedSQL statements selects rows from a table, the SQLSTMT package provides methods for interrogating the rows returned in a result set. The SQLSTMT package is predefined for DS2 programs.

You declare an SQLSTMT package by using the DECLARE PACKAGE statement. This associates an SQLSTMT package with an SQLSTMT name.

There are two ways to construct an instance of an SQLSTMT package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package sqlstmt sqlpkg;
sqlpkg = _new_ sqlstmt('update db2.dataset2 set y=? where x=?',[y x]);
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package sqlstmt sqlpkg('update db2.dataset2 set y=? where x=?',[y x]);
```

If the DECLARE statement includes arguments for construction within its parentheses (and no arguments is valid for the SQLSTMT package), then the package instance is allocated. If no parentheses are included, then a variable is created but the package instance is not allocated.

When an SQLSTMT instance is created with SQL text (Forms 1 and 2), the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the SQLSTMT package's SET*type* methods. The DECLARE PACKAGE statement binds the variables in the optional variable list to the parameters in the FedSQL statement.

When an SQLSTMT instance is created without FedSQL text (Form 3), the SQLSMT instance is allocated and left in an unprepared state. Use the PREPARE method to prepare the FedSQL statement at a later time.

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

For more information about SQLSTMT packages, see "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "BINDPARAMETERS Method" on page 1780
- "PREPARE Method" on page 1821
- SET "type" methods in this chapter

Operators:

- "_NEW_ Operator: SQLSTMT Package" on page 1819

DELETE Method: SQLSTMT Package

Deletes an instance of the SQLSTMT package.

Restriction: This method is not supported on the CAS server.

Note: The DELETE method is not required. When no SQLSTMT package variables reference an SQLSTMT package instance, the package instance is automatically deleted.

Syntax

package.**DELETE**( );

Required Argument

package
specifies an instance of the SQLSTMT package variable.

Details

When you no longer need the SQLSTMT package, delete it by using the DELETE method. If you attempt to use an SQLSTMT package after you delete it, an error will be written to the log.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

EXECUTE Method

Executes the FedSQL statement.

Restriction: This method is not supported on the CAS server.

Syntax

package.**EXECUTE**( );

Required Argument

package
specifies an instance of the SQLSTMT package.

Details

The EXECUTE method executes the FedSQL statement and returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error; 2 is returned if there is no data (NODATA). The NODATA condition exists when a FedSQL UPDATE or DELETE statement does not affect any rows.

An SQLSTMT instance maintains only one result set. The result set from the previous execution, if any, is released before the FedSQL statement is executed.

The FedSQL statement executes dynamically at run time. Because the statement is prepared at run time, it can be built and customized dynamically during the execution of the DS2 program.

See Also

"Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

FETCH Method

Fetches the next row of data from the result set of the FedSQL statement.

Restriction: This method is not supported on the CAS server.

Syntax

package.**FETCH** ([*result-variable-list*]);

Required Arguments

package
specifies an instance of the SQLSTMT package.

[*result-variable-list*]
specifies a variable list that contains the variables to bind to the columns of the result set.

Details

The FETCH method returns the next row of data from the result set. A status indicator is returned. Zero is returned for successful execution; 1 is returned if there is an error; 2 is returned if there is no data (NODATA). The NODATA condition exists if the next row to be fetched is located after the end of the result set.

If variables are bound to the result set columns with the BINDRESULTS method or by the FETCH method, then the fetched data for each result set column is placed in the variable bound to that column. If the variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

A run-time error results if FETCH is called before the statement is executed.

An SQLSTMT instance maintains only one result set. The result set from the previous execution, if any, is released before the FedSQL statement is executed. You can also use the CLOSERESULTS method to release the result set at any time.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "BINDRESULTS Method" on page 1781

- GET “type” methods in this chapter

GETBIGINT Method

Returns the value of the designated result set column as type BIGINT.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETBIGINT method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETBIGINT** (*index*);

package.**GETBIGINT** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column’s type is not type BIGINT, the column value is converted to type BIGINT and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETBIGINT method returns the value of the designated result set column as type BIGINT. If the designated result set column's type is not type BIGINT, the column value is converted to type BIGINT and then returned.

A run-time error results if the GETBIGINT method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETINTEGER Method" on page 1800
- "GETSMALLINT Method" on page 1808
- "GETTINYINT Method" on page 1813
- "SETBIGINT Method" on page 1825
- "SETINTEGER Method" on page 1832
- "SETSMALLINT Method" on page 1839
- "SETTINYINT Method" on page 1843

GETBINARY Method

Returns the designated result set column as type BINARY.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETBINARY method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the *rc* status indicator.

Syntax

variable=*package*.**GETBINARY** (*index*);

package.**GETBINARY** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type BINARY, the column value is converted to type BINARY and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETBINARY method returns the value of the designated result set column as type BINARY. If the designated result set column's type is not type BINARY, the column value is converted to type BINARY and then returned.

A run-time error results if the GETBINARY method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETVARBINARY Method" on page 1815
- "SETBINARY Method" on page 1826
- "SETVARBINARY Method" on page 1844

GETCHAR Method

Returns the designated result set column as type CHAR.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the *rc* status indicator.

Syntax

variable=*package*.**GETCHAR** (*index*);

package.**GETCHAR** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type CHAR, the column value is converted to type CHAR and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1 (ERROR) an error occurred during data retrieval.

2 (NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETCHAR method returns the value of the designated result set column as type CHAR. If the designated result set column's type is not type BIGINT, the column value is converted to type CHAR and then returned.

A run-time error results if the GETCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNCHAR Method" on page 1802
- "GETNVARCHAR Method" on page 1805
- "GETVARCHAR Method" on page 1816
- "SETCHAR Method" on page 1827
- "SETNCHAR Method" on page 1834
- "SETNVARCHAR Method" on page 1836
- "SETVARCHAR Method" on page 1845

GETCOLUMNCOUNT Method

Returns the number of columns in the result set.

Restriction: This method is not supported on the CAS server.

Syntax

variable=*package*.**GETCOLUMNCOUNT**();

Required Arguments

variable
specifies the variable that will hold the number of columns in the result set.

package
specifies an instance of the SQLSTMT package.

See Also

Methods:

- "GETCOLUMNNAME Method" on page 1794
- "GETCOLUMNTYPENAME Method" on page 1795

GETCOLUMNNAME Method

Returns the column name of the result set column with the designated index.

Restriction: This method is not supported on the CAS server.

Syntax

variable=*package*.**GETCOLUMNNAME**(*index*);

package.getColumnName(*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the name of the designated result set column.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

See Also

Methods:

- "GETCOLUMNCOUNT Method" on page 1793
- "GETCOLUMNTYPENAME Method" on page 1795

GETCOLUMNTYPENAME Method

Returns the data type of the result set column with the designated index.

Restriction: This method is not supported on the CAS server.

Syntax

variable=*package*.**GETCOLUMNTYPENAME**(*index*);

package.getColumnTypeName(*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the data type of the result set column.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

Details

GETCOLUMNTYPENAME returns only the data types that are supported by DS2. For more information, see "DS2 Data Types" in *SAS DS2 Programmer's Guide*.

See Also

Methods:

- "GETCOLUMNCOUNT Method" on page 1793
- "GETCOLUMNTYPENAME Method" on page 1795

GETDATE Method

Returns the designated result set column as type DATE.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETDATE method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETDATE** (*index*);

package.**GETDATE** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type DATE, the column value is converted to type DATE and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETDATE method returns the value of the designated result set column as type DATE. If the designated result set column's type is not type DATE, the column value is converted to type DATE and then returned.

A run-time error results if the GETDATE method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETTIME Method" on page 1810
- "GETTIMESTAMP Method" on page 1811
- "SETDATE Method" on page 1829
- "SETTIME Method" on page 1840
- "SETTIMESTAMP Method" on page 1841

GETDECIMAL Method

Returns the designated result set column as type DECIMAL.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETDECIMAL method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETDECIMAL** (*index*);

package.**GETDECIMAL** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type DECIMAL, the column value is converted to type DECIMAL and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETDECIMAL method returns the value of the designated result set column as type DECIMAL. If the designated result set column's type is not type DECIMAL, the column value is converted to type DECIMAL and then returned.

A run-time error results if the GETDECIMAL method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "SETDECIMAL Method" on page 1830

GETDOUBLE Method

Returns the designated result set column as type DOUBLE.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETDOUBLE method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETDOUBLE** (*index*);

package.**GETDOUBLE** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type DOUBLE, the column value is converted to type DOUBLE and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETDOUBLE method returns the value of the designated result set column as type DOUBLE. If the designated result set column's type is not type DOUBLE, the column value is converted to type DOUBLE and then returned.

A run-time error results if the GETDOUBLE method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "SETDOUBLE Method" on page 1831

GETINTEGER Method

Gets the designated result set column as type INTEGER.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETINTEGER method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETINTEGER** (*index*);

package.**GETINTEGER** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type INTEGER, the column value is converted to type INTEGER and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETINTEGER method returns the value of the designated result set column as type INTEGER. If the designated result set column's type is not type INTEGER, the column value is converted to type INTEGER and then returned.

A run-time error results if the GETINTEGER method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBIGINT Method" on page 1789
- "GETSMALLINT Method" on page 1808
- "GETTINYINT Method" on page 1813
- "SETBIGINT Method" on page 1825
- "SETINTEGER Method" on page 1832
- "SETSMALLINT Method" on page 1839
- "SETTINYINT Method" on page 1843

GETNCHAR Method

Gets the designated result set column as type NCHAR.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the *rc* status indicator.

Syntax

variable=*package*.**GETNCHAR** (*index*);

package.**GETNCHAR** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type NCHAR, the column value is converted to type NCHAR and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETNCHAR method returns the value of the designated result set column as type NCHAR. If the designated result set column's type is not type NCHAR, the column value is converted to type NCHAR and then returned.

A run-time error results if the GETNCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETCHAR Method" on page 1792
- "GETNVARCHAR Method" on page 1805
- "GETVARCHAR Method" on page 1816

- "SETCHAR Method" on page 1827
- "SETNCHAR Method" on page 1834
- "SETNVARCHAR Method" on page 1836
- "SETVARCHAR Method" on page 1845

GETNUMERIC Method

Returns the designated result set column as type NUMERIC.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETNUMERIC method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETNUMERIC** (*index*);

package.**GETNUMERIC** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type NUMERIC, the column value is converted to type NUMERIC and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETNUMERIC method returns the value of the designated result set column as type NUMERIC. If the designated result set column's type is not type NUMERIC, the column value is converted to type NUMERIC and then returned.

A run-time error results if the GETNUMERIC method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "SETNUMERIC Method" on page 1835

GETNVARCHAR Method

Returns the designated result set column as type NVARCHAR.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETVARCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the *rc* status indicator.

Syntax

variable=*package*.**GETNVARCHAR** (*index*);

package.**GETNVARCHAR** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type NVARCHAR, the column value is converted to type NVARCHAR and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETNVARCHAR method returns the value of the designated result set column as type NVARCHAR. If the designated result set column's type is not type NVARCHAR, the column value is converted to type NVARCHAR and then returned.

A run-time error results if the GETNVARCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETCHAR Method" on page 1792
- "GETNCHAR Method" on page 1802
- "GETVARCHAR Method" on page 1816
- "SETCHAR Method" on page 1827
- "SETNCHAR Method" on page 1834
- "SETNVARCHAR Method" on page 1836
- "SETVARCHAR Method" on page 1845

GETREAL Method

Returns the designated result set column as type REAL.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETREAL method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETREAL** (*index*);

package.**GETREAL** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type REAL, the column value is converted to type REAL and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETREAL method returns the value of the designated result set column as type REAL. If the designated result set column's type is not type REAL, the column value is converted to type REAL and then returned.

A run-time error results if the GETREAL method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "SETREAL Method" on page 1838

GETSMALLINT Method

Returns the designated result set column as type SMALLINT.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETSMALLINT method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETSMALLINT** (*index*);

package.**GETSMALLINT** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type SMALLINT, the column value is converted to type SMALLINT and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETSMALLINT method returns the value of the designated result set column as type SMALLINT. If the designated result set column's type is not type SMALLINT, the column value is converted to type SMALLINT and then returned.

A run-time error results if the GETSMALLINT method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBIGINT Method" on page 1789
- "GETINTEGER Method" on page 1800
- "GETTINYINT Method" on page 1813
- "SETBIGINT Method" on page 1825
- "SETINTEGER Method" on page 1832
- "SETSMALLINT Method" on page 1839
- "SETTINYINT Method" on page 1843

GETTIME Method

Returns the designated result set column as type TIME.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETTIME method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETTIME** (*index*);

package.**GETTIME** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type TIME, the column value is converted to type TIME and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The GETTIME method returns the value of the designated result set column as type TIME. If the designated result set column's type is not type TIME, the column value is converted to type TIME and then returned.

A run-time error results if the GETTIME method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETDATE Method" on page 1796
- "GETTIMESTAMP Method" on page 1811
- "SETDATE Method" on page 1829
- "SETTIME Method" on page 1840
- "SETTIMESTAMP Method" on page 1841

GETTIMESTAMP Method

Returns the designated result set column as type TIMESTAMP.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETTIMESTAMP method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETTIMESTAMP** (*index*);

package.**GETTIMESTAMP** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type TIMESTAMP, the column value is converted to type TIMESTAMP and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETTIMESTAMP method returns the value of the designated result set column as type TIMESTAMP. If the designated result set column's type is not type TIMESTAMP, the column value is converted to type TIMESTAMP and then returned.

A run-time error results if the GETTIMESTAMP method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETDATE Method" on page 1796
- "GETTIME Method" on page 1810
- "SETDATE Method" on page 1829
- "SETTIME Method" on page 1840
- "SETTIMESTAMP Method" on page 1841

GETTINYINT Method

Returns the designated result set column as type TINYINT.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETTINYINT method once to return the result set column data. Subsequent method calls result in a value of 2 (NODATA) for the *rc* status indicator.

Syntax

variable=*package*.**GETTINYINT** (*index*);

package.**GETTINYINT** (*index*, *variable*, *rc*);

Required Arguments

variable

specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type TINYINT, the column value is converted to type TINYINT and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETTINYINT method returns the value of the designated result set column as type TINYINT. If the designated result set column's type is not type TINYINT, the column value is converted to type TINYINT and then returned.

A run-time error results if the GETTINYINT method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBIGINT Method" on page 1789
- "GETINTEGER Method" on page 1800
- "GETSMALLINT Method" on page 1808
- "SETBIGINT Method" on page 1825
- "SETINTEGER Method" on page 1832

- "SETSMALLINT Method" on page 1839
- "SETTINYINT Method" on page 1843

GETVARBINARY Method

Returns the designated result set column as type VARBINARY.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETVARBINARY method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the *rc* status indicator.

Syntax

variable=*package*.**GETVARBINARY** (*index*);

package.**GETVARBINARY** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type VARBINARY, the column value is converted to type VARBINARY and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETVARBINARY method returns the value of the designated result set column as type VARBINARY. If the designated result set column's type is not type VARBINARY, the column value is converted to type VARBINARY and then returned.

A run-time error results if the GETVARBINARY method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBINARY Method" on page 1790
- "SETBINARY Method" on page 1826
- "SETVARBINARY Method" on page 1844

GETVARCHAR Method

Returns the designated result set column as type VARCHAR.

Restriction: This method is not supported on the CAS server.

Note: You can call the GETVARCHAR method repeatedly to return the result set column data. When all data has been retrieved, a value of 2 (NODATA) is returned for the *rc* status indicator.

Syntax

variable=*package*.**GETVARCHAR** (*index*);

package.**GETVARCHAR** (*index*, *variable*, *rc*);

Required Arguments

variable
specifies the variable that will hold the value of the designated result set column.

Note If the designated result set column's type is not type VARCHAR, the column value is converted to type VARCHAR and then returned.

package
specifies an instance of the SQLSTMT package.

index
specifies the result set column index ordered sequentially, starting at 1.

rc
specifies the variable in which to place the return code. The following values are possible:

0
(SUCCESS) the result set column data is retrieved.

1
(ERROR) an error occurred during data retrieval.

2
(NODATA) there is no more data to be retrieved.

Details

The FETCH method returns the next row of data from the result set. If variables are not bound to the result set columns, the fetched data can be returned by the GET*type* methods.

The GETVARCHAR method returns the value of the designated result set column as type VARCHAR. If the designated result set column's type is not type VARCHAR, the column value is converted to type VARCHAR and then returned.

A run-time error results if the GETVARCHAR method is called before the result set is fetched.

The result set must be accessed exclusively with bound variables or exclusively with the GET*type* methods. A run-time error results if variables are bound to result set columns and a GET*type* method is invoked.

For more information, see "Accessing Result Set Data" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- “GETCHAR Method” on page 1792
- “GETNCHAR Method” on page 1802
- “GETNVARCHAR Method” on page 1805
- “SETCHAR Method” on page 1827
- “SETNCHAR Method” on page 1834
- “SETNVARCHAR Method” on page 1836
- “SETVARCHAR Method” on page 1845

ISPREPARED Method

Returns a value that indicates whether the SQLSTMT package instance is prepared.

Restriction: This method is not supported on the CAS server.

Syntax

package.**ISPREPARED**( );

Required Argument

package
specifies an instance of the SQLSTMT package.

Details

The ISPREPARED method returns a value of 0 (false) if the SQLSTMT package instance is not prepared. A nonzero value (true) is returned if the SQLSTMT package instance is prepared.

See Also

Methods:

- “PREPARE Method” on page 1821

Statements:

- “DECLARE PACKAGE Statement: SQLSTMT Package” on page 1783

NEW Operator: SQLSTMT Package

Constructs an instance of an SQLSTMT package.

Restriction: This operator is not supported on the CAS server.

Notes: Braces in the syntax convention indicate a syntax grouping. The escape character (\ ) before a brace indicates that the brace is required in the syntax. *sql-text* must be enclosed in braces ({ }).

The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

Form 1: *package-variable*=**_NEW_ SQLSTMT** ('*sql-text*' [\[*parameter-variable-list*\]]);

Form 2: *package-variable*=**_NEW_ SQLSTMT** ('*sql-text*' [, *connection-string*]);

Form 3: *package-variable*=**_NEW_ SQLSTMT** ( );

Required Arguments

package-variable
specifies a name that can reference an instance of the SQLSTMT package.

'sql-text'
is a valid FedSQL statement that inserts into, updates, selects from, or deletes rows from a table.

CAUTION
Only FedSQL statements can be used with the SQLSTMT package. DBMS-specific SQL cannot be used. For more information, see *SAS FedSQL Language Reference*.

Requirement: The FedSQL statement must be enclosed in single quotation marks (').

Notes: The statement can be a string literal, a string value generated from an expression, or a string value that is stored in a variable.

The rules for identifiers for the FedSQL language apply to variables used in the SQLSTMT package, rather than the DS2 rules for identifiers. This occurs because FedSQL parses the string containing the SQL statement rather than DS2.

Optional Arguments

[*parameter-variable-list*]
specifies variables that are bound to the parameters contained in the FedSQL statement.

Requirements: Variables must be in the form of a variable list that must be enclosed in brackets ([]) or a named variable list.

Parameter values must be specified exclusively with either bound variables or exclusively with the SQLSTMT SET*type* methods.

See: "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*

connection-string
contains the fully specified connection string.

Default: If a connection string is not provided, the SQLSTMT package instance uses the connection string that is generated by the HPDS2 procedure or the DS2 procedure by using the attributes of the currently assigned libref.

Note: The connection string can be a string literal, a string value generated from an expression, or a string value that is stored in a variable.

Tip: A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

See: This parameter is primarily designed for use with the SAS Federation Server. For more information about creating a fully specified connection string, see the *SAS Federation Server: Administrator's Guide*.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use an SQLSTMT package to create and delete tables, select rows from a table, and access the returned result set. The SQLSTMT package is predefined for DS2 programs.

You can declare an SQLSTMT package by using the DECLARE PACKAGE statement. This associates an SQLSTMT package with an SQLSTMT name.

There are two ways to construct an instance of an SQLSTMT package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package sqlstmt sqlpkg;
sqlpkg = _new_ sqlstmt('update db2.dataset2 set y=? where x=?',[y x]);
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package sqlstmt sqlpkg('update db2.dataset2 set y=? where x=?',[y x]);
```

When an SQLSTMT instance is created with SQL text, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the SQLSTMT package's SET*type* methods. The DECLARE PACKAGE statement binds the variables in the optional variable list to the parameters in the FedSQL statement.

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Methods:

- "BINDPARAMETERS Method" on page 1780

Statements:

- "DECLARE PACKAGE Statement: SQLSTMT Package" on page 1783

PREPARE Method

Prepares a FedSQL statement.

Restriction: This method is not supported on the CAS server.

Syntax

Form 1: **PREPARE** ('*sql-text*');

Form 2: **PREPARE** ('*sql-text*' , *connection-string*);

Required Arguments

'sql-text'
is a valid FedSQL statement or string variable that contains a FedSQL statement that inserts into, updates, selects from, or deletes rows from a table.

CAUTION

Only FedSQL statements can be used with the SQLSTMT package. DBMS-specific SQL cannot be used. For more information, see *SAS FedSQL Language Reference*.

Requirement The FedSQL statement must be enclosed in single quotation marks (') unless the statement is stored in a string variable.

Notes The statement is a string literal.

The rules for identifiers for the FedSQL language apply to variables that are used in the SQLSTMT package, rather than the DS2 rules for identifiers. This occurs because FedSQL (not DS2) parses the string containing the FedSQL statement.

connection-string
contains the fully specified connection string.

Default If a connection string is not provided, the SQLSTMT package instance uses the connection string that is generated by the HPDS2 procedure or the DS2 procedure by using the attributes of the currently assigned libref.

Note The connection string is a string literal.

Tip A connection string defines how to connect to the data. A connection string identifies the query language to be submitted as well as the information required to connect to the data source or sources.

See This parameter is primarily designed for use with the SAS Federation Server. For more information about creating a fully specified connection string, see *SAS Federation Server: Administrator's Guide*.

Details

The PREPARE method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

A run-time error occurs if you call the PREPARE method and the FedSQL statement is already prepared.

See Also

Methods:

- "ISPREPARED Method" on page 1818

Statements:

- "DECLARE PACKAGE Statement: SQLSTMT Package" on page 1783

REGISTEROUTPARAMETER Method

Registers the designated parameter as an output parameter.

Restriction: This method is not supported on the CAS server.

Syntax

REGISTEROUTPARAMETER (*index*)

Required Argument

index
is the parameter index, ordered sequentially starting at 1.

Details

The REGISTEROUTPARAMETER is used to map output parameters in the FedSQL statement to IN_OUT parameters in a DS2 package METHOD statement.

The REGISTEROUTPARAMETER method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

Note: Parameter data must be specified exclusively with bound variables. A run-time error results if an SQLSTMT SET*type* method and the REGISTEROUTPARAMETER method are invoked.

Note: After the FedSQL statement is executed, the values of the statement's output parameters are retrieved and written to the bound variables.

Example

The following example illustrates how the SQLSTMT package can be used to execute a DS2 package METHOD statement. The example also shows how to map output parameters in the FedSQL statement to the IN_OUT parameters in the DS2 package METHOD statement.

```
proc ds2;
package swapper / overwrite=yes;
   method swap(in_out int x, in_out int y);
      dcl int t;
```

```
      t = x;
      x = y;
      y = t;
   end;
endpackage;
run;

data _null_;
   dcl int a b;
   method init();
      dcl package sqlstmt stmt();
      dcl int rc;

      rc = stmt.prepare('call swapper.swap(?, ?)');
      if (rc) then put 'TEST ERROR: prepare';

      rc = stmt.bindparameters([a, b]);
      if (rc) then put 'TEST ERROR: bindParameters';

      rc = stmt.registerOutParameter(1);
      if (rc) then put 'TEST ERROR: registerOutParameter';

      rc = stmt.registerOutParameter(2);
      if (rc) then put 'TEST ERROR: registerOutParameter';

      a = 42;
      b = 101;

      put 'Before executing swapper.swap:' a= b=;
      rc = stmt.execute();
      if (rc) then put 'TEST ERROR: execute';
      put 'After executing swapper.swap:' a= b=;
   end;
enddata;
run;
quit;
```

The following lines are written to the SAS log:

```
Before executing swapper.swap: a=42 b=101
After executing swapper.swap: a=101 b=42
```

See Also

Statements:

- “METHOD Statement” on page 1270

SETBIGINT Method

Sets the designated parameter to the specified value of type BIGINT.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETBIGINT** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type BIGINT, the BIGINT value is converted to the designated parameter's type. For example, if you use `setbigint(1, 3)` to set parameter 1 to BIGINT value 3, and parameter 1 is type CHAR, the BIGINT value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETBIGINT method is invoked.

The SETBIGINT method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBIGINT Method" on page 1789
- "GETINTEGER Method" on page 1800
- "GETSMALLINT Method" on page 1808
- "GETTINYINT Method" on page 1813
- "SETINTEGER Method" on page 1832
- "SETSMALLINT Method" on page 1839
- "SETTINYINT Method" on page 1843

SETBINARY Method

Sets the designated parameter to the specified value of type BINARY.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETBINARY** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for

the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type BINARY, the BINARY value is converted to the designated parameter's type. For example, if you use `setbinary(1, 0110)` to set parameter 1 to BINARY value 0110, and parameter 1 is type CHAR, the BINARY value 0110 is converted to the CHAR value 0110 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETBINARY method is invoked.

The SETBINARY method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBINARY Method" on page 1790
- "GETVARBINARY Method" on page 1815
- "SETVARBINARY Method" on page 1844

SETCHAR Method

Sets the designated parameter to the specified value of type CHAR.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETCHAR** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or an expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type CHAR, the CHAR value is converted to the designated parameter's type. For example, if you use `setchar(1, 3)` to set parameter 1 to CHAR value 3, and parameter 1 is type INTEGER, the CHAR value 3 is converted to the INTEGER value 3 and the INTEGER value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETCHAR method is invoked.

The SETCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETCHAR Method" on page 1792
- "GETNCHAR Method" on page 1802
- "GETNVARCHAR Method" on page 1805
- "GETVARCHAR Method" on page 1816
- "SETNCHAR Method" on page 1834
- "SETNVARCHAR Method" on page 1836
- "SETVARCHAR Method" on page 1845

SETDATE Method

Sets the designated parameter to the specified value of type DATE.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETDATE** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type DATE, the DATE value is converted to the designated parameter's type.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETDATE method is invoked.

The SETDATE method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETDATE Method" on page 1796
- "GETTIME Method" on page 1810
- "GETTIMESTAMP Method" on page 1811
- "SETTIME Method" on page 1840
- "SETTIMESTAMP Method" on page 1841

SETDECIMAL Method

Sets the designated parameter to the specified value of type DECIMAL.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETDECIMAL** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type DECIMAL, the DECIMAL value is converted to the designated parameter's type. For example, if you use `setdecimal(1, 13.4)` to set parameter 1 to DECIMAL value 13.4, and parameter 1 is type CHAR, the DECIMAL value 13.4 is converted to the CHAR value 13.4 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETDECIMAL method is invoked.

The SETDECIMAL method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETDECIMAL Method" on page 1797

SETDOUBLE Method

Sets the designated parameter to the specified value of type DOUBLE.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETDOUBLE** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type DOUBLE, the DOUBLE value is converted to the designated parameter's type. For example, if you use `setdouble(1, 33443452)` to set parameter 1 to DOUBLE value 33443452, and parameter 1 is type CHAR, the DOUBLE value 33443452 is converted to the CHAR value 33443452 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETDOUBLE method is invoked.

The SETDOUBLE method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETDOUBLE Method" on page 1799

SETINTEGER Method

Sets the designated parameter to the specified value of type INTEGER.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETINTEGER** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type INTEGER, the INTEGER value is converted to the designated parameter's type. For example, if you use `setinteger(1, 3)` to set parameter 1 to INTEGER value 3, and parameter 1 is type CHAR, the INTEGER value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETINTEGER method is invoked.

The SETINTEGER method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBIGINT Method" on page 1789
- "GETINTEGER Method" on page 1800
- "GETSMALLINT Method" on page 1808
- "GETTINYINT Method" on page 1813
- "SETBIGINT Method" on page 1825
- "SETSMALLINT Method" on page 1839
- "SETTINYINT Method" on page 1843

SETNCHAR Method

Sets the designated parameter to the specified value of type NCHAR.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETNCHAR** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type NCHAR, the NCHAR value is converted to the designated parameter's type. For example, if you use `setnchar(1, 3)` to set parameter 1 to NCHAR value 3, and parameter 1 is type INTEGER, the NCHAR value 3 is converted to the INTEGER value 3 and the INTEGER value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETNCHAR method is invoked.

The SETNCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETCHAR Method" on page 1792
- "GETNCHAR Method" on page 1802
- "GETNVARCHAR Method" on page 1805
- "GETVARCHAR Method" on page 1816
- "SETCHAR Method" on page 1827
- "SETNVARCHAR Method" on page 1836
- "SETVARCHAR Method" on page 1845

SETNUMERIC Method

Sets the designated parameter to the specified value of type NUMERIC.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETNUMERIC** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for

the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type NUMERIC, the NUMERIC value is converted to the designated parameter's type. For example, if you use `setnumeric(1, 3)` to set parameter 1 to NUMERIC value 3, and parameter 1 is type CHAR, the NUMERIC value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETNUMERIC method is invoked.

The SETNUMERIC method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETNUMERIC Method" on page 1804

SETNVARCHAR Method

Sets the designated parameter to the specified value of type NVARCHAR.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETNVARCHAR** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type NVARCHAR, the NVARCHAR value is converted to the designated parameter's type. For example, if you use `setnvarchar(1, 3)` to set parameter 1 to NVARCHAR value 3, and parameter 1 is type INTEGER, the NVARCHAR value 3 is converted to the INTEGER value 3 and the INTEGER value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETNVARCHAR method is invoked.

The SETNVARCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETCHAR Method" on page 1792
- "GETNCHAR Method" on page 1802
- "GETNVARCHAR Method" on page 1805
- "GETVARCHAR Method" on page 1816
- "SETCHAR Method" on page 1827
- "SETNCHAR Method" on page 1834
- "SETVARCHAR Method" on page 1845

SETREAL Method

Sets the designated parameter to the specified value of type REAL.

Syntax

package.**SETREAL** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type REAL, the REAL value is converted to the designated parameter's type. For example, if you use `setreal(1, 3)` to set parameter 1 to REAL value 3, and parameter 1 is type CHAR, the REAL value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETREAL method is invoked.

The SETREAL method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETREAL Method" on page 1807

SETSMALLINT Method

Sets the designated parameter to the specified value of type SMALLINT.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETSMALLINT** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type SMALLINT, the SMALLINT value is converted to the designated parameter's type. For example, if you use `setsmallint(1, 3)` to set parameter 1 to SMALLINT value 3, and parameter 1 is type CHAR, the SMALLINT value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETSMALLINT method is invoked.

The SETSMALLINT method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBIGINT Method" on page 1789
- "GETINTEGER Method" on page 1800
- "GETSMALLINT Method" on page 1808
- "GETTINYINT Method" on page 1813
- "SETBIGINT Method" on page 1825
- "SETINTEGER Method" on page 1832
- "SETTINYINT Method" on page 1843

SETTIME Method

Sets the designated parameter to the specified value of type TIME.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETTIME** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type TIME, the TIME value is converted to the designated parameter's type.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETTIME method is invoked.

The SETTIME method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETDATE Method" on page 1796
- "GETTIME Method" on page 1810
- "GETTIMESTAMP Method" on page 1811
- "SETDATE Method" on page 1829
- "SETTIMESTAMP Method" on page 1841

SETTIMESTAMP Method

Sets the designated parameter to the specified value of type TIMESTAMP.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETTIMESTAMP** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type TIMESTAMP, the TIMESTAMP value is converted to the designated parameter's type.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETTIMESTAMP method is invoked.

The SETTIMESTAMP method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETDATE Method" on page 1796
- "GETTIME Method" on page 1810
- "GETTIMESTAMP Method" on page 1811
- "SETDATE Method" on page 1829

■ "SETTIME Method" on page 1840

SETTINYINT Method

Sets the designated parameter to the specified value of type TINYINT.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETTINYINT** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type TINYINT, the TINYINT value is converted to the designated parameter's type. For example, if you use `settinyint(1, 3)` to set parameter 1 to TINYINT value 3, and parameter 1 is type CHAR, the TINYINT value 3 is converted to the CHAR value 3 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETTINYINT method is invoked.

The SETTINYINT method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBIGINT Method" on page 1789
- "GETINTEGER Method" on page 1800
- "GETSMALLINT Method" on page 1808
- "GETTINYINT Method" on page 1813
- "SETBIGINT Method" on page 1825
- "SETINTEGER Method" on page 1832
- "SETSMALLINT Method" on page 1839

SETVARBINARY Method

Sets the designated parameter to the specified value of type VARBINARY.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETVARBINARY** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type VARBINARY, the VARBINARY value is converted to the designated parameter's type. For example, if you use `setvarbinary(1, 0110)` to set parameter 1 to VARBINARY value 0110, and parameter 1 is type CHAR, the VARBINARY value 0110 is converted to the CHAR value 0110 and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETVARBINARY method is invoked.

The SETVARBINARY method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETBINARY Method" on page 1790
- "GETVARBINARY Method" on page 1815
- "SETBINARY Method" on page 1826

SETVARCHAR Method

Sets the designated parameter to the specified value of type VARCHAR.

Restriction: This method is not supported on the CAS server.

Syntax

package.**SETVARCHAR** (*index*, *value*);

Required Arguments

package
specifies an instance of the SQLSTMT package.

index
specifies the parameter index ordered sequentially, starting at 1.

value
specifies the value to which to set the designated parameter. The designated parameter is specified by *index*.

Tip *value* can be a literal, variable, or expression.

Details

When an SQLSTMT instance is created, the FedSQL statement is allocated and prepared. If the FedSQL statement contains parameters, values to substitute for the parameters must be obtained to execute the FedSQL statement. The substitution values can be specified with either the current values of bound variables or with the package's SET*type* methods.

If the designated parameter's type is not type VARCHAR, the VARCHAR value is converted to the designated parameter's type. For example, if you use `setvarchar(1, pass)` to set parameter 1 to VARCHAR value **`pass`**, and parameter 1 is type CHAR, the VARCHAR value **`pass`** is converted to the CHAR value **`pass`** and the CHAR value is used to set the parameter.

Parameter data must be specified exclusively with bound variables or exclusively with the SET*type* methods. A run-time error results if variables are bound to parameters and the SETVARCHAR method is invoked.

The SETVARCHAR method returns a status indicator. Zero is returned for successful execution; 1 is returned if there is an error.

For more information, see "Specifying FedSQL Statement Parameter Values" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the SQLSTMT Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETCHAR Method" on page 1792
- "GETNCHAR Method" on page 1802
- "GETNVARCHAR Method" on page 1805
- "GETVARCHAR Method" on page 1816
- "SETCHAR Method" on page 1827

- “SETNCHAR Method” on page 1834
- “SETNVARCHAR Method” on page 1836

23

DS2 TZ Package Methods, Operators, and Statements

Dictionary 1849
DECLARE PACKAGE Statement: TZ Package 1849
GETLOCALTIME Method 1851
GETOFFSET Method 1852
GETOFFSETUTC Method 1854
GETTIMEZONEID Method 1855
GETTIMEZONENAME Method 1857
GETUTCTIME Method 1858
NEW Operator: TZ Package 1859
TOISO8601 Method 1860
TOLOCALTIME Method 1861
TOTIMESTAMPZ Method 1862
TOUTCTIME Method 1863

Dictionary

DECLARE PACKAGE Statement: TZ Package

Creates a package variable and enables you to create an instance of the TZ package.

Category: Local

Syntax

DECLARE PACKAGE TZ *variable* ([*time-zone-id*]);

Arguments

variable
specifies a name that can reference an instance of the TZ package.

time-zone-id
specifies a time zone ID.

Default The value specified in the TIMEZONE= system option.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You use a TZ package for time zone processing. The TZ package is predefined for DS2 programs. For more information about time zones in SAS , see *SAS National Language Support (NLS): Reference Guide*.

You declare a TZ package by using the DECLARE PACKAGE statement. This associates a TZ package with a time zone name. After you declare the new TZ package, you can format your date and time data accordingly.

When a package is declared, a variable is created that can reference an instance of the package. If constructor arguments are provided with the package variable declaration, then a package instance is constructed and the package variable is set to reference the constructed package instance. Multiple package variables can be created and multiple package instances can be constructed with a single DECLARE PACKAGE statement, and each package instance represents a completely separate copy of the package.

There are two ways to construct an instance of a TZ package.

- Use the DECLARE PACKAGE statement along with the_NEW_ operator:

```
declare package tz tzpkg;
tzpkg = _new_ tz();
```

- Use the DECLARE PACKAGE statement along with its constructor syntax:

```
declare package tz tzpkg();
```

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Operators:

- "_NEW_ Operator: TZ Package" on page 1859

System Options:

- "TIMEZONE= System Option" in *SAS System Options: Reference*

GETLOCALTIME Method

Returns current local time.

Syntax

variable=*package*.**GETLOCALTIME** ([*time-zone-ID*]);

Required Arguments

variable
specifies the variable that will hold the value of the time zone ID of required local time.

package
specifies an instance of the TZ package.

Optional Argument

time-zone-ID
specifies the time zone ID of the required local time.

Example

The following example uses multiple time zones.

```
data _null_ ;
   method init();
      dcl package tz tokyo('Asia/Tokyo')
         london('Europe/London')
         new_york('America/New_York') ;
      dcl double tokyo_time london_time new_york_time utc_time ;
      dcl integer tokyo_off london_off new_york_off ;

      tokyo_time = tokyo.getLocalTime();
      tokyo_off = tokyo.getOffset();
      london_time = london.getLocalTime() ;
      london_off = london.getOffset() ;
      new_york_time = new_york.getLocalTime() ;
```

```
      new_york_off = new_york.getOffset();

      utc_time = tokyo.getUTCTime(); /* can use any timezone */

      put utc_time = datetime. ;
      put tokyo_time = datetime. tokyo_off time5.;
      put london_time = datetime. london_off time5. ;
      put new_york_time = datetime. new_york_off time5. ;

   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
utc_time=08APR15:12:48:41
tokyo_time=08APR15:21:48:41  9:00
london_time=08APR15:13:48:41  1:00
new_york_time=08APR15:08:48:41 -4:00
```

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Methods:

- "GETUTCTIME Method" on page 1858

GETOFFSET Method

Returns the time zone offset of the time zone from Universal Coordinated Time (UTC) at the specified local time. If local time is not specified, current local time is used.

Syntax

Form 1: *variable*=*package*.**GETOFFSET** ( );

Form 2: *variable*=*package*.**GETOFFSET** (*local-time*);

Form 3: *variable*=*package*.**GETOFFSET** (*time-zone-ID*);

Form 4: *variable*=*package*.**GETOFFSET** (*local-time*, *time-zone-ID*);

Required Arguments

variable
specifies the variable that will hold the value of the time zone offset.

package
specifies an instance of the TZ package.

local-time
specifies the local time used to get the time zone offset.

Default If local time is not specified, the current local time is used.

Tip *localTime* is a SAS date time value. It is used as the number of seconds since January 1, 1960 00:00:00 local time.

time-zone-ID
specifies the time zone ID of the required time zone offset.

See "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Details

UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a datetime value that uses the ISO 8601 basic form *yyyymmdd*T*hhmmss*+|–*hhmm* or the ISO 8601 extended form *yyyy-mm-dd*T*hh*:*mm*:*ss*+|–*hh*:*mm*.

The time zone offset specifies the number of hours and minutes that a time zone is off from the UTC in the form +|–*hhmm* or +|–*hh*:*mm*

Example

The following example returns the offset from the 'Asia/Tokyo' time zone to the 'America/New_York' time zone. The example also illustrates the different ways in which the time zone ID can be expressed.

```
data _null_;
   method init();

      declare package tz tzone('asia/tokyo');
      dcl double new_york;
      dcl char(40) cstr;

      new_york = tzone.getOffset('America/New_York');
      put new_york time.;

      new_york = tzone.getOffset(n'America/New_York');
      put new_york time.;

      cstr = 'America/New_York';
```

```
      new_york = tzone.getOffset(cstr);
      put new_york time.;

   end;
 enddata;
run;
```

The following lines are written to the SAS log.

```
-4:00:00
-4:00:00
-4:00:00
```

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETOFFSETUTC Method" on page 1854

GETOFFSETUTC Method

Returns the time zone offset of the time zone from UTC at the specified UTC time.

Syntax

variable=*package*.**GETOFFSETUTC** (*UTC-time*, *time-zone-ID*);

Without Arguments

If no arguments are specified, the GETOFFSETUTC method returns the time zone offset for the specified TIMEZONE= system option.

Required Arguments

variable
specifies the variable that will hold the value of the time zone offset.

package
specifies an instance of the TZ package.

UTC-time
specifies the UTC time used to get the time zone offset.

Tip *UTC-time* is a SAS datetime value at UTC. It is stored as the number of seconds since January 1, 1960 00:00:00 at UTC.

time-zone-ID
specifies the time zone ID of the required time zone offset.

See "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Details

UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a datetime value that uses the ISO 8601 basic form *yyyymmdd*T*hhmmss*+|–*hhmm* or the ISO 8601 extended form *yyyy-mm-dd*T*hh*:*mm*:*ss*+|–*hh*:*mm*.

The time zone offset specifies the number of hours and minutes that a time zone is off from the UTC in the form +|–*hhmm* or +|–*hh*:*mm*

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETOFFSET Method" on page 1852

GETTIMEZONEID Method

Returns the current time zone ID.

Syntax

variable=*package*.**GETTIMEZONEID** ( );

Required Arguments

package
specifies an instance of the TZ package.

variable
specifies the variable that will hold the value of the time zone ID of the TZ package instance.

Details

The time zone ID specifies a region or area value that is defined by SAS. For more information about time zone IDs, see *SAS National Language Support (NLS): Reference Guide*.

Example

The following example uses the TZ package to calculate time durations.

```
options timezone='asia/tokyo';  /* TIMEZONE ID of origin */

data _null_;
   method route(package tz origin,
      package tz dest,
      timestamp departure,
      time duration);

      dcl nvarchar(50) tzid dest_tzid;
      dcl nvarchar(8)  tzname dest_tzname home_tzn;
      dcl double dept dur arrival utc;
      dcl double home_dept home_arr;

      declare package tz home();

      utc = origin.toUTCTime(departure);
      dur = TO_DOUBLE(duration);
      arrival = dest.toLocalTime(utc+dur);
      tzid = origin.getTimezoneID();
      tzname = origin.getTimezoneName();
      dest_tzid = dest.getTimezoneID();
      dest_tzname = dest.getTimezoneName();

      home_dept = home.toLocalTime(utc);
      home_arr = home.toLocalTime(utc+dur);
      home_tzn = home.getTimezoneName();

      put 'Time Zone: ' tzid 'to' dest_tzid;
      put 'Departure Time: ' departure datetime.  tzname  '/'
         home_dept datetime. home_tzn;
      put '   Arrial Time: ' arrival datetime.  dest_tzname  '/'
        home_arr datetime. home_tzn;
      put;

   end ;
   method init();

      /* print itinerary */
      declare package tz NRT('Asia/Tokyo');
      declare package tz ORD('America/Chicago');
      declare package tz RDU('America/New_York');
      route(NRT,ORD,timestamp '2014-10-19 10:45:00',time '11:35:00');
      route(ORD,RDU,timestamp '2014-10-19 11:03:00',time '01:56:00');
```

```
      route(RDU,ORD,timestamp '2014-10-25 07:45:00',time '02:02:00');
      route(ORD,NRT,timestamp '2014-10-25 10:50:00',time '12:55:00');

   end;
enddata;
run;
```

The following lines are written to the SAS log.

```
Time Zone:  Asia/Tokyo to America/Chicago
Departure Time:  19OCT14:10:45:00JST / 19OCT14:10:45:00JST
   Arrial Time:  19OCT14:08:20:00CDT / 19OCT14:22:20:00JST

Time Zone:  America/Chicago to America/New_York
Departure Time:  19OCT14:11:03:00CDT / 20OCT14:01:03:00JST
   Arrial Time:  19OCT14:13:59:00EDT / 20OCT14:02:59:00JST

Time Zone:  America/New_York to America/Chicago
Departure Time:  25OCT14:07:45:00EDT / 25OCT14:20:45:00JST
   Arrial Time:  25OCT14:08:47:00CDT / 25OCT14:22:47:00JST

Time Zone:  America/Chicago to Asia/Tokyo
Departure Time:  25OCT14:10:50:00CDT / 26OCT14:00:50:00JST
   Arrial Time:  26OCT14:13:45:00JST / 26OCT14:13:45:00JST
```

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Methods:

- "GETTIMEZONENAME Method" on page 1857

GETTIMEZONENAME Method

Returns the current time zone name.

Syntax

variable=package.**GETTIMEZONENAME** ( );

Required Arguments

package
specifies an instance of the TZ package.

variable
specifies the variable that will hold the value of the time zone name of the TZ package instance.

Details

The time zone name specifies a region or area value that is defined by SAS. For more information about time zone names, see *SAS National Language Support (NLS): Reference Guide*.

Example

See the example in "GETTIMEZONEID Method" on page 1855.

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Methods:

- "GETTIMEZONEID Method" on page 1855

GETUTCTIME Method

Returns the current UTC time.

Syntax

variable=*package*.**GETUTCTIME** ( );

Required Arguments

variable
specifies the variable that will hold the value of the current UTC time.

package
specifies an instance of the TZ package.

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*

Methods:

- "GETLOCALTIME Method" on page 1851

NEW Operator: TZ Package

Constructs an instance of a TZ package.

Note: The escape character (\ ) before the bracket indicates that the bracket is required in the syntax.

Syntax

package-variable = **_NEW_ TZ**([*time-zone-id*]);

Required Argument

package-variable
specifies a name that can reference an instance of the package.

Optional Argument

time-zone-id
specifies a time zone ID.

Default The value specified in the TIMEZONE= system option.

Details

A DS2 package is a collection of variables and methods of which particular instances can be constructed and used in other DS2 programs.

You declare a TZ package variable using the DECLARE PACKAGE statement. After you declare a TZ package variable, use the _NEW_ operator to instantiate an instance of the TZ package and assign the new package instance to the TZ package variable.

```
declare package tz localtz;
localtz = _new_ tz( );
```

As an alternative to the two-step process of using the DECLARE PACKAGE and the _NEW_ operator to declare and instantiate a TZ package, you can declare and

instantiate the package in one step by using the DECLARE PACKAGE statement as a constructor method. Here is the same example using only the DECLARE PACKAGE statement.

```
declare package tz localtz( );
```

Note: Package variables are subject to all variable scoping rules. For more information, see "Packages, Scope, and Lifetime" in *SAS DS2 Programmer's Guide*.

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Package Constructors and Destructors" in *SAS DS2 Programmer's Guide*

Statements:

- "DECLARE PACKAGE Statement: TZ Package" on page 1849

TOISO8601 Method

Converts local time to an ISO8601 string with time zone offset.

Syntax

Form 1: *variable*=*package*.**TOISO8601** (*local-time*);

Form 2: *variable*=*package*.**TOISO8601** (*local-time*, *time-zone-ID*);

Required Arguments

variable
specifies the variable that will hold the value of an ISO8601 string such as '2014-10-10T00:01:02.00+09:00'.

package
specifies an instance of the TZ package.

local-time
specifies the local time to convert. *local-time* can be DOUBLE or TIMESTAMP format.

time-zone-ID
specifies the time zone ID of the required local time. Time zone ID 'UTC' can be used to specify UTC time.

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Methods:

- "TOLOCALTIME Method" on page 1861
- "TOTIMESTAMPZ Method" on page 1862
- "TOUTCTIME Method" on page 1863

TOLOCALTIME Method

Converts UTC time to local time.

Syntax

Form 1: *variable*=*package*.**TOLOCALTIME** (*UTC-time*);

Form 2: *variable*=*package*.**TOLOCALTIME** (*UTC-time*, *time-zone-ID*);

Required Arguments

variable
specifies the variable that will hold the value of the local time that is converted from the specified UTC time.

package
specifies an instance of the TZ package.

UTC-time
specifies the current UTC time in DOUBLE or TIMESTAMP format.

time-zone-ID
specifies the time zone ID of the required local time.

Details

UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a datetime value that uses the ISO 8601 basic form *yyyymmdd*T*hhmmss*+|–*hhmm* or the ISO 8601 extended form *yyyy-mm-dd*T*hh*:*mm*:*ss*+|–*hh*:*mm*.

Example

See the example in "GETTIMEZONEID Method" on page 1855.

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Methods:

- "TOISO8601 Method" on page 1860
- "TOTIMESTAMPZ Method" on page 1862
- "TOUTCTIME Method" on page 1863

TOTIMESTAMPZ Method

Converts local time to a TIMESTAMP string with time zone.

Syntax

variable=*package*.**TOTIMESTAMPZ** (*local-time*[, *time-zone-ID*]);

Required Arguments

variable
specifies the variable that will hold the value of a string such as '2014-10-14 00:01:20 Asia/Tokyo'.

package
specifies an instance of the TZ package.

local-time
specifies the local time to convert. *local-time* can be DOUBLE or TIMESTAMP format.

Optional Argument

time-zone-ID
specifies the time zone ID of the required local time. Time zone ID 'UTC' can be specified to use UTC time.

See Also

- "Using the TZ Package" in *SAS DS2 Programmer's Guide*
- "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS): Reference Guide*

Methods:

- "TOISO8601 Method" on page 1860
- "TOLOCALTIME Method" on page 1861
- "TOUTCTIME Method" on page 1863

TOUTCTIME Method

Converts local time to UTC time.

Syntax

Form 1: *variable*=*package*.**TOUTCTIME** (*local-time*);

Form 2: *variable*=*package*.**TOUTCTIME** (*local-time*, *time-zone-ID*);

Required Arguments

variable
specifies the variable that will hold the value of the current UTC time in DOUBLE or TIMESTAMP format.

package
specifies an instance of the TZ package.

local-time
specifies the local time to convert. *local-time* can be DOUBLE or TIMESTAMP format.

time-zone-ID
specifies the time zone ID of the required local time.

Details

UTC specifies the time at the zero meridian, near Greenwich, England. UTC is a datetime value that uses the ISO 8601 basic form *yyyymmdd*T*hhmmss*+|–*hhmm* or the ISO 8601 extended form *yyyy-mm-dd*T*hh*:*mm*:*ss*+|–*hh*:*mm*.

Example

See the example in “GETTIMEZONEID Method” on page 1855.

See Also

- “Using the TZ Package” in *SAS DS2 Programmer’s Guide*
- “Time Zone IDs and Time Zone Names” in *SAS National Language Support (NLS): Reference Guide*

Methods:

- “TOISO8601 Method” on page 1860
- “TOLOCALTIME Method” on page 1861
- “TOTIMESTAMPZ Method” on page 1862
- “TOUTCTIME Method” on page 1863

PART 4

Appendixes

Appendix 1
Data Type Reference 1867

Appendix 2
DS2 Expressions 1931

Appendix 3
DS2 Example Programs 1955

1866

Appendix 1

Data Type Reference

Data Types for SAS Data Sets ... 1868
Data Types for SPD Engine Data Sets ... 1869
Data Types for SPD Server Tables ... 1871
Data Types for CAS ... 1873
Data Types for Amazon Redshift ... 1875
Data Types for Aster ... 1877
Data Types for DB2 under UNIX and PC Hosts ... 1879
Data Types for Google BigQuery ... 1881
Data Types for Greenplum ... 1883
Data Types for HAWQ ... 1884
Data Types for HDMD ... 1885
Data Types for Hive ... 1887
Data Types for Impala ... 1890
Data Types for JDBC ... 1891
Data Types for MDS ... 1893
Data Types for Microsoft SQL Server ... 1895
Data Types for MongoDB ... 1898
Data Types for MySQL ... 1899
Data Types for Netezza ... 1901
Data Types for ODBC ... 1903
Data Types for Oracle ... 1905
Data Types for PostgreSQL ... 1907
Data Types for Reading from Salesforce ... 1909
Data Types for Writing to Salesforce ... 1912
Data Types for SAP ... 1913

Data Types for SAP HANA 1915
Data Types for SAP IQ 1917
Data Types for Snowflake 1919
Data Types for Spark 1921
Data Types for Teradata 1923
Data Types for Vertica 1925
Data Types for Yellowbrick 1928

Data Types for SAS Data Sets

The following table lists the data type support for a SAS data set.

The BINARY and VARBINARY data types are not supported for data definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source-specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for SAS data sets in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.1 *Mapping of FedSQL Data Types to Data Types Used by SAS Data Sets*

Data Type Definition Keyword [1]	SAS Data Set Data Type	Description	Data Type Returned
BIGINT[2]	DOUBLE	A 64-bit double precision, floating-point number. **Note:** There is potential for loss of precision.	DOUBLE
CHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)
DATE [3]	DOUBLE	A 64-bit double precision, floating-point number. by default, applies the DATE9 SAS format.	DOUBLE
DECIMAL|NUMERIC(*p*,*s*)[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
DOUBLE[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE

Data Type Definition Keyword [1]	SAS Data Set Data Type	Description	Data Type Returned
FLOAT[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
INTEGER[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
NCHAR(*n*)	CHAR(*n*)	A fixed-length character string. By default, sets the encoding to Unicode UTF-8. [4]	CHAR(*n*)
NVARCHAR(*n*)	CHAR(*n*)	A fixed-length character string. By default, sets the encoding to Unicode UTF-8. [4]	CHAR(*n*)
REAL[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
TIME(*p*)[3]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(*p*)[3]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
VARCHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Do not apply date and time SAS formats to a numeric data type. For date and time values, use the DATE, TIME, or TIMESTAMP data types.

3 Because the values are stored as a double precision, floating-point number, you can use the values in arithmetic expressions.

4 UTF-8 is an MBCS encoding. Depending on the operating environment, UTF-8 characters are of varying width, from one to four bytes. The value for *n*, which is the maximum number of multibyte characters to store, is multiplied by the maximum length for the operating environment. Note that when you are transcoding, such as from UTF-8 to Wlatin2, the variable lengths (in bytes) might not be sufficient to hold the values, and the result is character data truncation.

Data Types for SPD Engine Data Sets

The following table lists the data type support for an SPD Engine data set.

The BINARY, DECIMAL, NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source-specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for SPD Engine data sets in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.2 *Mapping of FedSQL Data Types to Data Types Used by SPD Engine Data Sets*

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
BIGINT[1]	DOUBLE	A 64-bit double precision, floating-point number. **Note:** There is potential for loss of precision.	DOUBLE
CHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)
DATE [2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATE9 SAS format.	DOUBLE
DOUBLE[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
FLOAT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
INTEGER[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
REAL[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
TIME(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
VARCHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
		Note: Cannot contain ANSI SQL null values.	

1 Do not apply date and time SAS formats to a numeric data type. For date and time values, use DATE, TIME, or TIMESTAMP data types.

2 Because the values are stored as double precision, floating-point numbers, you can use the values in arithmetic expressions.

Data Types for SPD Server Tables

The following table lists the data type support for an SPD Server table.

The BINARY, DECIMAL, NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not supported for data type definition.

For some data type definitions, the data type is mapped to CHAR, which is a Base SAS character data type, or DOUBLE, which is a Base SAS numeric data type. For data source-specific information about the SAS numeric and SAS character data types, see *SAS Language Reference: Concepts*.

Note: This data source is not supported on the CAS server.

Table A18.3 *Mapping of FedSQL Data Types to Data Types Used by SPD Server Tables*

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
BIGINT[1]	DOUBLE	A 64-bit double precision, floating-point number. **Note:** There is potential for loss of precision.	DOUBLE
CHAR(*n*)	CHAR(*n*)	A fixed-length character string. **Note:** Cannot contain ANSI SQL null values.	CHAR(*n*)
DATE [2]	DOUBLE	A 64-bit double precision, floating-point number. By	DOUBLE

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
		default, applies the DATE9 SAS format.	
DOUBLE[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
FLOAT [1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
INTEGER[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
REAL[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
TIME(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the TIME8 SAS format.	DOUBLE
TIMESTAMP(*p*)[2]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATETIME19.2 SAS format.	DOUBLE
TINYINT[1]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
VARCHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)

Data Type Definition Keyword	SPD Data Set Data Type	Description	Data Type Returned
		Note: Cannot contain ANSI SQL null values.	

1 Do not apply date and time SAS formats to a numeric data type. For date and time values, use DATE, TIME, or TIMESTAMP data types.

2 Because the values are stored as double precision, floating-point numbers, you can use the values in arithmetic expressions.

Data Types for CAS

The following table lists the data type support for a CAS table.

The BINARY, DECIMAL/NUMERIC, and REAL data types are not supported.

In order to read or update a CAS table that contains a BINARY column with DS2, you must drop the BINARY column from the table first. Beginning with SAS Viya 3.5, DS2 returns an error when it encounters a BINARY column in a CAS table. In earlier SAS Viya releases, no error was reported when reading BINARY columns; however, the data might be corrupted when it is read.

Table A18.4 *Mapping of FedSQL Data Types to Data Types Used by CAS Tables*

Data Type Definition Keyword[1]	CAS Data Type	Description	Data Type Returned
BIGINT[2]	INT64	A large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Integer data types do not store decimal values; fractional portions are discarded. A DS2 value that corresponds to the most negative possible BIGINT (INT64) value in a BIGINT column in a CAS table is treated as a SAS missing or null value.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string. If a string value is less than *n* in length, the value is blank-padded to *n*. [4] **Note:** ANSI SQL null values in a string are converted to SAS missing values.	CHAR(*n*) [5]

Data Type Definition Keyword[1]	CAS Data Type	Description	Data Type Returned
DATE [3]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATE9. SAS format.	DOUBLE
DOUBLE[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
FLOAT[2]	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
INTEGER[2]	INT32	A regular signed, exact whole number, with a precision of 10 digits. The range of integers is -2,147,483,648 to 2,147,483,647. Integer data types do not store decimal values; fractional portions are discarded. A DS2 value that corresponds to the most negative possible INTEGER (INT32) value in an INTEGER column in a CAS table is treated as a SAS missing or null value.	INTEGER
NCHAR(*n*)	CHAR(*n*)	A fixed-length national character string. If a string value is less than *n* in length, the value is blank-padded to *n*. [4] **Note:** ANSI SQL null values in a string are converted to SAS missing values.	CHAR(*n*)
NVARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string. *n* is the maximum length of the string that can be stored.[6] **Note:** ANSI SQL null values in a string are converted to SAS missing values.	VARCHAR(*n*)
SMALLINT[2]	INT32	A small signed, exact whole number.	INTEGER
TIME(*p*)[3]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the TIME8. SAS format.	DOUBLE
TIMESTAMP(*p*)[3]	DOUBLE	A 64-bit double precision, floating-point number. By default, applies the DATETIME25.6 SAS format.	DOUBLE
TINYINT[2]	INT32	A very small signed, exact whole number.	INTEGER
VARBINARY[7]	VARBINARY	Varying-length binary opaque data. This data type is used to store image data, audio, documents, and other unstructured data.	VARBINARY

Data Type Definition Keyword[1]	CAS Data Type	Description	Data Type Returned
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string. *n* is the maximum length of the string that can be stored.[6] **Note:** ANSI SQL null values in a string are converted to SAS missing values.	VARCHAR(*n*)[5]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Do not apply date and time SAS formats to a numeric data type. For date and time values, use the DATE, TIME, or TIMESTAMP data types.

3 Because the values are stored as a double precision, floating-point number, you can use the values in arithmetic expressions.

4 For a DS2 CHAR and NCHAR character data types, the value for *n* is the number of characters to store. For a CAS CHAR and NCHAR character types, the value for *n* is the number of bytes to store. CAS tables use the UTF-8 character set. UTF-8 is a multibyte character set encoding. UTF-8 characters are of varying width, from 1 to 4 bytes. When a DS2 character value is saved to a CAS character column, the string data is truncated if the value requires more than *n* bytes in its UTF-8 encoded representation. When a DS2 character value is used to read a CAS character column, the string data is truncated if the value requires more than *n* characters in the active session encoding.

5 DS2 character types use the session encoding by default. This data type enables you to specify an alternate character set encoding when defining columns. When the DS2 data type specifies an encoding other than UTF-8 in a DS2 CAS action, the data is transcoded to and from UTF-8 when data is written to or read from CAS.

6 For DS2 and CAS NVARCHAR and VARCHAR data types, the value for *n* is the number of characters to store.

7 Support for the VARBINARY data type begins with SAS Viya 3.5. In earlier SAS Viya releases, DS2 does not return an error when reading a VARBINARY column; however, the data is incorrectly treated as character data.

Data Types for Amazon Redshift

The following table lists the data type support for an Amazon Redshift database.

The BINARY, TINYINT, and VARBINARY data types are not supported for data type definition.

For data source-specific information about Redshift data types, see the Amazon Redshift database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Amazon Redshift support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.5 *Mapping of FedSQL Data Types to Data Types Used by Amazon Redshift*

Data Type Definition Keyword [1]	Amazon Redshift Data Type	Description	Data Type Returned
BIGINT	BIGINT	A signed eight-byte integer.	BIGINT
[2]	BOOLEAN	A logical Boolean (true/false) value.	[2]
CHAR(*n*)	CHAR	A fixed-length character string.	CHAR(*n*)
DATE	DATE	Date value.	DATE
DECIMAL|NUMERIC(*p*,*s*)	DECIMAL, NUMERIC	A signed, fixed-point decimal number.	DECIMAL|NUMERIC(*p*,*s*)[4]
DOUBLE	DOUBLE PRECISION	A signed, double precision, floating-point number.	DOUBLE
FLOAT	FLOAT	A signed, double precision, floating-point number.	DOUBLE
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
NCHAR	NCHAR	A fixed-length Unicode character string.	CHAR
NVARCHAR	NVARCHAR	A varying-length Unicode character string.	VARCHAR
REAL	REAL	A signed, single precision floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	[3]	A time value.	TIMESTAMP
TIMESTAMP(*p*)	TIMESTAMP	A date and time value.	TIMESTAMP(*p*)

Data Type Definition Keyword [1]	Amazon Redshift Data Type	Description	Data Type Returned
[2]	TIMESTAMPZ	A date and time value with time zone.	[2]
VARCHAR(*n*)	VARCHAR	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Amazon Redshift data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 Amazon Redshift does not support the TIME(*p*) data type. When you define a column of type TIME, a column of type TIMESTAMP is created instead.

4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Aster

The following table lists the data type support for an Aster database.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about Aster database data types, see the Aster database documentation.

Note: This data source is not supported on the CAS server.

Table A18.6 *Mapping of FedSQL Data Types to Data Types Used by Aster*

Data Type Definition Keyword	Aster Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	BYTEA	A varying-length binary string.	BINARY(*n*)
[1]	BOOL	One byte integral data type that can contain values 0, 1, or NULL.	[1]

Data Type Definition Keyword	Aster Data Type	Description	Data Type Returned
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[2]
DOUBLE	FLOAT(*p*)	A signed, double precision, floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A signed, double precision, floating-point number.	DOUBLE
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
REAL	REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	BYTEA	A varying-length binary string.	VARBINARY(*n*)
[1]	TEXT	A varying-length large character string.	VARCHAR(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The Aster data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for DB2 under UNIX and PC Hosts

The following table lists the data type support for a DB2 database under UNIX and PC hosts.

The NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about the DB2 database data types, see the DB2 database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about DB2 support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.7 *Mapping of FedSQL Data Types to Data Types Used by DB2 under UNIX and PC Hosts*

Data Type Definition Keyword[1]	DB2 Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	CHAR(*n*) FOR BIT DATA	A fixed-length binary string.	BINARY(*n*)
[2]	BLOB(*n*[K|M|G])	A varying-length binary large object string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
[2]	CLOB(*n*[K|M|G])	A varying-length character large object string.	[2]
DATE	DATE	A date value.	DATE
[2]	DBCLOB(*n*[K|M|G])	A varying-length double-byte character large object.	[2]

Data Type Definition Keyword[1]	DB2 Data Type	Description	Data Type Returned
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL| NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A signed, double precision, floating-point number.	DOUBLE
[2]	GRAPHIC(*n*)	A fixed-length graphic string.	[2]
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	LONG VARCHAR [FOR BIT DATA]	A varying-length character or binary string.	[2]
[2]	LONG VARGRAPHIC(*n*)	A varying-length graphic string.	[2]
REAL	REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	VARCHAR(*n*) FOR BIT DATA	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)
[2]	VARGRAPHIC(*n*)	A varying-length graphic string	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The DB2 data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Google BigQuery

The following table lists the data type support for a Google BigQuery project. SAS/ACCESS to Google BigQuery uses the GoLang BiqQuery API to communicate with Google BigQuery.

The Google BigQuery ARRAY, STRUCT, and GEOGRAPHY data types are not supported.

For data source-specific information about Google BigQuery data types, see the Google BigQuery documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Google BigQuery support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.8 *Mapping of FedSQL Data Types to Data Types Used by Google BigQuery*

Data Type Definition Keyword	BigQuery Data Type	BigQuery Description	Data Type Returned
BIGINT	INT64	A large signed, exact whole number, with a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
[1]	BIGNUMERIC(p,s)	A double precision, floating-point number.	DOUBLE[5, 6]
BINARY	BYTES	A varying-length binary data.	VARBINARY[2]
[1]	BOOL	A Boolean value (True or False) that is case insensitive..	[1]
CHAR(*n*)	STRING	A varying-length Unicode character string.	VARCHAR[3]
DATE	DATE	A date value from 0001-01-01 to 9999-12-31.	DATE
DOUBLE	FLOAT64	Double precision, floating-point number.	DOUBLE

Data Type Definition Keyword	BigQuery Data Type	BigQuery Description	Data Type Returned
FLOAT	FLOAT64	Double precision, floating-point number.	DOUBLE
INTEGER	INT64	A regular signed, exact whole number.	BIGINT
NCHAR	STRING	A fixed-length Unicode character string.	VARCHAR
NUMERIC(p,s)	NUMERIC(p,s)	A double precision, floating-point number.	DOUBLE[5, 6]
NVARCHAR	STRING	A varying-length Unicode character string.	VARCHAR
SMALLINT	INT64	A small signed, exact whole number	BIGINT
TIME(*p*)	TIME	A time value in hours, minutes, and seconds, in the range 00:00:00 to 23:59:59.999999.	TIME(*p*)
[1]	TIMESTAMP	A date and time value.	[1]
TIMESTAMP(*p*)[4]	DATETIME	A date and time value with microsecond precision, independent of time zone. Supported values are 0001-01-01 00:00:00 to 9999-12-31 23:59:59.999999.	TIMESTAMP(*p*)
TINYINT	INT64	A very small signed, exact whole number.	BIGINT
VARBINARY(*n*)	BYTES	A varying-length binary string.	VARBINARY [2]
VARCHAR(*n*)	STRING	A varying-length Unicode character string.	VARCHAR[3]

1 The BigQuery data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 Length is determined by the MAX_BINARY_LEN= LIBNAME option. If MAX_BINARY_LEN= is not set, the default length is 2,000 bytes.

3 The default length is 16,384 characters unless the MAX_CHAR_LEN= LIBNAME option or SCANSTRINGCOLUMNS table option is set.

4 Beginning with the April 2021 release of SAS/ACCESS software for SAS 9.4M7 and SAS Viya 3.5, the TIMESTAMP(*p*) data type maps to the Google BigQuery DATETIME data type. In earlier software releases, TIMESTAMP(*p*) maps to the BigQuery TIMESTAMP(*p*) data type.

5 Beginning with the September 2024 update to SAS/ACCESS Interface to Google BigQuery on SAS 9.4M7, Google BigQuery NUMERIC(p,s) and BIGNUMERIC(p,s) columns are treated as DOUBLEs, which map to the Google BigQuery FLOAT64 type. The FETCH_NUMERIC_TYPE= table option enables you to request that these types be treated as standard NUMERIC(p,s) types.

6 When FETCH_NUMERIC_TYPE=NUMERIC is set, the NUMERIC(p,s) and BIGNUMERIC(p,s) types are supported in requests that can be fully passed down to the data source. NUMERIC(p,s) and BIGNUMERIC(p,s) columns in requests

that cannot be fully passed down to the data source are handled as DOUBLE. For more information, see "What Are the Data Types?" in *SAS DS2 Programmer's Guide*.

Data Types for Greenplum

The following table lists the data type support for a Greenplum database.

The BINARY, NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about Greenplum data types, see the Greenplum database documentation.

Note: This data source is not supported on the CAS server.

Table A18.9 *Mapping of FedSQL Data Types to Data Types Used by Greenplum*

Data Type Definition Keyword[1]	Greenplum Data Type	Description	Data Type Returned
BIGINT	INT8	A large signed, exact whole number.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL|NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL|NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	A floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A floating-point number.	DOUBLE
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
REAL	REAL	A floating-point number.	REAL
SMALLINT	INT8	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value in hours, minutes, and seconds.	TIME(*p*)[2]

Data Type Definition Keyword[1]	Greenplum Data Type	Description	Data Type Returned
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	BYTEA	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Due to the ODBC-style interface that is used to communicate with the Greenplum server, fractional seconds are lost in the data transfer from server to client.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for HAWQ

The following table lists the data type support for a HAWQ database.

The BINARY, NCHAR, NVARCHAR, and TINYINT data types are not supported for data type definition.

For data source-specific information about HAWQ data types, see the HAWQ database documentation.

Note: This data source is not supported on the CAS server.

Table A18.10 *Mapping of FedSQL Data Types to Data Types Used by HAWQ*

Data Type Definition Keyword [1]	HAWQ Data Type	Description	Data Type Returned
BIGINT	INT8	A large signed, exact whole number.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]

Data Type Definition Keyword [1]	HAWQ Data Type	Description	Data Type Returned
DOUBLE	DOUBLE	A floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A floating-point number.	DOUBLE
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
REAL	REAL	A floating-point number.	REAL
SMALLINT	INT8	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value in hours, minutes, and seconds.	TIME(*p*)[2]
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	BYTEA	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Due to the ODBC-style interface that is used to communicate with the HAWQ server, fractional seconds are lost in the data transfer from server to client.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for HDMD

The following table lists the data type support for HDMD.

The BINARY, NCHAR, and NVARCHAR data types are not supported for data type definition.

Note: This data source is not supported on the CAS server.

Table A18.11 *Mapping of FedSQL Data Types to Data Types Used by HDMD Tables*

Data Type Definition Keyword	HDMD Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number, with a precision of 19 digits. The range of integers is -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	Date value.	DATE
DECIMAL/ NUMERIC(*p*,*s*)	DECIMAL	A fixed-point decimal number, with 38 digits precision.	DECIMAL(*p*,*s*)[1]
DOUBLE	DOUBLE	A signed, approximate, double-precision, floating-point number. **Note:** Supports SAS missing values	DOUBLE
FLOAT	DOUBLE	A signed, approximate, double-precision, floating-point number. **Note:** Supports SAS missing values	DOUBLE
INTEGER	INTEGER	A regular size signed, exact whole number, with a precision of 10 digits. The range of integers is -2,147,483,647 to 2,147,483,647.	INTEGER
REAL	REAL	A signed, approximate, single-precision, floating-point number. **Note:** Supports SAS missing values	REAL

Data Type Definition Keyword	HDMD Data Type	Description	Data Type Returned
SMALLINT	SMALLINT	A small signed, exact whole number, with a precision of five digits. The range of integers is -32,767 to 32,767.	SMALLINT
TIME(*p*)	TIME[(*p*)]	A time value with optional precision.	TIME[(*p*)]
TIMESTAMP(*p*)	TIMESTAMP[(*p*)]	A date and time value with optional precision.	TIMESTAMP[(*p*)]
TINYINT	TINYINT	A very small signed, exact whole number, with a precision of three digits. The range of integers is -127 to 127.	TINYINT
VARCHAR(*n*)	VARCHAR(n)	A varying-length character string data.	VARCHAR(*n*)

1 The DECIMAL data type is handled as double precision.

Data Types for Hive

The following table lists the data type support for Hive. Hive database requirements can vary, depending on your SAS software release. See the Support for Hadoop for information about SAS 9.4 database requirements.

The NCHAR and NVARCHAR data types are not available for data definition. The VARBINARY data type is available for data definition beginning with SAS 9.4M3.

The Hive complex data types can be read, beginning with SAS 9.4M3.

For data source-specific information about Hive data types, see the Hive database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Hive support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.12 *Mapping of FedSQL Data Types to Data Types Used by Hive*

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
[1, 6]	ARRAY<*data-type*>	An array of integers (indexable lists).	STRING[10]
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.	BIGINT
BINARY(*n*)	BINARY[2]	A varying-length binary string up to 32K.	BINARY
[1]	BOOLEAN	A textual true or false value.	TINYINT
CHAR(*n*)	CHAR(n)[2]	A character string up to 255 characters. If you specify CHAR with a value greater than 255 characters, the column is created as VARCHAR(n) instead.	CHAR
DATE	DATE[3, 4]	An ANSI SQL date type.	DATE
DECIMAL/ NUMERIC(*p,s*)	DECIMAL[2]	A fixed-point decimal number, with 38 digits precision.	DECIMAL(*p,s*)[5]
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
FLOAT	FLOAT	An eight-byte, double-precision floating-point number.	DOUBLE
INTEGER	INTEGER	A regular size signed, exact whole number, with a precision of 10 digits. The range of integers is -2,147,483,647 to 2,147,483,647.	INTEGER
[1, 6]	MAP<*primitive-type*, *data-type*>	An associative array of key-value pairs.	STRING[10]
REAL	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,767 to 32,767.	SMALLINT
[1, 6]	STRING	A varying-length character string.	VARCHAR(n)[8]

Data Type Definition Keyword	Hive Data Type	Description	Data Type Returned
[1, 6]	STRUCT<*col-name*: *data-type*>	A structure with established column elements and data types. Column elements and data types are mapped using a double-dot (:) notation.	STRING[10]
TIME(*p*)	[9]	A time value.	STRING
TIMESTAMP(*p*)	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP[(*p*)]
TINYINT	TINYINT	A signed one-byte integer, from -127 to 127.	TINYINT
[1, 6]	UNION<*data-type*, *data-type-n*>	A type that can hold one of several specified data types.	STRING[10]
VARCHAR(*n*)[7]	VARCHAR(*n*)	A varying-length character string.	STRING
VARBINARY	BINARY[2]	A varying-length binary string up to 32K.	BINARY

1 The Hive data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 Full support for this data type is available in Hive 0.13 and later. In Hadoop environments that use earlier Hive versions (which do not support the CHAR and DECIMAL types), columns defined as CHAR are mapped to VARCHAR. Columns that are defined as DECIMAL are not supported. In Hadoop environments that use Hive versions earlier than Hive 0.13, BINARY columns can be created but not retrieved.

3 Full support for this data type is available in Hive 0.12 and later. In Hadoop environments that use earlier Hive versions (which do not support the DATE type), when the DATE data type is used for data definition, the DATE type is mapped to a STRING column with SASFMT TableProperties. Any SASFMT TableProperties that are defined on STRING columns are applied when reading Hive, allowing the STRING columns to be treated as DATE columns. For more information about SASFMT TableProperties, see "SAS Table Properties for Hive and HADOOP" in *SAS/ACCESS for Relational Databases: Reference*.

4 The supported date values are between October 15, 1582 and December 31, 9999. Date values containing years earlier than 1582 return an error. Date values later than 9999 are read back as null values.

5 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

6 String and Hive complex types are loaded as VARCHAR(*n*), whose length is determined by the DBMAX_TEXT= data source connection option.

7 VARCHAR columns with values greater than 65,535 characters are created as STRING.

8 SASFMT TableProperties are applied when reading STRING columns.

9 Hive does not support the TIME(p) data type. When data is read from Hive, STRING columns that have SASFMT TableProperties defined that specify the SAS TIME8. format are converted to the TIME(*p*) data type. When the TIME type is used for data definition, it is mapped to a STRING column with SASFMT TableProperties. Fractional seconds are not preserved. For more information about SASFMT TableProperties, see "SAS Table Properties for Hive and HADOOP" in *SAS/ACCESS for Relational Databases: Reference*.

10 The complex types ARRAY, MAP, STRUCT, and UNION are read as their STRING representation of the underlying complex type. ARRAY values are read back within brackets, for example: [1, 2, 4]. STRUCT and MAP values are read back within braces, for example: {"firstname":"robert","nickname":"bob"}.

Data Types for Impala

The following table lists the data type support for Impala. Impala version 2.0 and later are supported, running on CDH 5.1 and later.

The BINARY, DECIMAL(*p*,*s*)/NUMERIC, NCHAR, NVARCHAR, and VARBINARY data types are not available for data definition.

For data source-specific information about Impala data types, see the vendor documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Impala support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.13 *Mapping of FedSQL Data Types to Data Types Used by Impala*

Data Type Definition Keyword	Impala Data Type	Description	Data Type Returned
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.	BIGINT
CHAR(*n*)	CHAR[1]	A fixed-length character string up to 255 characters.	CHAR
DATE	[2]	An ANSI SQL date type.	TIMESTAMP
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
FLOAT	FLOAT	An eight-byte, double-precision floating-point number.	DOUBLE
INTEGER	INT	A signed four-byte integer, from -2,147,483,647 to 2,147,483,647.	INTEGER

Data Type Definition Keyword	Impala Data Type	Description	Data Type Returned
REAL	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,767 to 32,767.	SMALLINT
TIME	[2]	A time value.	TIMESTAMP
TIMESTAMP	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP
TINYINT	TINYINT	A signed one-byte integer, from -127 to 127.	TINYINT
VARCHAR(*n*)	VARCHAR[1]	A varying-length character string.	VARCHAR

1 Support for this data type is available in CDH 5.2 and later. In environments that use earlier CDH versions, which do not support the CHAR and VARCHAR types, columns defined as CHAR or VARCHAR are mapped to STRING.

2 Impala does not support this data type. When a DATE or TIME column is defined, it is created as a column of type TIMESTAMP.

Data Types for JDBC

The following table lists the data type support for JDBC. JDBC 4.1 and later is supported.

For data source-specific information about JDBC data types, see the database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for JDBC databases in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.14 *Mapping of FedSQL Data Types to Data Types Supported by JDBC*

Data Type Definition Keyword	JDBC SQL Identifier	Description	Data Type Returned
BIGINT	SQL_BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	SQL_BINARY	A fixed-length binary string.	BINARY(*n*)
[1]	SQL_BIT	Single bit binary data.	[1]
CHAR(*n*)[2]	SQL_CHAR	A fixed-length character string.	CHAR(*n*)
DATE	SQL_TYPE_DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	SQL_DECIMAL| SQL_NUMERIC	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	SQL_DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT	SQL_FLOAT	A signed, double precision, floating-point number.	DOUBLE
[1]	SQL_GUID	A globally unique identifier.	[1]
INTEGER	SQL_INTEGER	A regular signed, exact whole number.	INTEGER
[1]	SQL_INTERVAL	Intervals between two years, months, days, dates or times.	[1]
[1]	SQL_LONGVARBINARY	A varying-length binary string.	[1]
[1]	SQL_LONGVARCHAR	A varying-length Unicode character string.	[1]
NCHAR(*n*)	SQL_WCHAR	A fixed-length Unicode character string.	NCHAR(*n*)

Data Type Definition Keyword	JDBC SQL Identifier	Description	Data Type Returned
NVARCHAR(*n*)	SQL_WVARCHAR	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	SQL_REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SQL_SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	SQL_TYPE_TIME	A time value.	TIME(*p*)
TIMESTAMP(*p*)	SQL_TYPE_TIMESTAMP	A date and time value.	TIMESTAMP(*p*)
TINYINT	SQL_TINYINT	A very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	SQL_VARBINARY	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)[2]	SQL_VARCHAR	A varying-length character string.	VARCHAR(*n*)
[1]	SQL_WLONGVARCHAR	A varying-length Unicode character string.	[1]

1 The JDBC SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 When you use the CHAR(n) or VARCHAR(n) data type to store multibyte data, you must specify the encoding in the CLIENT_ENCODING= connection option. Or, to avoid having to set the encoding, use the NCHAR or NVARCHAR data types for multibyte data instead.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for MDS

The following table lists the data type support for the in-memory database.

Note: This data source is not supported on the CAS server.

Table A18.15 Mapping of FedSQL Data Types to Data Types Used by MDS Tables

Data Type Definition Keyword [2]	MDS Data Type	Description	Data Type Returned
BIGINT	BIGINT	A 64-bit, signed integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary data.	BINARY(*n*)
CHAR(*n*)	CHAR(*n*)	A fixed-length character string data.	CHAR(*n*)
DATE	DATE	Date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	Precision scale numeric.	DECIMAL/ NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	An eight-byte IEEE floating-point value. **Note:** Supports ANSI SQL null values	DOUBLE
FLOAT	DOUBLE	An eight-byte IEEE floating-point value. **Note:** Supports ANSI SQL null values	DOUBLE
INTEGER	INTEGER	32-bit, signed integer.	INTEGER
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	DOUBLE	An eight-byte IEEE floating-point value. **Note:** Supports ANSI SQL null values	DOUBLE
SMALLINT	INTEGER	32-bit, signed integer.	INTEGER
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)

Data Type Definition Keyword [2]	MDS Data Type	Description	Data Type Returned
TINYINT	INTEGER	32-bit, signed integer.	INTEGER
[1]	UBIGINT	A 64-bit, unsigned integer.	BIGINT
[1]	UINTEGER	32-bit, unsigned integer.	INTEGER
VARCHAR(*n*)	VARCHAR(n)	A varying-length character string data.	VARCHAR(*n*)
VARBINARY(n)	VARBINARY(n)	A varying-length binary data.	VARBINARY(n)

1 The MDS SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Microsoft SQL Server

The following table lists the data type support for a Microsoft SQL Server database. SAS supports SQL Server 2008 and later.

For data source-specific information about the Microsoft SQL Server data types, see the Microsoft SQL Server database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Microsoft SQL Server support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.16 *Mapping of FedSQL Data Types to Data Types Used by Microsoft SQL Server*

Data Type Definition Keyword[1]	SQL Server SQL Identifier	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT

Data Type Definition Keyword[1]	SQL Server SQL Identifier	Description	Data Type Returned
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[2]	BIT	Single bit binary data.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
[2]	DATETIME	Date combined with time of day with fractional seconds that is based on a 24-hour clock.	TIMESTAMP(*p*)
[2]	DATETIME2(*p*)	An extension of DATETIME with larger date range, a larger default fractional precision, and optional user-specified precision.	TIMESTAMP(*p*)
[2]	DATETIMEOFFSET(*p*)	Date/time value with time zone awareness.	[2]
DECIMAL|NUMERIC(*p*,*s*)	DECIMAL|NUMERIC(*p*,*s*)	A signed, fixed precision and scale numbers.	DECIMAL|NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT	FLOAT	A signed, double precision, floating-point number.	DOUBLE
[2]	IMAGE	A varying-length binary string.	[2]
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	MONEY	An eight-byte currency value.	DECIMAL(19,4)[3]

Data Type Definition Keyword[1]	SQL Server SQL Identifier	Description	Data Type Returned
NCHAR(*n*)	NCHAR	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	REAL	A signed, single precision, floating-point number.	REAL
[2]	SMALLDATETIME	A date and time value without fractional seconds.	TIMESTAMP(*p*)
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
[2]	SMALLMONEY	A four-byte currency value.	DECIMAL(10,4)[3]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	BINARY(8)	A date and time value.	TIMESTAMP(*p*)
TINYINT	TINYINT	A very small signed, exact whole number.	TINYINT
[2]	UNIQUEIDENTIFIER	A globally unique identifier.	CHAR(36)
VARBINARY(*n*)	VARBINARY	A varying-length binary string.	VARBINARY(*n*)
[2]	VARBINARY(MAX)	A varying-length binary string. Maximum 2GB	[2]
VARCHAR(*n*)	VARCHAR(*n*), TEXT	A varying-length character string.[2]	VARCHAR(*n*)
[2]	VARCHAR(MAX)	A varying-length Unicode character string. Maximum 2GB	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The

values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Microsoft SQL Server data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for MongoDB

The following table lists the data type support for a MongoDB non-relational database. MongoDB 3.6 and later is supported.

The MongoDB types Timestamp, Min key, and Max key are internal to MongoDB. Therefore, they are not read by SAS.

The FedSQL BINARY, DATE, DECIMAL, FLOAT, NCHAR, NVARCHAR, REAL, SMALLINT, TINYINT, and VARBINARY data types are not supported for data definition.

For data source-specific information about MongoDB data types, see the MongoDB documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about MongoDB support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.17 *Mapping of FedSQL Data Types to Data Types Used by MongoDB*

Data Type Definition Keyword	MongoDB Data Type	Description	Data Type Returned
[2]	Array	Array.	[4]
BIGINT	int	A regular 64-bit exact whole number.	BIGINT
[1]	BinData	Binary data.	VARCHAR[5]
[1]	Bool	Boolean value.	INTEGER
CHAR(*n*)	String	A fixed-length UTF-8 character string.	CHAR(*n*)
[1]	Date	A date and time value.	TIMESTAMP
[5]	Decimal	Decimal128.	DOUBLE

Data Type Definition Keyword	MongoDB Data Type	Description	Data Type Returned
DOUBLE	Double	A signed double-precision, floating-point number.	DOUBLE
INTEGER	Int	A regular 32-bit exact whole number.	INTEGER
[1]	Javascript	Javascript string.	[4]
[1]	Long	A large signed, exact whole number.	BIGINT
[1]	Null	Null value.	VARCHAR
[1]	Object	JSON object.	[4]
[1]	ObjectId	Object identifier.	VARCHAR
[1]	Regex	A regular expression string.	VARCHAR
TIMESTAMP	Date	A date and time value.	TIMESTAMP
VARCHAR	String	A varying-length UTF-8 character string.	VARCHAR, CHAR(*n*)[3]

1 The MongoDB data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 Arrays can be created as child tables. For more information, see information about MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*.

3 You can customize the default MongoDB schema to use CHAR(n). For more information, see the documentation for MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*.

4 The array and object types are modeled as separate child tables. For more information, see the documentation for MongoDB in *SAS/ACCESS for Nonrelational Databases: Reference*. The Javascript type is mapped to VARCHAR.

5 Contains a hexadecimal string representation of binary data.

Data Types for MySQL

The following table lists the data type support for a MySQL database. Support for MySQL begins with MySQL 5.1 and, beginning in SAS Viya 3.5, includes MySQL 8 server.

The NCHAR, NVARCHAR, REAL, and VARBINARY data types are not supported for data type definition.

For data source-specific information about MySQL data types, see the MySQL database documentation.

Note: This data source is not supported on the CAS server.

Table A18.18 *Mapping of FedSQL Data Types to Data Types Used by MySQL*

Data Type Definition Keyword[1]	MySQL Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[2]	BLOB	A varying-length binary large object string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
[2]	DATETIME	A date and time value.	[2]
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL(*p*,*s*)[3]
DOUBLE	DOUBLE	A signed, double precision, floating-point number.	DOUBLE
[2]	ENUM(*values*)	A character value from a list of allowed values.	[2]
FLOAT	FLOAT(*p*)	A signed, double precision, floating-point number.	DOUBLE
INTEGER	INT	A regular signed, exact whole number.	INTEGER
[2]	JSON	A varying-length character string in JSON format.	VARCHAR
[2]	LONGBLOB	A varying-length binary data.	[2]
[2]	LONGTEXT	A varying-length character string.	[2]
[2]	MEDIUMBLOB	A varying-length binary data.	[2]
[2]	MEDIUMINT	A regular signed, exact whole number.	[2]
[2]	MEDIUMTEXT	A varying-length character string.	[2]

Data Type Definition Keyword[1]	MySQL Data Type	Description	Data Type Returned
[2]	SET(*values*)	Character values from a list of allowed values.	[2]
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
[2]	TEXT	A varying-length text data.	[2]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
[2]	TINYBLOB	A varying-length binary large object string.	[2]
TINYINT	TINYINT	A very small signed, exact whole number.	TINYINT
[2]	TINYTEXT	A varying-length text data.	[2]
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The MySQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Netezza

The following table lists the data type support for a Netezza database.

The BINARY and VARBINARY data types are not supported for data type definition.

For data source-specific information about Netezza data types, see the Netezza database documentation.

Note: This data source is not supported on the CAS server.

Table A18.19 Mapping of FedSQL Data Types to Data Types Used by Netezza

Data Type Definition Keyword[1]	Netezza Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT
CHAR(*n*)	CHAR(*n*)	A fixed-length character string data.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p,s*)	DECIMAL(*p,s*)	A fixed-point decimal number.	DECIMAL(*p,s*)[3]
DOUBLE	DOUBLE	A floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A 64-bit double precision, floating-point number.	DOUBLE
INTEGER	INTEGER	A large integer.	INTEGER
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	REAL	A floating-point number.	REAL
SMALLINT	SMALLINT	A small integer.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)[2]
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	BYTEINT	A tiny integer.	TINYINT
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string data.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 Due to the ODBC-style interface that is used to communicate with the Netezza server, fractional seconds are lost in the data transfer from server to client.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for ODBC

The following table lists the data type support for a data source that is compliant with ODBC.

For data source-specific information about ODBC SQL data types, see the specific ODBC data source documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about support for ODBC databases in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.20 *Mapping of FedSQL Data Types to Data Types Supported by ODBC*

Data Type Definition Keyword[1]	ODBC SQL Identifier	Description	Data Type Returned
BIGINT	SQL_BIGINT	A large signed, exact whole number.	BIGINT
BINARY(*n*)	SQL_BINARY	A fixed-length binary string.	BINARY(*n*)
[2]	SQL_BIT	Single bit binary data.	[2]
CHAR(*n*)[3]	SQL_CHAR	A fixed-length character string.	CHAR(*n*)
DATE	SQL_TYPE_DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	SQL_DECIMAL| SQL_NUMERIC	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[4]
DOUBLE	SQL_DOUBLE	A signed, double precision, floating-point number.	DOUBLE
FLOAT	SQL_FLOAT	A signed, double precision, floating-point number.	DOUBLE
[2]	SQL_GUID	A globally unique identifier.	[2]

Data Type Definition Keyword[1]	ODBC SQL Identifier	Description	Data Type Returned
INTEGER	SQL_INTEGER	A regular signed, exact whole number.	INTEGER
[2]	SQL_INTERVAL	Intervals between two years, months, days, dates or times.	[2]
[2]	SQL_LONGVARBINARY	A varying-length binary string.	[2]
[2]	SQL_LONGVARCHAR	A varying-length Unicode character string.	[2]
NCHAR(*n*)	SQL_WCHAR	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	SQL_WVARCHAR	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	SQL_REAL	A signed, single precision, floating-point number.	REAL
SMALLINT	SQL_SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	SQL_TYPE_TIME	A time value.	TIME(*p*)
TIMESTAMP(*p*)	SQL_TYPE_TIMESTAMP	A date and time value.	TIMESTAMP(*p*)
TINYINT	SQL_TINYINT	A very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	SQL_VARBINARY	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)[3]	SQL_VARCHAR	A varying-length character string.	VARCHAR(*n*)
[2]	SQL_WLONGVARCHAR	A varying-length Unicode character string.	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The

values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The ODBC SQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 When you use the CHAR(*n*) or VARCHAR(*n*) data type to store multibyte data in a DB2, Greenplum, or Oracle database, you must specify the encoding in the CLIENT_ENCODING= connection option. Or, for Oracle only, to avoid having to set the encoding, use the NCHAR or NVARCHAR data types for multibyte data instead.

4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Oracle

The following table lists the data type support for an Oracle database.

For data source-specific information about Oracle data types, see the Oracle database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Oracle support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.21 *Mapping of FedSQL Data Types to Data Types Used by Oracle*

Data Type Definition Keyword[1]	Oracle Data Type	Description	Data Type Returned
BIGINT	NUMBER	A large signed, exact whole number.	BIGINT
BINARY(*n*)	RAW(*n*)	A fixed or varying-length binary string.	BINARY(*n*)
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	TIMESTAMP(*p*)[3]
DECIMAL| NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)	A signed, fixed-point decimal number.	DOUBLE[4]
DOUBLE	BINARY_DOUBLE	A signed, double precision floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A signed, double precision floating-point number.	DOUBLE

Data Type Definition Keyword[1]	Oracle Data Type	Description	Data Type Returned
INTEGER	NUMBER(*p*,*s*)	A regular signed, exact whole number.	INTEGER
[2]	LONG	A varying-length character string data.	[2]
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character string.	NVARCHAR(*n*)
REAL	FLOAT	A signed, single precision floating-point number.	REAL
SMALLINT	NUMBER	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIMESTAMP(*p*)[3]
TIMESTAMP(*p*)[5]	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	NUMBER	A very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	RAW(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR2(*n*)[6]	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Oracle data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The timestamp returned by the DATE and TIME data types can be changed to date and time values by using the DATEPART function with the PUT function.

4 The ORNUMERIC= connection argument and table option determine how numbers read from or inserted into the Oracle NUMBER column are treated. ORNUMERIC=YES, which is the default, indicates that non-integer values with explicit precision are treated as NUMERIC values.

5 The TIMESTAMP(*p*) data type is not available on z/OS.

6 The VARCHAR2(*n*) type is supported for up to 32,767 bytes if the Oracle version is 12c and the Oracle MAX_STRING_SIZE= parameter is set to EXTENDED.

Data Types for PostgreSQL

The following table lists the data type support for a PostgreSQL database.

The BINARY, NCHAR, NVARCHAR, TINYINT, and VARBINARY data types are not supported for data type definition.

For data source-specific information about PostgreSQL data types, see the PostgreSQL database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about PostgreSQL support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.22 *Mapping of FedSQL Data Types to Data Types Used by PostgreSQL*

Data Type Definition Keyword[1]	PostgreSQL Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number or a signed eight-byte integer.	BIGINT
[2]	BIGSERIAL	An auto-incrementing eight-byte integer.	[2]
[2]	BIT(*n*)	A fixed-length bit string.	[2]
[2]	BIT VARYING(*n*)	A varying-length bit string.	[2]
[2]	BOOLEAN	A logical Boolean (true/false) value.	[2]
[2]	BOX	A rectangular box on a plane.	[2]
[2]	BYTEA	A byte array.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
[2]	CIDR	An IPv4 or IPv6 network address.	[2]
[2]	CIRCLE	A circle on a plane.	[2]

Data Type Definition Keyword[1]	PostgreSQL Data Type	Description	Data Type Returned
DATE	DATE	A date value.	DATE
DECIMAL|NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL|NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE PRECISION	A signed, double precision, floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A signed, double precision, floating-point number.	DOUBLE
[2]	INET	An IPv4 or IPv6 host address.	[2]
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	INTERVAL	A time span.	[2]
[2]	LINE	An infinite line on a plane.	[2]
[2]	LSEG	A line segment on a plane.	[2]
[2]	MACADDR	A Media Access Control address.	[2]
[2]	MONEY	A currency value.	[2]
[2]	PATH	A geometric path on a plane.	[2]
[2]	POINT	A geometric point on a plane.	[2]
[2]	POLYGON	A closed geometric path on a plane.	[2]
REAL	REAL	A signed, single precision floating-point number.	REAL
[2]	SERIAL	An auto-incrementing four-byte integer.	[2]
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
[2]	SMALL SERIAL	An auto-incrementing two-byte integer.	[2]

Data Type Definition Keyword[1]	PostgreSQL Data Type	Description	Data Type Returned
[2]	TEXT	A varying-length character string.	[2]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
[2]	TSQUERY	A text search query.	[2]
[2]	TSVECTOR	A text search document.	[2]
[2]	TXID_SNAPSHOT	A snapshot of a user-level transaction ID.	[2]
[2]	UUID	A universally unique identifier.	[2]
VARCHAR(*n*)	CHARACTER VARYING(*n*)	A varying-length character string.	VARCHAR(*n*)
[2]	XML	XML data.	[2]

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The PostgreSQL data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Reading from Salesforce

The following table lists the data type support for reading from Salesforce. SAS/ACCESS software uses the Salesforce SOAP API to read from Salesforce. The Salesforce SOAP API returns primitive types.

SAS supports the Winter '19 release of Salesforce and later.

The calculated type and compound types are not supported. Examples of compound types are address and location.

For more information, see documentation for the Salesforce SOAP API.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about

Salesforce support in CAS, see the SAS Viya documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.23 *Data Type Mapping When Reading from Salesforce*

Salesforce Field or Primitive Type	Description	Data Type Returned by FedSQL
anyType	A polymorphic data type that can be used differently throughout the interface based on context.	VARCHAR(*n*)
base64	A Base-64 binary string, including attachment records, document records, and scontrol records. The Body/Binary field contains data. The BodyLength field defines length of data. [1]	VARCHAR(*n*)
boolean	A Boolean value.	TINYINT
byte	A set of bits.	TINYINT
combobox	A set of enumerated values that allows the user to define a value that is not in the list.	VARCHAR(*n*)
currency	A currency value.	DOUBLE
DataCategoryGroupReference	A reference to a data category group or category unique name on Article and Question objects.	VARCHAR(*n*)
date	A date value.	DATE[2]
dateTime	A timestamp.	TIMESTAMP[2]
double	A signed, double precision, floating-point number.	DOUBLE[3]
email	An email address.	VARCHAR(*n*)
encryptedstring	An encrypted text string that contain a combination of letters, numbers, and symbols that are stored in encrypted form. The strings have a maximum length of 175 characters.	VARCHAR(*n*)
ID	The primary key value for an object.	VARCHAR(*n*)
int	A regular signed, exact whole number whose size varies depending on field settings.	BIGINT

Salesforce Field or Primitive Type	Description	Data Type Returned by FedSQL
JunctionIdList	A string array of referenced IDs values that represent the many-to-many relationship of an underlying junction entity.	VARCHAR(*n*)
masterrecord	Identifier of a record that is merged and the original records are deleted.	VARCHAR(*n*)
multipicklist	A set of enumerated values from which multiple values can be selected. The selected values are returned as a string of semicolon-separated values.	VARCHAR(*n*)
percent	A percentage value.	DOUBLE
phone	A phone number.	CHAR(40)
picklist	A set of enumerated values from which one value can be selected.	VARCHAR(*n*)
reference	A cross-reference for a different object. Analogous to a foreign key in SQL.	VARCHAR(*n*)
String	A text string whose maximum size varies depending on field settings.	CHAR(*n*)
textarea	A string that is displayed as a multiline text field.	VARCHAR(*n*)
Time	A time value.	TIME[2]
url	A Uniform Resource Locator (URL) value.	VARCHAR(*n*)

1 base64 fields remain in base64 format as a VARCHAR type.

2 The earliest valid date is 1700-01-01T00:00:00Z GMT, or just after midnight on January 1, 1700. The latest valid date is 4000-12-31T00:00:00Z GMT, or just after midnight on December 31, 4000. These values are offset by time zone. For example, in the Pacific time zone, the earliest valid date is 1699-12-31T16:00:00, or 4:00 PM on December 31, 1699.

3 double columns that match the precision and scale described for the numeric types in "Data Types for Writing to Salesforce" on page 1912 are mapped as shown in that table. For example, a double column that has a precision of 10 and scale of 0 is mapped to INTEGER.

Data Types for Writing to Salesforce

The following table lists the data type support for writing to Salesforce. SAS/ACCESS software uses the Salesforce Metadata API to write to Salesforce. The Salesforce Metadata API uses Custom Field types.

SAS supports the Winter '19 release of Salesforce and later.

The BINARY, FLOAT, and VARBINARY data types are not supported for data definition.

For more information, see documentation for the Salesforce Metadata API.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Salesforce support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.24 *Data Type Mapping When Writing to Salesforce*

Data Type Definition Keyword	Salesforce Custom Field Type	Description	Data Type Returned by CREATE TABLE with AS Expression
BIGINT	Number	A number with precision 18, scale 0.	BIGINT
CHAR(*n*)[1]	Text	A character string that is less than or equal to 255 characters.	CHAR(*n*)
DATE	Date	A date value.[2]	DATE
DECIMAL| NUMERIC(*p,s*)	Number	A number with precision and scale.	DOUBLE
DOUBLE	Number	A number with precision 18, scale 4.	DOUBLE
INTEGER	Number	A number with precision 10, scale 0.	INTEGER
NCHAR(*n*)	Text	A Unicode character string of fixed length.	CHAR(*n*)
NVARCHAR(*n*)	Text	A Unicode character string of varying length.	VARCHAR(*n*)

Data Type Definition Keyword	Salesforce Custom Field Type	Description	Data Type Returned by CREATE TABLE with AS Expression
REAL	Number	A number with precision 18, scale 4.	DOUBLE
SMALLINT	Number	A number with precision 5, scale 0.	SMALLINT
TIME	Time	A time value. [2]	TIME
TIMESTAMP	Datetime	A timestamp. [2]	TIMESTAMP
TINYINT	Number	A number with precision 3, scale 0.	TINYINT
VARCHAR(*n*)[3]	LongTextArea	A character string that is greater than 255 characters and less than 32K characters.	VARCHAR(*n*)

1 CHAR(*n*) values where *n* is greater than 255 characters are created as VARCHAR(*n*).

2 The earliest valid date is 1700-01-01T00:00:00Z GMT, or just after midnight on January 1, 1700. The latest valid date is 4000-12-31T00:00:00Z GMT, or just after midnight on December 31, 4000. These values are offset by time zone. For example, in the Pacific time zone, the earliest valid date is 1699-12-31T16:00:00, or 4:00 PM on December 31, 1699.

3 VARCHAR(*n*) values where *n* is less than or equal to 255 characters are created as CHAR(*n*).

Data Types for SAP

The following table lists the data type support for an SAP system.

For an SAP system, no data types are supported for column definition. Native ABAP SAP data types are mapped to similar data types for data retrieval only.

For data source-specific information about the ABAP SAP data types, see the SAP system database documentation.

Note: This data source is not supported on the CAS server.

Table A18.25 Mapping of SAP Data Types to FedSQL Data Types

ABAP SAP Data Type	Description	FedSQL Data Type Used For Data Retrieval
ACCP	Posting period.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CHAR	A fixed-length character string.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CLNT	A client value.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CUKY	A currency key. Fields of this type are referenced by fields of type CURR.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
CURR	A currency value. Corresponds to the DEC field. Field refers to a field of type CUKY.	CHAR(*n*)
DATS	A date value.	DATE
DEC	A signed, fixed-point decimal number.	CHAR(*n*)
FLTP	A floating-point number.	DOUBLE
INT1	A very small signed, exact whole number.	TINYINT
INT2	A small signed, exact whole number.	SMALLINT
INT4	A regular signed, exact whole number.	INTEGER
LANG	A language key, which has its own field format for special functions. The conversion exit ISOLA converts the value to be displayed to that of the database and the opposite is true.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
LCHR	A fixed-length character string.	VARCHAR(*n*) for non-Unicode SAP system; NVARCHAR(*n*) for Unicode SAP system

ABAP SAP Data Type	Description	FedSQL Data Type Used For Data Retrieval
LRAW	An uninterpreted varying-length byte string.	VARBINARY(*n*)
NUMC	Text string.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
PREC	The precision of a QUAN field.	CHAR(*n*)
QUAN	A quantity that corresponds to the DEC field.	CHAR(*n*)
RAW	An uninterpreted byte string.	BINARY
TIMS	A time value.	TIME(*p*)
UNIT	Units key and referenced by a QUAN data type.	CHAR(*n*) for non-Unicode SAP system; NCHAR(*n*) for Unicode SAP system
VARC	A varying-length character string data. As of SAP release 3.0, creating fields of this data type is no longer supported. Existing fields with this data type can be used, except in a WHERE condition in the SELECT statement.	VARCHAR(*n*) for non-Unicode SAS system; NVARCHAR(*n*) for Unicode SAP system

Data Types for SAP HANA

The following table lists the data type support for an SAP HANA database.

For data source-specific information about the SAP HANA data types, see the SAP HANA database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about SAP HANA support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.26 Mapping of FedSQL Data Types to Data Types Used by SAP HANA

Data Type Definition Keyword [1]	SAP HANA Data Type	Description	Data Type Returned
[2]	ALPHANUM(*n*)	A varying-length character string.	NVARCHAR(*n*)
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary data.	BINARY(*n*)
[2]	BLOB	A varying-length binary large object string.	[2]
CHAR(*n*)	CHAR(*n*)	A varying-length character string.	CHAR(*n*)
[2]	CLOB	A varying-length character large object string.	[2]
DATE	DATE	Year, month, and day values.	DATE
DECIMAL| NUMERIC(**p**,*s*)	DECIMAL(*p*,*s*)	A signed, exact, fixed-point decimal number.	DECIMAL(*p*,*s*)[3]
DOUBLE	DOUBLE	Double-precision floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	Double-precision floating-point number.	DOUBLE
INTEGER	INTEGER	32-bit integer.	INTEGER
NCHAR(*n*)	NCHAR(*n*)	A fixed-length Unicode character string.	NCHAR(*n*)
[2]	NCLOB	A fixed-length character large object string.	[2]
NVARCHAR(*n*)	NVARCHAR(*n*)	A varying-length Unicode character large object string.	NVARCHAR(*n*)
REAL	REAL	A floating-point number.	REAL
[2]	SECONDATE	A date and time value.	[2]

Data Type Definition Keyword [1]	SAP HANA Data Type	Description	Data Type Returned
[2]	SHORTTEXT(*n*)	A varying-length character string.	NVARCHAR(*n*)
[2]	SMALLDECIMAL(*p*,*s*)	A floating-point decimal number.	DECIMAL(*p*,*s*)[3]
SMALLINT	SMALLINT	16-bit integer.	SMALLINT
[2]	TEXT	A varying-length Unicode character large object string.	[2]
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	TINYINT	An unsigned eight-bit integer.	TINYINT
VARBINARY(*n*)	VARBINARY(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The SAP HANA data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for SAP IQ

The following table lists the data type support for an SAP IQ database.

For data source-specific information about the SAP IQ database data types, see the SAP IQ database documentation.

Note: This data source is not supported on the CAS server.

Table A18.27 Mapping of FedSQL Data Types to Data Types Used by SAP IQ

Data Type Definition Keyword[1]	SAP IQ Data Type	Description	Data Type Returned
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[3]	BIT	Integer that stores only the values 0 or 1.	[3]
CHAR(*n*)	CHAR(*n*)	A varying-length character string.	VARCHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p,s*)	DECIMAL(*p,s*)	A signed, exact, fixed-point decimal number.	DECIMAL(*p,s*)[4]
DOUBLE	DOUBLE	Double-precision floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	Double-precision floating-point number.	DOUBLE
INTEGER	INTEGER	32-bit integer.	INTEGER
[2]	LONG BINARY	A varying-length binary string.	VARBINARY(*n*)
REAL	REAL	A floating-point number.	REAL
SMALLINT	SMALLINT	16-bit integer.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	TINYINT	An unsigned eight-bit integer.	TINYINT
VARBINARY(*n*)	VARBINARY(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	CHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The SAP IQ data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The SAP IQ data type cannot be defined or retrieved.

4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Snowflake

The following table lists the data type support for Snowflake. SAS support for Snowflake is based on version 2.19.2 or later of the Snowflake ODBC driver.

The FedSQL NCHAR, NVARCHAR, and TINYINT data types are not supported for data definition.

For data source-specific information about the Snowflake data types, see the Snowflake documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Snowflake support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.28 *Mapping of FedSQL Data Types to Data Types Used by Snowflake*

Data Type Definition Keyword	Snowflake Data Type	Description	Data Type Returned
[1]	ARRAY	Dense or sparse arrays of arbitrary size, and values have VARIANT type.	VARCHAR
BIGINT	BIGINT	64-bit integer.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY
[1]	BOOLEAN	A true, false, or unknown (null) value.	BINARY
CHAR(*n*)	VARCHAR(1)	A fixed-length character string.	VARCHAR
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p,s*)	NUMBER(*p,s*)	A signed, exact, fixed-point decimal number.	DECIMAL(*p,s*)[2]
DOUBLE	FLOAT	Double-precision floating-point number.	DOUBLE

Data Type Definition Keyword	Snowflake Data Type	Description	Data Type Returned
FLOAT	FLOAT, FLOAT4, FLOAT8	Double-precision floating-point number.	DOUBLE
INTEGER	NUMBER without precision and scale	32-bit integer.	INTEGER
[1]	OBJECT	a collection of key-value pairs, where the key is a non-empty string, and the value is a value of the VARIANT type.	VARCHAR
REAL	FLOAT	A floating-point number.	FLOAT
SMALLINT	NUMBER without precision and scale	16-bit integer.	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP, TIMESTAMP_LTZ, TIMESTAMP_NTZ, TIMESTAMP_TZ	A date and time value.	TIMESTAMP(*p*)
VARBINARY(*n*)	VARBINARY	A varying-length binary string.	BINARY
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string. The default (and maximum) length is 16,777,216.	VARCHAR
[1]	VARIANT	A universal data type that can store values of any other Snowflake data type, including OBJECT and ARRAY, up to 16 MB.	VARCHAR

1 The Snowflake data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

2 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Spark

The following table lists the data type support for Apache Spark.

The NCHAR and NVARCHAR data types are not available for data definition.

For data source-specific information about Spark SQL data types, see the Spark SQL documentation.

Note: This data source is not supported on the CAS server.

Table A18.29 *Mapping of FedSQL Data Types to Data Types Used by Apache Spark*

Data Type Definition Keyword	Spark Data Type	Description	Data Type Returned
[1]	ARRAY<*data-type*>	An array of integers (indexable lists).	STRING[5, 7]
BIGINT	BIGINT	A signed eight-byte integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	BIGINT
BINARY(*n*)	BINARY	A varying-length binary string up to 32K.	BINARY
[1]	BOOLEAN	A textual true or false value.	TINYINT
CHAR(*n*)	CHAR(*n*)[9]	A character string up to 255 characters. [3]	CHAR
DATE	DATE[2]	An ANSI SQL date value.	DATE
DECIMAL/ NUMERIC(*p*,*s*)	DECIMAL	A fixed-point decimal number, with 38 digits precision.	DECIMAL(*p*,*s*)[4]
DOUBLE	DOUBLE	An eight-byte, double-precision floating-point number.	DOUBLE

Data Type Definition Keyword	Spark Data Type	Description	Data Type Returned
FLOAT	FLOAT	An eight-byte, double-precision floating-point number.	DOUBLE
INTEGER	INTEGER	A signed four-byte integer.	INTEGER
[1]	MAP<*primitive-type*, *data-type*>	An associative array of key-value pairs.	STRING[5, 7]
REAL	DOUBLE	A 64-bit double precision, floating-point number.	DOUBLE
SMALLINT	SMALLINT	A signed two-byte integer, from -32,768 to 32,767.	SMALLINT
[1]	STRING	A varying-length character string.	VARCHAR(*n*)[5, 6]
[1]	STRUCT<*col-name*: *data-type*>	A structure with established column elements and data types. Column elements and data types are mapped using a double-dot (:) notation.	STRING[5, 7]
TIME(*p*)	[8]	A time value.	STRING
TIMESTAMP(*p*)	TIMESTAMP	A UNIX timestamp with optional nanosecond precision.	TIMESTAMP[(*p*)]
TINYINT	TINYINT	A signed one-byte integer, from -128 to 127.	TINYINT
VARBINARY	BINARY	A varying-length binary string up to 32K.	BINARY

Data Type Definition Keyword	Spark Data Type	Description	Data Type Returned
VARCHAR(*n*)	VARCHAR(*n*)[9]	A varying-length character string.	VARCHAR(*n*)[5]

1 The Spark data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.
2 The supported date values are between January 1, 0001 and December 31, 9999. Date values later than 9999 are read back as null values.
3 If you specify CHAR with a value greater than 255 characters, the column is created as VARCHAR(*n*) instead.
4 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.
5 The maximum length of VARCHAR(*n*) and the Hive complex types is determined by the DBMAX_TEXT= data source connection option.
6 SASFMT TableProperties are applied when reading STRING columns.
7 The complex types ARRAY, MAP, STRUCT are read as their STRING representation of the underlying complex type. ARRAY values are read back within brackets, for example: [1, 2, 4]. STRUCT and MAP values are read back within braces, for example: {"firstname":"robert","nickname":"bob"}.
8 Spark does not support the TIME(*p*) data type. When data is read from Spark, STRING columns that have SASFMT TableProperties defined that specify the SAS TIME8. format are converted to the TIME(*p*) data type. When the TIME type is used for data definition, it is mapped to a STRING column with SASFMT TableProperties. Fractional seconds are not preserved. For more information about SASFMT TableProperties, see "SAS Table Properties for Spark and Hadoop" in *SAS/ACCESS for Relational Databases: Reference*.
9 In Apache Spark 2.0.0, the maximum number of columns allowed in a Spark table might be limited by https://issues.apache.org/jira/browse/SPARK-18016 .

Data Types for Teradata

The following table lists the data type support for a Teradata database.

The NCHAR, NVARCHAR, and REAL data types are not supported for data type definition.

For data source-specific information about the Teradata data types, see the Teradata database documentation.

Note: The information in this table does not apply to data that is processed in CAS. Data that is loaded into CAS is converted to CAS data types. For information about Teradata support in CAS, see the documentation for SAS data connectors in *SAS Cloud Analytic Services: User's Guide*.

Table A18.30 *Mapping of FedSQL Data Types to Data Types Used by Teradata*

Data Type Definition Keyword[1]	Teradata Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number.	BIGINT

Data Type Definition Keyword[1]	Teradata Data Type	Description	Data Type Returned
BINARY(*n*)	BYTE(*n*)	A fixed-length binary string	BINARY(*n*)
[2]	BLOB	A large Binary Object string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
[2]	CLOB	A large Character Object string.	[2]
DATE	DATE	A date value.	DATE
DECIMAL(*p*,*s*)	DECIMAL(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL(*p*,*s*)[3]
NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)	A signed, varying-length decimal number.	NUMERIC(*p*,*s*)[3]
DOUBLE	FLOAT	A signed, double precision, floating-point number.	DOUBLE
FLOAT	FLOAT(*p*)	A signed, double precision, floating-point number.	DOUBLE
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER
[2]	LONG VARCHAR	A varying-length character string.	[2]
SMALLINT	SMALLINT	A small signed, exact whole number	SMALLINT
TIME(*p*)	TIME(*p*)	A time value.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
TINYINT	BYTEINT	A very small signed, exact whole number.	TINYINT
VARBINARY(*n*)	VARBYTE(*n*)	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The

values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Teradata data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL and NUMERIC data types are supported in requests that can be fully passed down to the data source. DECIMAL and NUMERIC columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Vertica

The following table lists the data type support for a Vertica database.

The NCHAR and NVARCHAR data types are not supported for data definition.

For data source-specific information about Vertica data types, see the Vertica database documentation.

Note: This data source is not supported on the CAS server.

Table A18.31 *Mapping of FedSQL Data Types to Data Types Used by Vertica*

Data Type Definition Keyword [1]	Vertica SQL Identifier	Description	Data Type Returned
BIGINT	BIGINT	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
BINARY(*n*)	BINARY(*n*)	A fixed-length binary string.	BINARY(*n*)
[2]	BOOLEAN	A logical Boolean (true/ false) value.	[2]
[2]	BYTEA	A varying-length binary string.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
[2]	DATETIME	A date and time value with or without time zone.	TIMESTAMP(*p*)

Data Type Definition Keyword [1]	Vertica SQL Identifier	Description	Data Type Returned
DECIMAL| NUMERIC(*p*,*s*)	DECIMAL| NUMERIC(*p*,*s*)	A signed, fixed precision and scale numbers. Default precision and scale: 37, 15.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE PRECISION	A signed 64-bit IEEE floating point number, requiring 8 bytes of storage	DOUBLE
FLOAT	FLOAT, FLOAT8	A signed 64-bit IEEE floating point number, requiring 8 bytes of storage	DOUBLE
INTEGER	INTEGER	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
[2]	INTERVAL	Time span.	[2]
[2]	INTERVAL DAY TO SECOND	Time span.	[2]
[2]	INTERVAL YEAR TO MONTH	Time span.	[2]
[2]	LONG VARCHAR	Varying-length, raw-byte data, such as spatial data.	[2]
[2]	LONG BINARY	Long varying length binary string.	[2]
[2]	MONEY(*p*,*s*)	A currency value of signed, fixed precision and scale. Default precision and scale: 18, 4.	[2]
[2]	NUMBER(*p*,*s*)	A signed, fixed precision and scale numbers. Default precision and scale: 38,0.	[2]
[2]	RAW	A varying-length binary string.	[2]

Data Type Definition Keyword [1]	Vertica SQL Identifier	Description	Data Type Returned
REAL	REAL	A signed 64-bit IEEE floating point number, requiring 8 bytes of storage.	DOUBLE
[2]	SMALLDATETIME	A date and time value with or without time zone.	TIMESTAMP(*p*)
SMALLINT	SMALLINT	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
TIME(*p*)	TIME(*p*)	A time value without time zone.	TIME(*p*)
[2]	TIMETZ	A time value with time zone.	TIME(*p*)
TIMESTAMP(*p*)	TIMESTAMP	A date and time without time zone.	TIMESTAMP(*p*)
[2]	TIMESTAMPTZ	A date and time with time zone	TIMESTAMP(*p*)
TINYINT	TINYINT	A signed 64-bit integer, requiring 8 bytes of storage.	BIGINT
VARBINARY(*n*)	VARBINARY	A varying-length binary string.	VARBINARY(*n*)
VARCHAR(*n*)	VARCHAR(*n*), TEXT	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Vertica data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Data Types for Yellowbrick

The following table lists the data type support for a Yellowbrick database.

The BINARY, NCHAR, NVARCHAR, TINYINT, and VARBINARY data types are not supported for data type definition.

Note: This data source is not supported on the CAS server.

For data source-specific information about Yellowbrick data types, see the Yellowbrick database documentation.

Table A18.32 *Mapping of FedSQL Data Types to Data Types Used by Yellowbrick*

Data Type Definition Keyword[1]	Yellowbrick Data Type	Description	Data Type Returned
BIGINT	BIGINT	A large signed, exact whole number or a signed eight-byte integer.	BIGINT
[2]	BOOLEAN	A logical Boolean (true/false) value.	[2]
CHAR(*n*)	CHAR(*n*)	A fixed-length character string.	CHAR(*n*)
DATE	DATE	A date value.	DATE
DECIMAL| NUMERIC(*p*,*s*)	NUMERIC(*p*,*s*)	A signed, fixed-point decimal number.	DECIMAL| NUMERIC(*p*,*s*)[3]
DOUBLE	DOUBLE PRECISION	A signed, double precision, floating-point number.	DOUBLE
FLOAT	DOUBLE PRECISION, FLOAT, FLOAT8	A signed, double precision, floating-point number.	DOUBLE
INTEGER	INTEGER	A regular signed, exact whole number.	INTEGER

Data Type Definition Keyword[1]	Yellowbrick Data Type	Description	Data Type Returned
[2]	MACADDR, MACADDR8	A Media Access Control address.	[2]
REAL	REAL, FLOAT4	A signed, single precision floating-point number.	REAL
SMALLINT	SMALLINT	A small signed, exact whole number.	SMALLINT
TIME(*p*)	TIME	A time value.	TIME
TIMESTAMP(*p*)	TIMESTAMP(*p*)	A date and time value.	TIMESTAMP(*p*)
[2]	UUID	A universally unique identifier.	[2]
VARCHAR(*n*)	CHARACTER VARYING(*n*)	A varying-length character string.	VARCHAR(*n*)

1 The CT_PRESERVE= connection argument, which controls how data types are mapped, can affect whether a data type can be defined. The values FORCE (default) and FORCE_COL_SIZE do not affect whether a data type can be defined. The values STRICT and SAFE can result in an error if the requested data type is not native to the data source, or the specified precision or scale is not within the data source range.

2 The Yellowbrick data type cannot be defined, and when data is retrieved, the native data type is mapped to a similar data type.

3 The DECIMAL data type is supported in requests that can be fully passed down to the data source. DECIMAL columns in requests that cannot be fully passed down to the data source are handled as DOUBLE, which can affect precision.

Appendix 2

DS2 Expressions

What Is an Expression? 1931

Types of Expressions 1932
- Overview of Expressions 1932
- Primary Expression 1933
- Complex Expression 1934
- Array Expression 1935
- Function Expression 1936
- IN Expression 1936
- LIKE Expression 1937
- Method Expression 1939
- Package Method Expression 1941
- SYSTEM Expression 1942
- THIS Expression 1942
- IF Expression 1943
- SELECT Expression 1946

Operators in Expressions 1949
- Operator Precedence 1949
- Expression Values by Operator 1950
- Short-Circuit Evaluation in SAS 1953

What Is an Expression?

An expression is made of up of operands, and optional operators, that form a set of instructions and that resolves to a value.

An operand can be a single constant or variable, or it can be an expression. Operators are the symbols that represent either a calculation, comparison, or concatenation of operators.

Here are some examples of DS2 expressions:

```
a = b * c
```

```
"col1"
s || 1 || z
a >= b**(c - 8)
system.put(a*5,hex.)
```

Types of Expressions

Overview of Expressions

The basic type of expression is a primary expression. Complex expressions combine expressions and operators. Other expressions invoke a DS2 construct, such as a method expression or a function expression. The system expression and the THIS expression refer to expressions that are global in scope. In SAS mode, the IN expression returns a Boolean result based on whether the result of an expression is contained in a list.

Expression kinds commonly refer to a segment of code. For example, an AND expression refers to the AND operator and the operands that it processes. A binary expression refers to a binary constant or a hexadecimal constant such as `x'ff00effc'`.

Whether an expression is simple or complex, invokes a construct, or is global in scope, expressions of all kinds have a value and a data type.

Note: In logical operations, a missing value in any expression, such as an IF expression, evaluates to False in SAS mode. A null value evaluates to neither true nor false in ANSI mode. If you run this example in SAS mode, `False` is written to the SAS log. If you run this example is ANSI mode, `Null` is written to the SAS log.

```
proc ds2;
data _null_;
   declare double d;
   method init();
     if d then put 'True';
       else if not d then put 'False';
       else put 'Null';
   end;
   enddata;
run;
quit;
```

Primary Expression

In their simplest form, primary expressions are numbers, character strings, binary and hexadecimal constants, literal values, date and time values, identifiers, and null values, as in these primary expressions:

```
a
"var_a"
5.33
'Company'
date '2007-08-24'
```

The following table shows basic primary expressions, their data type(s), and an example:

Table A19.1 *Data Types and Examples of Primary Expressions*

Type of Expression	Short Description	Data Type	Example
Array	Groups same type data	Same type as individual items in the array	`a[5, b+c]`
Binary	Binary and hexadecimal constants	VARBINARY	`x'FE'` `b'01000011'`

Character	Character string	CHAR, VARCHAR	`'New Report'`
			`a='Stock';`
			`b='Report';`
			`c=a|| b;`
			`1`

National Character	National Character string	NCHAR, NVARCHAR	`n'New Report'`
			`a=n'Stock';`
			`b=n'Report';`
			`1`

Dot	System, THIS, and package method expressions	The resolved type of the expression	`system.put(x,5.)`
			`this.s`
			`p.calc(2,6,9)`

Date / time	Date and time values	DATE, TIME, TIMESTAMP, or DOUBLE	`date '2007-01-01'`

Type of Expression	Short Description	Data Type	Example

Identifier	Provides a name for various language elements or is a keyword	The declared or default type. The default is DOUBLE.	`a`
			`"part1"`
			`IN`

Integer	Integer numbers	INTEGER, BIGINT	`123`
New	Instantiates a package method	The resolved type of the expression	`a=_new_ package_name();`[1]
Numeric	Integer, real, and floating point numbers, or a missing or null value	DOUBLE, FLOAT, REAL	`5` `4.3`
Null	Null expression	none	`NULL`
Parentheses	Operator and operands enclosed in parentheses for higher evaluation precedence	The resolved type of the expression enclosed in parentheses.	`(a + b) - c`[1]

1 For the purpose of the example, the primary expression is contained in an expression or an assignment statement. The example expression is highlighted.

Complex Expression

A complex expression combines expressions and operators to create a more expansive expression, as in these expressions:

```
a + b * -c - 5
a = b = 5
x | y & a < c * d
x**y**z - 9 >= f
z >< c + e <> u**y * 10
x || c + 7
a in (1,2,3)
```

Evaluation of complex expressions is based on the operator order of precedence, as shown in Table A2.5 on page 1949, and the data types of the primary expressions. Before any calculations can be done, operand data types must be the same general data type: numeric, character, binary, or date/time. If the data types are the same, processing can proceed. If they are not the same, the operand data types are converted based on the operator and the data type of the operands.

In the expression `a + b / -c - 5`, assume that `a` is 1.35 with a type of DOUBLE, `b` is 2 with a data type of INTEGER, and `c` is 3 with a data type of INTEGER. `b/-c` or

`2/-3` evaluates to INTEGER 0. The INTEGER 0 is converted to a DOUBLE 0 before being added to `a`. Then `a + 0.0` evaluates to DOUBLE 1.35. The INTEGER 5 is converted to a DOUBLE 5 before the addition of the DOUBLE 1.35. The final result of the expression is DOUBLE −3.65.

For information about data type conversion, see "DS2 Type Conversions" in *SAS DS2 Programmer's Guide*.

In the expression `a = b = 5`, if `b` is a value other than 5, then `b = 5` is evaluated to 0. Therefore, `a` is assigned a value of 0. The first equal sign (=) is an assignment operator and the second equal sign is a logical equality operator. For more information, see "Example 2: Using an Expression with Multiple Equal Signs" on page 1213.

Note: DS2 supports using `eq` as well as the equal sign.

Array Expression

An array expression is a primary expression that represents a grouping of data items of the same data type. Although an array can have multiple dimensions, individual data item values are scalar values. Data items are accessed by specifying an index into the array.

The array expression consists of an array identifier followed by an array index expression for each dimension in the array, as in this syntax:

array-identifier \[*index-expression* [, ...*index-expression*] \]

Note: Brackets in the syntax convention indicate optional syntax. The escape character (\ ) before a bracket indicates that the bracket is required for the syntax. Indexes in an array expression must be contained by brackets ([ ]).

The array identifier can be either a declared array variable or a variable used in a THIS expression. The index expression is a primary expression that resolves to an integer.

Here are some examples of array expressions:

```
a[i]
s[j * 2, k-3]
this.c[2, vwind, a[i]]
```

When an array is declared, the index values specify the boundaries for the array. If an index expression is beyond the boundaries of the array, DS2 issues an error.

The value of an array expression is the value of the indexed value in the array. For example, if the array values are a[1] = 12, a[2] = 15, and a[3] = 20, the value of the array expression a[2] is 15.

Note: Arrays are 1-based. The array index starts at 1.

Function Expression

A function expression invokes a function within a DS2 program. To invoke a function, use this syntax:

function-name ([*argument* [, ...*argument*]])

Functions might require arguments. If the function expression contains arguments, the argument data types are converted, if necessary, to the data types of the function **signature**, which is the argument order and data type for a function. Parentheses in the function call are required, whether the function takes arguments or no arguments are required. For example, the TIME function does not require arguments:

```
t = time();
```

A function expression resolves to the value returned by the function. In the function expression above, `time()` resolves to the current time of day.

Methods and functions are similar. Functions have global scope. Methods are programming blocks and have local scope.

If the name of a function is identical to a method name, DS2 invokes the method. Functions with the same name as a method can be invoked only by using a SYSTEM expression.For more information, see "SYSTEM Expression" on page 1942.

For a list of DS2 functions, see "DS2 Functions" on page 229.

IN Expression

An IN expression determines whether an expression is contained in a constant list. See "Constant List" in *SAS DS2 Programmer's Guide*.

Here is the syntax of an IN expression:

expression [*not-operator*] **IN** *constant-list*

Note: Any valid data type for your data source can be used in *constant-list*. If any argument is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already). If any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already). Otherwise, all arguments are converted to a BIGINT. The result is always INTEGER, either 0 or 1.

Any of the NOT operators (~, ^, or NOT) are valid before the IN operator, which results in the logical negation of the expression value.

Consider these differences, according to mode:

- In SAS mode, an IN expression evaluates to TRUE (a nonzero value) or FALSE (0).
- In ANSI mode, an IN expression evaluates to TRUE (a nonzero value), FALSE (0), or NULL.

This is the same behavior as DS2's logical (AND, OR, NOT) and relational (<=, <, and so on) operations.

If the IN expression and the constants in the constant list do not contain a NULL or SAS missing value, the result of the IN operation is TRUE or FALSE regardless of the mode.

If the IN expression or a constant in the constant list contains a SAS missing value or NULL, consider these differences, according to mode:

- In SAS mode, if the IN expression or a constant in the constant list contains a SAS missing value, the result of the IN operation is still TRUE or FALSE.
- In ANSI mode, if the IN expression or a constant in the constant list contains a NULL value, the result of the IN operation is NULL.

For more information about how DS2 processes nulls and SAS missing values, see "DS2 Modes for Nonexistent Data" in *SAS DS2 Programmer's Guide*.

The following table shows the results of some IN expressions:

Table A19.2 *Examples of IN Expressions*

Input Values	IN Expression	Results
`a = 2`	`a in (5,34,2,67)`	`1`
`b = 3`	`b not in (3,22,43,65)`	`0`

LIKE Expression

Overview of the LIKE Expression

A LIKE expression determines whether a character string matches a pattern-matching specification.

Here is the syntax of a LIKE expression:

expression [NOT] **LIKE** *pattern-matching-expression* [ESCAPE *character-expression*]

The expressions can be any character string or binary string data type.

If *expression* matches the pattern specified by *pattern-matching-expression*, a value of 1 (true) is returned. Otherwise, a value of 0 (false) is returned.

NOT LIKE returns the inverse value of LIKE. For example, if `x like y` is true, then `x not like y` is false.

The ESCAPE argument is used to search for literal instances of the percent (%) and underscore (_) characters, which are usually used for pattern matching.

Patterns for Searching

Patterns consist of three classes of characters.

Table A19.3 *Pattern-matching Characters*

Character	Description
underscore (_)	matches any single character
percent sign (%)	matches any sequence of zero or more characters **Note:** Be aware of the effect of trailing blanks. To match values, you might have to use the TRIM function to remove trailing blanks.
any other character	matches that character

Searching for Literal % and _

Because the % and _ characters have special meaning in the context of the LIKE expression, you must use the ESCAPE argument to search for these character literals in the input character string.

These examples use the values **app**, **a_%**, **a__**, **bbaa1**, and **ba_1**.

- The condition `like 'a_%'` matches **app**, **a_%**, and **a__**, because the underscore (_) in the search pattern matches any single character (including the underscore), and the percent (%) in the search pattern matches zero or more characters, including '%' and '_'.
- The condition `like 'a_^%' escape '^'` matches only **a_%**, because the escape character (^) specifies that the pattern search for a literal '%'.
- The condition `like 'a_%' escape '_'` matches none of the values, because the escape character (_) specifies that the pattern search for an 'a' followed by a literal '%', which does not apply to any of these values.

Searching for Mixed-case Strings

The DS2 LIKE expression is case sensitive. To search for mixed-case strings, use the UPCASE function as the following example shows:

```
upcase(str) like 'SM%'
```

LIKE Expression Examples

The following table shows examples of the matches that would result when searching these strings: `Smith, Smooth, Smothers, Smart, Smuggle`.

Table A19.4 *Examples of LIKE Expressions*

LIKE Expression Example	Matching Results
`str like 'Sm%'`	`Smith, Smooth, Smothers, Smart, Smuggle`
`str like '%th'`	`Smith, Smooth`
`str LIKE 'S__gg%'`	`Smuggle`
`str like 'S_o'`	(no matches)
`str like 'S_o%'`	`Smooth, Smothers`
`str like 'S%th'`	`Smith, Smooth`
`str not like 'Z'`	`Smith, Smooth, Smothers, Smart, Smuggle`

Method Expression

A method expression invokes a method that has been defined by the METHOD statement.

To invoke a method, use this syntax:

method-name ([*expression* [, ... *expression*]])

Methods are invoked based on the method name and signature. DS2 first identifies the method name. If a method name is identical to a function name, DS2 invokes the method. A function with the same name as a method can be invoked by using a SYSTEM expression. For more information , see “SYSTEM Expression” on page 1942.

Because DS2 allows overloaded methods, DS2 invokes the method whose arguments best match the number of arguments and the argument data types in the method signature, which is the argument order and data type for the method. The best match is the one for which the number of method parameters is equal to the number of arguments, and such that no other method signature has as many exact parameter type matches for the given argument list. If a best match is not found, an error occurs.

> **TIP** DS2 methods can support up to 1000 arguments. A DS2 method that has more than 1000 arguments can generate a compilation error.

Once the method to execute is identified, DS2 converts argument data types to the data type of the corresponding method parameter, if necessary. The method then executes.

A method expression resolves to the value returned by the method.

In the following example, methods CONCAT and ADD are defined and then invoked in the INIT() method. The highlighted expressions in the INIT() method are method expressions:

```
method concat(char(100) x, char(100) y) returns char(200);
   return trim(x)|| y;
end;

method add(double x, double y) returns double;
   return x + y;
end;

method init();
   dcl char(200) r;
   r = concat('abc', 'def');
   d = add(100,101);
end;
```

In this next example, the D method is an overloaded method. DS2 must compare the method expression arguments to the method signatures to find the best match:

```
method d(double x, double y) returns double;
   return x + y;
end;

method d(int x, int y) returns int;
   return x + y;
end;

method init();
   dcl double r;
   dcl int i;
   r = d(1.2345, 5.6789);
   i = d(99, 100);
end;
```

The first method calls the D method whose signature requires values with DOUBLE data types. The second method calls the D method whose signature requires values with INTEGER data types.

This final example shows that DS2 cannot determine whether the values in the method expression have a data type of INTEGER or DOUBLE. Because it is ambiguous, DS2 issues an error:

```
method d(int x, double y) returns double;
   return x + y;
end;

method d(double x, int y) returns double;
   return x + y;
end;

method run();
   d = d(100, 102);
end;
```

For more information, see the "METHOD Statement" on page 1270.

Package Method Expression

A package method expression instantiates a method that is defined in a package. To invoke a package method expression, use this syntax:

package-name.method-name ([*method-argument* [, ... *method-argument*]])

Package method expressions execute in a manner similar to method expressions. That is, once DS2 has determined that the package and the method exist, the best match of method signatures is determined, argument data types are converted if necessary, and the method executes.

> **TIP** DS2 methods can support up to 1000 arguments. A DS2 method that has more than 1000 arguments can generate a compilation error.

In the following example, the highlighted expressions are package method expressions:

```
declare package myadd a1() a2();
a1.sale(3,4);
a1.add(1,2);
a2.bonus(5,12);
a2.add(10,20);
```

The first two package method expressions invoke the SALE and ADD methods in the A1 package, which was instantiated from the MYADD package. The last two package method expressions invoke the BONUS and ADD methods in the A2 package.

Note: You can invoke a DS2 package method expression as a function in a FedSQL SELECT statement. For more information, see "Using DS2 Packages in Expressions" in *SAS FedSQL Language Reference*.

For information about packages, see the "PACKAGE Statement" on page 1286 and "DS2 Packages" in *SAS DS2 Programmer's Guide*.

SYSTEM Expression

When a method and a function have identical names, the method call takes precedence over the function call. The function can then be invoked only by using a SYSTEM expression.

To invoke a SYSTEM expression, use this syntax:

SYSTEM.*function-expression*

A SYSTEM expression prepends a function expression with the dot notation, `system.`. For example, if SUM is the name of a method as well as the name of a function, the SUM function only can be invoked by using the SYSTEM expression: `system.sum(a,b,c)`.

THIS Expression

A THIS expression provides an alternate method to simultaneously declare and use a global scalar variable from anywhere within a DS2 program. A THIS expression is used to circumvent the standard variable lookup. In a THIS expression, DS2 searches for a scalar variable declaration of the identifier in global scope. If there is no such declaration, DS2 declares the identifier in global scope with DOUBLE type. Global variables can be referenced by all programming blocks in a DS2 program.

To invoke a THIS expression, use this syntax:

THIS.*variable-name*

A THIS expression prepends a variable with the dot notation, `THIS.`.

Note: DS2 stores `THIS.`*`variable-name`* only as *variable-name*. If you have a local variable with the same name as the global scalar variable and DS2 issues a diagnostic message about the variable, you will not be able to distinguish which variable is a problem. For example, if DS2 issues a warning message that **x** is not declared, you would not know whether the message refers to the global variable, **THIS.x**, or the local variable, **x**.

In the following example, the variable `s` becomes a global variable by using a THIS expression:

```
method init();
  this.s = sum(a,b,c,d,e);
end;
method run();
   t = put(this.s 5.4);
end;
```

The THIS expression provides a method to access a global variable that is hidden by a local variable with the same name. Here is an example.

```
 data;
   declare double x;    /* declare global x */

   method run();
      declare double x;  /* declare local x */

      /* Two variables exist with same name, "x". */
      /* Identifier "x" refers to local x in      */
      /* scope of run method.  Global x is hidden */
      /* by local x.                              */

      this.x = 1.0;    /* assign 1.0 to global x   */
      x = 0.0;         /* assign 0.0 to local x    */

      output;          /* global x with 1.0 output */
    end;
enddata;
run;
```

IF Expression

Overview of the IF Expression

The conditional IF expression is used to select between two values based on whether a conditional expression evaluates to true (a nonzero value) or false (zero).

To invoke an IF expression, use this syntax:

IF *expression-1* **THEN** *expression-2* **ELSE** *expression-3*

If *expression-1*is a nonzero value, the result of the IF expression is the value of *expression-2*. Otherwise, the result of the IF expression is the value of *expression-3*. Here is an example.

```
 m=(if missing(u) then 0 else u);
```

The IF expression can be used wherever any other expression can be used. The precedence of an IF expression is lower than the arithmetic and logic operators, Therefore, parentheses are necessary in mixed expressions like this one.

```
r = 25.5 + (if sum < 15 then -a else b*2);
```

Without the parentheses, the plus (+) operator would be evaluated first resulting in a parse error from the subexpression `25.5 + if`.

Nested IF Expressions

IF expressions can be nested to select between many values for a multi-way decision.

IF *condition-expression-1***THEN** *result-expression-1*

ELSE IF *condition-expression-2* **THEN** *result-expression-2*

...

ELSE IF *condition-expression-n* **THEN** *result-expression-n*

ELSE *result-expression-default*

The condition expressions are evaluated in order. The result of the nested IF expression chain is the associated *result-expression* of the first *condition-expression* that evaluates to true (a nonzero value). If all the *condition-expressions* evaluate to false (zero), the result of the IF expression is the *result-expression-default*. Here is an example.

```
grade = if score >= 90 then 'A'
   else if score >= 80 then 'B'
   else if score >= 70 then 'C'
   else if score >= 60 then 'D'
   else if score >= 0  then 'F'
   else NULL;
```

Note: Use a SELECT statement rather than a series of IF-THEN/ELSE statements when you have a long series of mutually exclusive conditions. A large number of nested IF-THEN/ELSE statements could cause an internal error. By contrast, the SELECT statement is evaluated only once, which could improve performance without causing errors.

For example, assume that a program contains many ELSE IF constructs like the following program.

```
if (e1) then …
else if (e2) then …
…
else if (en) then …
else …
```

Use this SELECT statement instead.

```
select;
when (e1): …
when (e2): …
…
when (en): …
otherwise: …
```

IF Expression Data Type

The data type of an IF expression is determined by examining the type of the first result expression, *expression-2*.

```
IF expression-1 THEN expression-2 ELSE expression-3
```

If *expression-2* is not a numeric data type, then the IF expression is assigned the type of *expression-2*.

If *expression-2* is a numeric data type, then the IF expression is assigned the wider numeric data type of *expression-2* and *expression-3*. For example, if *expression-2* is an SMALLINT and *expression-3* is a DOUBLE, then the IF expression is assigned type DOUBLE. If *expression-2* is a numeric data type and *expression-3* is not a numeric data type, then the IF expression is assigned the type of *expression-2*.

If the first result expression in a nested IF expression chain is a numeric data type, then all the result expressions are examined to find the widest numeric data type to assign as the type of the nested IF expression chain. In the following example, `t` is a TINYINT, `b` is a BIGNINT, and `d` is a DECIMAL(10,5). The 0 in the ELSE is assigned type BIGINT. Therefore, the nested IF expression chain is assigned the type decimal(10,5), the widest numeric type of TINYINT, BIGINT, and DECIMAL(10,5).

```
r = if n < 0 then t
   else if n = 0 then b
   else if n > 0 then d
   else 0;
```

Note: If 0.0 had been used for the ELSE value instead of 0, then the ELSE result would have been assigned type DOUBLE instead of type BIGINT. With the type DOUBLE ELSE expression, the widest numeric type of the result expressions would be type DOUBLE. Therefore, the nested IF expression chain would be assigned type DOUBLE instead of type decimal(10,5).

Lazy Evaluation of IF Expressions

The IF expression uses lazy evaluation for the result expressions.

```
IF expression-1 THEN expression-2 ELSE expression-3
```

For example, you could use this code to check for division by zero.

```
a = if c ne 0 then b/c else null;
```

Expression *expression-1* is always evaluated, but only one of *expression-2* or *expression-3* is evaluated. The expression that is not selected as the result of the IF expression is not evaluated. Thus, if *expression-1* is a nonzero value (true), then only *expression-2* is evaluated. If *expression-1* is zero (false), then only *expression-3* is evaluated.

Lazy evaluation also applies to the result expressions of nested IF expression chains.

```
IF condition-expression-1 THEN result-expression-1
   ELSE IF condition-expression-2 THEN result-expression-2
   ...
   ELSE IF condition-expression-n THEN result-expression-2
   ELSE result-expression-default
```

The selected result expression is the only result expression evaluated. Lazy evaluation also applies to the condition expressions. Condition expressions are evaluated in order until a condition evaluates to true (nonzero) or all conditions are evaluated. If the IF expression has *n* condition expressions and the *i*th condition is the first nonzero condition, then only the first 1 to *i* conditions are evaluated. The *i* +1 to *n* conditions are not evaluated.

SELECT Expression

Overview of the SELECT Expression

A SELECT expression is used to select between multiple expressions based on the values of other expressions.

To invoke a SELECT expression, use this syntax:

SELECT [(*select-expression*)]

WHEN (*when-expression*) [...WHEN (*when-expression*)] *result-expression*

[...WHEN (*when-expression*) [...WHEN (*when-expression*)] *result-expression*]

OTHERWISE [*default-result-expression*]

END

The SELECT expression evaluates each WHEN expression in order until a matching expression is found. Then the associated *result-expression* is evaluated as the result of the SELECT expression.

The SELECT expression can be used wherever any other expression can be used. Here is an example.

```
r = 25.5 + select (t) when (1) -a when (3) b*2 end;
```

SELECT Expression with a Selection Expression

If a selection expression is present, then it is evaluated. Then the WHEN expressions are evaluated in order. The result of the SELECT expression is the result expression of the first WHEN expression that evaluates to the same value as the selection expression. If all the WHEN expressions evaluate to different values than the selection expression, the result of the SELECT expression is the default

result expression if present. Otherwise, it is a missing or null value. Here is an example.

```
s = select (t)
   when (1) x*10
   when (3) x
   when (5) x*100
   when (0) 0
   otherwise .
end;
```

If **t** is 5, then the SELECT expression evaluates to 'x*100'.

SELECT Expression without a Selection Expression

If a selection expression is not present, then the WHEN expressions are evaluated in order. The result of the SELECT expression is the result expression of the first WHEN expression that evaluates to true (a nonzero value). If all the WHEN expressions evaluate to false (zero), the result of the select expression is the default result expression if present. Otherwise, it is a missing or null value. Here is an example.

```
grade = select
   when (score >= 90) 'A'
   when (score >= 80) 'B'
   when (score >= 70) 'C'
   when (score >= 60) 'D'
   when (score >= 0 ) 'F'
end;
```

If **score** is 76, then the first *when-expression* to evaluate to true is **score>=70**. The *select-expression* evaluates to 'C'.

Optional Otherwise Expression

If an otherwise default result expression is not supplied, then DS2 provides a default result value to select when none of the WHEN expressions are selected. If the SELECT expression has type DOUBLE or CHAR in SAS mode, the default result value is a missing value (.). For all other data types in either mode, the default result value is NULL.

Result Expression with Multiple When Expressions

Multiple WHEN expressions can be associated with a single result expression. The WHEN expressions are listed consecutively followed by the single result expression. If any of the WHEN expressions associated with a result expression is the first matching WHEN expression, then the result of the SELECT expression is the result expression.

For example, the following SELECT expression evaluates to 'airplane' if the value of variable **t** is either 'A', 'a', 'P', or 'p'.

```
s = select (t)
   when ('A')
   when ('a')
   when ('P')
   when ('p') 'airplane'
   when ('C')
   when ('c') 'car'
   when ('T')
   when ('t') 'train'
   otherwise  'walk'
 end;
```

SELECT Expression Data Type

The type of a SELECT expression is determined by examining the type of the first result expression. If the first result expression is not a numeric data type, then the SELECT expression is assigned the type of the first result expression.

If the first result expression is a numeric data type, then all the result expressions are examined to find the widest numeric data type to assign as the type of the SELECT expression.

In the following example, **t** is a TINYINT, **b** is a BIGINT, **d** is a DECIMAL(10,5), and **s** is a CHAR(10). The select expression is assigned the type DECIMAL(10,5), the widest numeric type of TINYINT, BIGINT, DECIMAL(10,5), and CHAR(10).

```
r = select (t)
   when (1) t
   when (2) b
   when (3) d
   otherwise s
end;
```

Note: If **s** had been assigned to the first result expression, then the type of the SELECT expression would have been CHAR(10). If the first result expression has a non-numeric type, then the non-numeric type is assigned as the type of the SELECT expression.

Lazy Evaluation of the SELECT Expression

The SELECT expression uses lazy evaluation for the WHEN expressions. The WHEN expressions are evaluated in order until a matching WHEN expression is found or all when expressions are evaluated. If the SELECT expression has *n* WHEN expressions and the *i*th WHEN expression is selected, then only the first 1 to *i* WHEN expressions are evaluated. The *i*+1 to n when expressions are not evaluated.

Lazy evaluation also applies to the result expressions of the SELECT expression. The selected result expression is the only result expression evaluated.

In the following example, if `n[i]` equals 0, then only the first two WHEN expressions (`n[i] < 0` and `n[i] = 0`) are evaluated and only the second result expression `(y*10-r2)` is evaluated.

```
m(select
    when (n[i] < 0) y*100-r1
    when (n[i] = 0) y*10-r2
    when (n[i] > 0) y*10
 end);
```

Operators in Expressions

Operator Precedence

An operator symbolizes a type of operation that is to be performed on an operand, such as addition, comparison, and logical negation. When an expression contains multiple operators and operands, DS2 resolves the expression by using operator precedence. Operations are performed from the highest order of precedence to the lowest order of precedence.

The highest order of precedence is 1 and the lowest order of precedence is 9. Within a precedence level, with the exception of exponentiation, minimum, and maximum operators, operators associate from left to right. The exponentiation, minimum, and maximum operators associate from right to left.

By using the precedence order in Table A2.5 on page 1949, in the expression `5+a**b*3`, `a**b` is calculated first and then multipled by `3`, and that result is added to `5`.

> **TIP** In DS2, x < y < z is evaluated like x < y and y < z.

The following table lists the operators and their order of precedence:

Table A19.5 *Operators and Their Order of Precedence in DS2 Expressions*

Order of Precedence	Symbol	Associativity
1	( )	left to right
1	SELECT expression	left to right

Order of Precedence	Symbol	Associativity
2	+, –	right to left
2	^ or ~	left to right
2	**, <>, ><	left to right
3	*, /	left to right
4	+, –	left to right
5	|| or !!	left to right
5	..	left to right
6	IN, LIKE	left to right
7	=, ^= or ~=	right to left
7	>=, <=, >, < [1]	left to right
8	&	left to right
9	| or !	left to right
10	IF expression	right to left
none	:=	none
none	_NEW_	none
none	Method expression	none

1 In DS2, x < y < z is evaluated like x < y and y < z.

Expression Values by Operator

The following table shows the resolved value for expressions that are based on an operator:

Table A19.6 *Expression Values by Operator*

Operator	Value
Unary expressions	
Unary plus	Is the same as the expression operand
Unary minus	Is the arithmetic negation of the operand
NOT or ^ or ~	If the operand is nonzero, result is 0. If the operand is zero or missing, result is 1. If the operand is null, result is null.
Logical expressions	
OR or | or !	■ the logical OR of the two operands ■ null when one operand is zero or missing and the other operand is null ■ null when both operands are null ■ 1 when either operand is nonzero (even if the other operand is null or missing) ■ 0 when both operands are zero or missing
AND or &	■ the logical AND of the two operands ■ null when one operand is nonzero and the other operand is null ■ null when both operands are null ■ 1 when both operands are nonzero ■ 0 when either operand is zero or missing (even if the other operand is null)
Arithmetic expressions	
+	the arithmetic sum of the operands
–	the arithmetic difference of the operands
*	the arithmetic product of the operands
/	the arithmetic quotient of the operands
**	the left operand raised to the power of the right operand
><	the minimum of the left and right operands
<>	the maximum of the left and right operands

Operator	Value
any arithmetic operator	null when either or both operands are null
Relational expressions	
<	1 when the left operand is less than the right operand; otherwise, 0
>	1 when the left operand is greater than the right operand; otherwise, 0
<=	1 when the left operand is less than or equal to the right operand; otherwise, 0
>=	1 when the left operand is greater than or equal to the right operand; otherwise, 0
=	1 when the left operand is equal to the right operand; otherwise, 0
^=	1 when the left operand is not equal to the right operand; otherwise, 0
any logical operator	null when either or both operators are null
Concatenation expression	
|| or !!	the left operand merged with the right operand to form a new value
..	strips each argument before concatenating
IF expression	
IF	1 when the comparison expression is contained in the constant list
IN expression	
IN	1 when the comparison expression is contained in the constant list
LIKE expression	
LIKE	1 when the comparison expression is contained in the constant list
SELECT expression	
SELECT	1 when the comparison expression is contained in the constant list

1 The || concatenation operator does not remove any spaces. You can use the TRIM function to remove trailing spaces. However, the .. operator performs the same function as TRIM followed by

concatenation. It is faster to use the .. operator (a .. b) then to use the TRIM function (TRIM(a) || TRIM(b)).

Short-Circuit Evaluation in SAS

Minimal evaluation, or short-circuit evaluation, is a technique that some programming languages use to evaluate Boolean operators. In short-circuit evaluation, the second argument in a Boolean expression is executed only if the first argument does not determine the value of the overall expression. For example, in the expression (X AND Y), if the first argument, (X) evaluates to FALSE, then the overall expression (X AND Y) must evaluate to FALSE. So there is no need to calculate the second argument (Y).

SAS does not guarantee short-circuit evaluation. When using Boolean operators to join expressions, you might get undesired results if your intention is to short circuit, or avoid the evaluation of, the second expression. To guarantee the order in which SAS evaluates an expression, you can rewrite the expression using nested IF statements.

In the first example below, the logical expression does not short-circuit at the first expression, and the program fails because a division by zero is detected in the second expression.

```
data _null_;
 declare double a;
 method init();
   a=0;
   if (a>1 AND 1/a) then put 'hello';
   else put 'goodbye';
 end;
enddata;
run;
quit;
```

The following lines are written to the SAS log:

```
ERROR: Float divide by zero
ERROR: General error
```

In the next example, the same logical expression is written using nested IF statements to guarantee the order of evaluation and to ensure that the program runs successfully.

```
data _null_;
 declare double a;
 method init();
   a=0;
   if a>1 then
    if 1/a then put 'hello';
    else put 'goodbye';
   else put 'goodbye again';
 end;
enddata;
```

```
run;
quit;
```

The following lines are written to the SAS log:

```
goodbye again
```

Appendix 3

DS2 Example Programs

Example: Find Minimums **1957**
Example Overview: Find Minimums 1957
Example Code: Find Minimums 1957
Example Output: Find Minimums 1958

Example: SQL in a DS2 Program **1959**
Example Overview: SQL in a DS2 Program 1959
Example Code: SQL in a DS2 Program 1959
Example Output: SQL in a DS2 Program 1960

Example: Make Two New Tables Based on a Condition **1961**
Example Overview: Make Two New Tables Based on a Condition 1961
Example Code: Make Two New Tables Based on a Condition 1961
Example Output: Make Two New Tables Based on a Condition 1963

Example: Change Case of Text Output **1963**
Example Overview: Change Case of Text Output 1963
Example Code: Change Case of Text Output 1963
Example Output: Change Case of Text Output 1964

Example: Scope **1964**
Example Overview: Scope 1964
Example Code: Scope 1965
Example Output: Scope 1966

Example: Functions **1966**
Example Overview: Functions 1966
Example Code: Functions 1966
Example Output: Functions 1967

Example: Arrays **1967**
Example Overview: Arrays 1967
Example Code: Arrays 1968
Example Output: Arrays 1970

Example: SELECT Statement **1972**
Example Overview: SELECT Statement 1972
Example Code: SELECT Statement 1972
Example Output: SELECT Statement 1973

Example: GOTO and LEAVE Statements with Labels **1973**

Example Overview: GOTO and LEAVE Statements with Labels ... 1973
Example Code: GOTO and LEAVE Statements with Labels ... 1974
Example Output: GOTO and LEAVE Statements with Labels ... 1975

Example: Overloaded Methods ... **1976**
Example Overview: Overloaded Methods ... 1976
Example Code: Overloaded Methods ... 1976
Example Output: Overloaded Methods ... 1977

Example: Analyze a Table Using Multiple Threads ... **1978**
Example Overview: Analyze a Table Using Multiple Threads ... 1978
Example Code: Analyze a Table Using Multiple Threads ... 1978
Example Output: Analyze a Table Using Multiple Threads ... 1979

Example: Using Four Threads to Compute Summary Statistics ... **1980**
Example Overview: Using Four Threads to Compute Summary Statistics ... 1980
Example Code: Using Four Threads to Compute Summary Statistics ... 1980
Example Output: Using Four Threads to Compute Summary Statistics ... 1982

Example: Data Cleaning ... **1982**
Example Overview: Data Cleaning ... 1982
Example Code: Data Cleaning ... 1983
Example Output: Data Cleaning ... 1984

Example: SUBSTR Function ... **1984**
Example Overview: SUBSTR Function ... 1984
Example Code: SUBSTR Function ... 1984
Example Output: SUBSTR Function ... 1985

Example: PUT with SAS Formats ... **1985**
Example Overview: PUT with SAS Formats ... 1985
Example Code: PUT with SAS Formats ... 1985
Example Output: PUT with SAS Formats ... 1986

Example: Generate Statistics from Table Data ... **1986**
Example Overview: Generate Statistics from Table Data ... 1986
Example Code: Generate Statistics from Table Data ... 1987
Example Output: Generate Statistics from Table Data ... 1988

Example: Matrices and Non-linear Equations ... **1988**
Example Overview: Matrices and Non-linear Equations ... 1988
Example Code: Matrices and Non-linear Equations ... 1989
Example Output: Matrices and Non-linear Equations ... 1991

Example: DS2 Matrices and Regression Analysis ... **1992**
Example Overview: DS2 Matrices and Regression Analysis ... 1992
Example Code: DS2 Matrices and Regression Analysis ... 1992
Example Output: DS2 Matrices and Regression Analysis ... 1995

Example: Use the SQLSTMT Package in Another Package ... **1995**
Example Overview: Use the SQLSTMT Package in Another Package ... 1995
Example Code: Use the SQLSTMT Package in Another Package ... 1995

Example: Update the Values of a Table By Using Two Databases ... **1997**
Example Overview: Update the Values of a Table By Using Two Databases ... 1997
Example Code: Update the Values of a Table By Using Two Databases ... 1997

Example: Store FedSQL Statements in a Hash Package ... **1998**
Example Overview: Store FedSQL Statements in a Hash Instance ... 1998
Example Code: Store FedSQL Statements in a Hash Package ... 1998

Example Output: Store FedSQL Statements in a Hash Package ... 2001

Example: Run a DS2 Program from z/OS Batch ... **2004**
Example Overview: Run a DS2 Program from z/OS Batch ... 2004
Example Code: Run a DS2 Program from z/OS Batch ... 2004

Example: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File ... **2005**
Example Overview: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File ... 2005
Example Code: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File ... 2005
Example Output: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File ... 2007

Example: Reading JSON Text from HTTP and Creating a SAS Data Set ... **2009**
Example Overview: Reading JSON Text from HTTP and Creating a SAS Data Set ... 2009
Example Code: Reading JSON Text from HTTP and Creating a SAS Data Set ... 2009
Example Output: Reading JSON Text from HTTP and Creating a SAS Data Set ... 2012

Example: Find Minimums

Example Overview: Find Minimums

This example demonstrates how to use the three system-defined methods, INIT, RUN, and TERM.

A DS2 program executes in the following sequence:

1. Any global variables are declared.
2. The INIT method is called. INIT is typically used for variable initialization.
3. The RUN method is called. The RUN method is where the implicit loop exists. RUN executes until all input tables are completely read.
4. The TERM method is called. Final processing is performed.

The following program demonstrates this flow of control by finding the minimum values in a table. In this program, the INIT method initializes the variables used to find the current minimum, the RUN method compares input values with the current minimum, and the TERM method writes the minimums to an output table.

Example Code: Find Minimums

```
proc ds2;
```

```
/* Create table to work with in this example */
data xy_data;
   dcl double x y;
   method init();
      do x = 1 to 5;
         y = 2*x;
         output;
      end;
   end;
enddata;
run;
/* Find the minimum value for x and y */
data xy_mins;
   dcl double min_x min_y;
   retain min_x min_y;
   keep min_x min_y;

   method init();
      min_x = 999999;
      min_y = 999999;
   end;

   method run();
      set xy_data;
      if x < min_x then min_x = x;
      if y < min_y then min_y = y;
   end;

   method term();
      output;
   end;
enddata;
run;
/* Send result table of minimums to the PROC for output */
data;
   method run();
      set xy_mins;
   end;
enddata;
run;
quit;
```

Example Output: Find Minimums

The SAS System

MIN_X	MIN_Y
1	2

Example: SQL in a DS2 Program

Example Overview: SQL in a DS2 Program

This example illustrates how to use an SQL statement in a DS2 program. To access the output from an SQL query, put the query in a SET statement. When the SET statement runs, it sequentially reads the rows that are returned by the query.

Example Code: SQL in a DS2 Program

In this example, the first DS2 program calculates annual balances for an account into which contributions of $2000 are made every year from 2004 to 2014. The account carries a 7% interest rate, compounded annually. The second program generates the table. The third program writes the results of an SQL query. The query selects all rows from INVESTMENT where the value of INVESTMENT_YEAR is greater than 2010.

```
proc ds2;
data investment;
   dcl integer investment_year;
   dcl double capital;
   method init();
      capital = 0;
      do investment_year = 2004 to 2014;
         capital = capital + (2000 + .07 * (capital+2000));
         output;
      end;
   end;
enddata;
run;
data;
   method run();
      set investment;
   end;
enddata;
run;
data;
   method run();
      set {select * from investment where investment_year > 2010};
   end;
enddata;
run;
quit;
```

Example Output: SQL in a DS2 Program

The SAS System

INVESTMENT_YEAR	CAPITAL
2004	2140
2005	4429.8
2006	6879.886
2007	9501.478
2008	12306.58
2009	15308.04
2010	18519.61
2011	21955.98
2012	25632.9
2013	29567.2
2014	33776.9

The SAS System

INVESTMENT_YEAR	CAPITAL
2011	21955.98
2012	25632.9
2013	29567.2
2014	33776.9

Example: Make Two New Tables Based on a Condition

Example Overview: Make Two New Tables Based on a Condition

This example illustrates how to create tables based on a condition. Programs 1 and 2 create two tables, DEPT1_ITEMS and DEPT2_ITEMS, that hold costs for items used by two departments. The third program creates two tables, HIGHCOSTS and LOWCOSTS, based on the costs of the items in the two items tables. Programs 4 and 5 output the contents of the costs tables.

Example Code: Make Two New Tables Based on a Condition

```
proc ds2;
/* Program 1 */
data dept1_items (overwrite=yes);
   dcl varchar(20) item;
   dcl double cost;
   method init();
      item = 'staples';   cost =  1.59; output;
      item = 'pens';      cost =  3.26; output;
      item = 'envelopes'; cost = 11.42; output;
   end;
enddata;
run;
/* Program 2 */
data dept2_items (overwrite=yes);
   dcl varchar(20) item;
   dcl double cost;
   method init();
      item = 'erasers'; cost =  5.43; output;
      item = 'paper';   cost = 26.92; output;
      item = 'toner';   cost = 62.29; output;
   end;
enddata;
run;
/* Program 3 */
```

```
data lowCosts (overwrite=yes) highCosts (overwrite=yes);
   method run();
      set dept1_items dept2_items;
      if cost <= 10.00 then
         output lowCosts;
      else
         output highCosts;
      end;
enddata;
run;
/* Program 4 */
data;
   method run();
      set lowCosts;
   end;
enddata;
run;
/* Program 5 */
data;
   method run();
       set highCosts;
   end;
enddata;
run;
quit;
```

Example Output: Make Two New Tables Based on a Condition

The SAS System

ITEM	COST
staples	1.59
pens	3.26
erasers	5.43

The SAS System

ITEM	COST
envelopes	11.42
paper	26.92
toner	62.29

Example: Change Case of Text Output

Example Overview: Change Case of Text Output

The code in this example reads the two tables created in the previous example, DEPT1_COSTS and DEPT2_COSTS, and writes rows with values in the COST column of less than or equal to $10 in lowercase, and values greater than $10 in uppercase.

Example Code: Change Case of Text Output

```
proc ds2;
data;
   method run();
      set dept1_items dept2_items;
         if cost <= 10.00 then
            item = lowcase(item);
         else
            item = upcase(item);
         end;
enddata;
run;
quit;
```

Example Output: Change Case of Text Output

The SAS System

ITEM	COST
staples	1.59
pens	3.26
ENVELOPES	11.42
erasers	5.43
PAPER	26.92
TONER	62.29

Example: Scope

Example Overview: Scope

This example shows the use of CHAR and VARCHAR types and their operators and functions. Also, in this example, the variables A, B, and C are locally scoped to the INIT method. That is, their value is not seen outside of the INIT method and is not written to the result table.

Example Code: Scope

```
proc ds2;
data;
  dcl char(24) abc abc2;
  method init();
    dcl char(8) a b c;

    a = repeat('a',5);
    b = repeat('b',6);
    c = repeat('c',7);

    abc = a || b || c;
    abc2 = trim(a) || trim(b) || c;
  end;
enddata;
run;
data;
  dcl char(24) abc abc2;
  method init();
    dcl varchar(8) a b c;

    a = repeat('a',5);
    b = repeat('b',6);
    c = repeat('c',7);

    abc = a || b || c;
    abc2 = trim(a) || trim(b) || c;
  end;
enddata;
run;
quit;
```

Example Output: Scope

The SAS System

abc	abc2
aaaaaa bbbbbbb cccccccc	aaaaaabbbbbbbccccccccc

The SAS System

abc	abc2
aaaaaabbbbbbbccccccccc	aaaaaabbbbbbbccccccccc

Example: Functions

Example Overview: Functions

This example shows how to use a function in DS2. The code computes the number of bits required to store an integer.

Example Code: Functions

```
proc ds2;
data;
   dcl integer i bits;
   method init();
      do i = 1 to 1000 by 100;
         bits = ceil(log(i) / log(2));
         output;
      end;
   end;
enddata;
run;
```

```
quit;
```

Example Output: Functions

The SAS System

i	bits
1	0
101	7
201	8
301	9
401	9
501	9
601	10
701	10
801	10
901	10

Example: Arrays

Example Overview: Arrays

The first program illustrates basic array procedures. In the final section, elements of dblNegSubArray are specified by using numeric expressions inside the array brackets. Expressions that resolve to a number that falls out of the bounds of the declared size of the array give an error message.

The second program gives several examples of array assignments. When an array is assigned, data types that do not match the type of the target array are converted to the target array type.

Note that if you add an equal sign after a variable or array element in a PUT statement, then the output is preceded by the variable or array element name and an equal sign.

Example Code: Arrays

```
proc ds2;
/* Basic Arrays */
data _null_;
   dcl char(10)  strArray[4] str;
   dcl double    dblArray[3];
   dcl int       intArray[10];
   dcl double    dblNegSubArray[-4:-1];
   dcl int x;
   method init();
      put 'BASIC ARRAYS';
      strArray[1] = 'abc';
      strArray[2] = 'def';
      strArray[3] = 'ghi';
      strArray[4] = 'jkl';
      put strArray[1]= ;
      put strArray[2]= ;
      put strArray[3]= ;
      put strArray[4]= ;
      put;

      str = strArray[1];
      put str=;
      put;

      dblArray[1] = 3;
      dblArray[2] = 99;
      dblArray[3] = dblArray[2];
      put dblArray[1]= ;
      put dblArray[2]= ;
      put dblArray[3]= ;
      put;

      do j = 1 to 10;
         intArray[j] = j;
         put intArray[j]=;
      end;
      put;

      dblArray[3] = intArray[5];
      put dblArray[3]=;
      put;

      dblNegSubArray[-4] = 102;
      dblNegSubArray[-3] = 101;
      dblNegSubArray[-2] = 100;
      dblNegSubArray[-1] = 99;

      x = 5;
      y = 7;
```

```
      a = dblNegSubArray[x-y];
      b = dblNegSubArray[x-y-1];
      c = dblNegSubArray[x-y-2];
      put a= b= c=;
      put;

      /* These will produce out-of-bounds messages */
      a = dblNegSubArray[x-y-3];
      e = dblNegSubArray[0];
   end;
enddata;
run;
/* Array Assignment */
data _null_;
   dcl int x;
   dcl char(10) s1[4] s2[4];
   dcl double d1[10] d2[10];
   dcl double arr2x3[2,3] arr3x2[3:5,-1:0];

   method init();

      put 'ARRAY ASSIGNMENT';

      /* Assign array of constants to array s1 */
      s1 := ('abc', 'def', 'ghi', 'jkl');

      /* Assign array s to array s2 */
      s2 := s1;
      put s2[1]= ;
      put s2[2]= ;
      put s2[3]= ;
      put s2[4]= ;
      put;

      /*
      * Assign array of constants to array d1.  Use
      * iterators for repeated values.  Mismatched
      * types will be converted automatically.
      */
      d1 := (3*3.14159, 2*'5', 2*(1,2), 99);

      /* Assign array d to array e */
      d2 := d1;
      put d2[1]= ;
      put d2[2]= ;
      put d2[3]= ;
      put d2[4]= ;
      put d2[5]= ;
      put d2[6]= ;
      put d2[7]= ;
      put d2[8]= ;
      put d2[9]= ;
      put d2[10]=;
      put;

      /* Assign array of constants to array arr3x2 */
```

```
      arr2x3 := (2*(1,2,3));

      /* Assign  arr2x3 to arr3x2 */
      arr3x2 := arr2x3;
      put arr2x3[1,1]= arr3x2[3,-1]= ;
      put arr2x3[1,2]= arr3x2[3,0]= ;
      put arr2x3[1,3]= arr3x2[4,-1]= ;
      put arr2x3[2,1]= arr3x2[4,0]= ;
      put arr2x3[2,2]= arr3x2[5,-1]= ;
      put arr2x3[2,3]= arr3x2[5,0]= ;
   end;
enddata;
run;
quit;
```

Example Output: Arrays

The following lines are written to the SAS log.

```
BASIC ARRAYS
strarray[1]=abc
strarray[2]=def
strarray[3]=ghi
strarray[4]=jkl

str=abc

dblarray[1]=3
dblarray[2]=99
dblarray[3]=99

intarray[1]=1
intarray[2]=2
intarray[3]=3
intarray[4]=4
intarray[5]=5
intarray[6]=6
intarray[7]=7
intarray[8]=8
intarray[9]=9
intarray[10]=10

dblarray[3]=5

a=100 b=101 c=102

NOTE: Execution succeeded. No rows affected.

ERROR: Invalid index value -5 for subscript 1 of array dblnegsubarray; allowed range
is [-4,-1].
ERROR: Invalid index value 0 for subscript 1 of array dblnegsubarray; allowed range
is [-4,-1].



ARRAY ASSIGNMENT
s2[1]=abc
s2[2]=def
s2[3]=ghi
s2[4]=jkl

d2[1]=3.14159
d2[2]=3.14159
d2[3]=3.14159
d2[4]=5
d2[5]=5
d2[6]=1
d2[7]=2
d2[8]=1
d2[9]=2
d2[10]=99

arr2x3[1,1]=1 arr3x2[3,-1]=1
arr2x3[1,2]=2 arr3x2[3,0]=2
arr2x3[1,3]=3 arr3x2[4,-1]=3
arr2x3[2,1]=1 arr3x2[4,0]=1
arr2x3[2,2]=2 arr3x2[5,-1]=2
arr2x3[2,3]=3 arr3x2[5,0]=3
```

Example: SELECT Statement

Example Overview: SELECT Statement

This example illustrates the SELECT statement. In this example, a DO loop encloses a SELECT statement. The SELECT statement reads the current value of the loop counter I and writes a character when the WHEN statement is true. The REPEAT statement repeatedly prints the character based on the value of the loop counter.

Example Code: SELECT Statement

```
proc ds2;
data;
   dcl char(10) s;
   method run();
      dcl char(1) x;
      dcl int i;
      do i=1 to 10;
         select(i);
            when(1) x='A';
            when(2) x='B';
            when(3) x='C';
            when(4) x='D';
            when(5) x='E';
            when(6) x='F';
            when(7) x='G';
            when(8) x='H';
            when(9) x='I';
            otherwise x='J';
         end;
         s=repeat(x,i);
         output;
      end;
   end;
enddata;
run;
quit;
```

Example Output: SELECT Statement

The SAS System

s
AA
BBB
CCCC
DDDDD
EEEEEE
FFFFFFF
GGGGGGGG
HHHHHHHHH
IIIIIIIII
JJJJJJJJJ

Example: GOTO and LEAVE Statements with Labels

Example Overview: GOTO and LEAVE Statements with Labels

This example presents three programs that show branching. The first uses GOTO to branch to a label. The second uses LEAVE without a label, and the third uses LEAVE with a label. For more information, see "GOTO Statement" on page 1253 and "LEAVE Statement" on page 1262.

Example Code: GOTO and LEAVE Statements with Labels

```
proc ds2;
data;
   dcl double i j;
   method init();
      i = 1;
   head:
      j = 2*i;
      i = i+1;
      if i < 10 then do;
         output;
         goto head;
      end;
   end;
enddata;
run;
data _null_;
   dcl int x y;
   method init();
      put 'loop test 1';
      x = 1;
      y = 2;
      if (x ~= -5) then
         do i = 1 to 10;
            put i;
            if i > 4 then leave;
         end;
      else
          put 'else';
   end;
enddata;
run;
data _null_;
   dcl int g;
   method init();
      put 'label test 1';
      x = 1;
      y = 2;
      if (x > -5) then
         lab:
         do i = 1 to 10;
            do j = 1 to 5;
               put i j;
               if i > 4 then leave lab;
            end;
         end;
      else
         do;
```

```
            put 'else';
          end;
   end;
enddata;
run;
quit;
```

Example Output: GOTO and LEAVE Statements with Labels

The following output is the result of using GOTO to branch to a label.

The SAS System

i	j
2	2
3	4
4	6
5	8
6	10
7	12
8	14
9	16

These lines from the loop test are written to the SAS log.

```
loop test 1
1
2
3
4
5
```

These lines from the label test are written to the SAS log.

```
label test 1
1 1
1 2
1 3
1 4
1 5
2 1
2 2
2 3
2 4
2 5
3 1
3 2
3 3
3 4
3 5
4 1
4 2
4 3
4 4
4 5
5 1
```

Example: Overloaded Methods

Example Overview: Overloaded Methods

This example illustrates how to set up overloaded methods. For more information about methods, see the "METHOD Statement" on page 1270.

Example Code: Overloaded Methods

```
proc ds2;
data _null_;
   method concat(nvarchar(200) x, nvarchar(200) y) returns nvarchar(400);
      return x || y;
   end;

   method concat(nvarchar(200) x, nvarchar(200) y, nvarchar(200) z)
      returns nvarchar(600);
      return x || y || z;
   end;

   method run();
      y = concat(n'abc', n'def');
      put 'y= ' y;
      y = concat(n'abc', n'def', n'ghi');
```

```
      put 'y= ' y;
   end;
enddata;
run;
data _null_;
   method d() returns double;
      return 99;
   end;

   method d(double x, double y, double z) returns double;
      return x + y + z;
   end;

   method d(int x, int y, int z) returns int;
      return x + y + z + 500;
   end;

   method run();
      dcl double d;
      d = d(1,2,3);
      put 'd= ' d;
      d = d(100.1, 100.2, 100.3);
      put 'd= ' d;
   end;
enddata;
run;
quit;
```

Example Output: Overloaded Methods

The following lines are written to the SAS log.

```
y=  abcdef
y=  abcdefghi

d=  506
d=  300.6
```

Example: Analyze a Table Using Multiple Threads

Example Overview: Analyze a Table Using Multiple Threads

This example shows the creation of a table that is then analyzed using multiple threads.

Example Code: Analyze a Table Using Multiple Threads

```
libname spde spde 'c:\temp\spde';
proc delete data=spde.incomes; run;
proc delete data=spde.results1; run;
proc delete data=spde.results2; run;

proc ds2;
data spde.incomes;
   dcl double income citycode;
   dcl char(8) name;
   method run();
      do j = 1 to 1E6;
         name = 'John';  income = 23234; citycode=1; output;
         name = 'Jane';  income = 62348; citycode=1; output;
         name = 'Joe';   income = 32932; citycode=2; output;
         name = 'Jan';   income = 58239; citycode=2; output;
         name = 'Josh';  income =  6523; citycode=3; output;
         name = 'Jill';  income = 80392; citycode=3; output;
         /* The three people to find during mining */
         if j = 5E5 then do;
           name = 'James'; income = 103243; citycode=1; output;
         end;
         if j = 7E5 then do;
           name = 'Joan'; income = 233923; citycode=2; output;
         end;
         if j = 8E5 then do;
           name = 'Joyce'; income = 132443; citycode=3; output;
         end;
      end;
```

```
   end;
enddata;
run;

thread score;
   method run();
      set spde.incomes;

      accept = 0;
      if citycode = 1 and income > 100000 then accept = 1;
      else if citycode = 2 and income > 200000 then accept = 1;
      else if citycode = 3 and income > 120000 then accept = 1;

      /* faux work */
      do i = 1 to 50; end;

      if accept then output;
   end;
endthread;
run;

data spde.results1;
   dcl thread score score;
   method run();
      set from score threads=1;
   end;
enddata;
run;

data spde.results2;
   dcl thread score score;
   method run();
      set from score threads=2;
   end;
enddata;
run;
quit;
```

Example Output: Analyze a Table Using Multiple Threads

The following lines are written to the SAS log.

```
NOTE: Execution succeeded. 6000003 rows affected.
NOTE: Execution succeeded. No rows affected.
NOTE: Execution succeeded. 3 rows affected.
NOTE: Execution succeeded. 3 rows affected.
```

Example: Using Four Threads to Compute Summary Statistics

Example Overview: Using Four Threads to Compute Summary Statistics

The following example creates a thread program that computes some summary statistics. The thread program is run in four threads. The program looks at which thread generates which values.

Example Code: Using Four Threads to Compute Summary Statistics

```
/* Expand sashelp.class */
data class;
   do i = 1 to 1E5;
      do j = 1 to nobs;
         set sashelp.class nobs=nobs point=j;
         output;
      end;
   end;
   stop;
run;

proc ds2;

/* Create the thread program
 * When the thread program executes in N threads, each thread receives
 * a unique set of rows from the table work.class.
 * This thread program sums all the student ages it sees along
 * with counting the number of men and women it sees.
 * Each thread's partial sum is output to a data program for
 * final summing.
 */
   thread sum_student_measures / overwrite=yes;
      dcl double id cnt sum_age cnt_male cnt_female;
      keep id cnt sum_age cnt_male cnt_female;
      method run();
         set class;
```

```
         sum_age + age;
         cnt_male + (if sex = 'M' then 1 else 0);
         cnt_female + (if sex = 'F' then 1 else 0);
      end;

      method term();
      /* _threadid_ is the thread's "number" from 0 to N-1, when using N threads. */
         id = _threadid_;
         cnt = _N_;
         output;
      end;
   endthread;
run;

/* Start the thread program in 4 threads, sum the values
 * received and output averages and total counts.
 * The variable ID is the thread number for the thread that
 * produced a particular set of counts.  This is useful
 * in looking at which thread output which values.
 * Note in the output how the PUT statement output isn't ordered
 * by ID.  This is because some threads finish sooner than others.
 * Also notice the variable CNT which indicates that different
 * threads operate on different numbers of rows.
 */
data class_counts / overwrite=yes;
   dcl thread sum_student_measures t();
   dcl double tot_cnt avg_age tot_male tot_female tot_age;
   keep tot_count avg_age tot_male tot_female;
   method run();
      set from t threads=4;
      put id= cnt= sum_age= cnt_male= cnt_female=;

      tot_age + sum_age;
      tot_male + cnt_male;
      tot_female + cnt_female;
      tot_cnt + cnt;
   end;

   method term();
      avg_age = tot_age / tot_cnt;
      output;
   end;
enddata;
run;
quit;

proc print data=class_counts; run;
```

Example Output: Using Four Threads to Compute Summary Statistics

The following lines are written to the SAS log. Note that every time you run the program, the log output is different.

```
id=2 cnt=1 sum_age=0 cnt_male=0 cnt_female=0
id=0 cnt=1404506 sum_age=18702097 cnt_male=739215 cnt_female=665290
id=3 cnt=1 sum_age=0 cnt_male=0 cnt_female=0
id=1 cnt=495496 sum_age=6597903 cnt_male=260785 cnt_female=234710
```

The following table is generated.

Obs	avg_age	tot_male	tot_female
1	13.3158	1000000	900000

Example: Data Cleaning

Example Overview: Data Cleaning

Sometimes, due to input error, data values might have leading and trailing blanks. Unless you specifically filter for this possibility, joins and similar database operations that depend on identically formatted data values will not perform correctly. Another problem that can occur is unintentional duplication of records (that is, multiple records (rows) with the same key). This example shows programs that remove blanks from data values, and list duplicate rows for potential deletion.

The first DS2 program creates a simple data table, EMPLOYEES1, that contains duplicate rows and string values. Some of the rows contain blanks.

The second program uses the STRIP function to remove leading and trailing blanks from values in the EMP column. The table is written to EMPLOYEES2.

The third program uses an embedded SQL SELECT statement to display the duplicates. You can use this output to determine which records should be deleted. Note that this program would have not generated correct output if you did not first remove the blanks from the data values, because the SELECT statement would not have grouped the data accurately.

Example Code: Data Cleaning

```
proc ds2;
data employees1 (overwrite=yes);
   dcl double id;
   dcl char emp;
   method init();
      id =  60918 ; emp = 'user1'; output;
      id =  60919 ; emp = ' user2'; output;
      id =  60920 ; emp = ' user3'; output;
      id =  60918 ; emp = 'user1'; output;
      id =  60922 ; emp = 'user4'; output;
      id =  60925 ; emp = ' user5 '; output;
      id =  60926 ; emp = 'user6'; output;
      id =  60919 ; emp = 'user2'; output;
      id =  60928 ; emp = ' user7'; output;
      id =  60918 ; emp = 'user1'; output;
   end;
enddata;
run;
data employees2 (overwrite=yes);

   method run();
      set employees1;
      emp = strip(emp);
   end;
enddata;
run;
data ;
   method run();
      set {select id as dupid,  count(id) as dups
         from  employees2
         group by id
         having count(id) > 1};

   end;
enddata;
run;
quit;
```

Example Output: Data Cleaning

The SAS System

DUPID	DUPS
60918	3
60919	2

Example: SUBSTR Function

Example Overview: SUBSTR Function

The example shows how to use the SUBSTR function. SUBSTR is used to convert the word CAT to DOG by changing one character at a time.

Example Code: SUBSTR Function

```
proc ds2;
data _null_;
   dcl char(3) a b;
   method init();
      a = 'cat';
      put 'a=' a;

      substr(a,2,1) = 'o';
      put 'a=' a;

      substr(a,1,1) = 'd';
      put 'a=' a;

      substr(a,3,1) = 'g';
      put 'a=' a;
      b = a;
   end;
enddata;
run;
```

Example Output: SUBSTR Function

The following lines are written to the SAS log.

```
a= cat
a= cot
a= dot
a= dog
```

Example: PUT with SAS Formats

Example Overview: PUT with SAS Formats

This example illustrates how to use SAS formats with the PUT statement.

Example Code: PUT with SAS Formats

```
proc ds2;
data _null_;
   method init();
      dcl double d;
      dcl int i;
      dcl char(32) x;

      d = 99;
      x = put(d, best12.6);
      put 'x=' x;

      d = 100;
      x = put(d, best.);
      put 'x=' x;

      i = 10847;
      x = put(i, mmddyy8.);
      put 'x=' x;

      c = 'abc';
      x = put(c, $char8.);
      put 'x=' x;
```

```
   end;
enddata;
run;
quit;
```

Example Output: PUT with SAS Formats

The following lines are written to the SAS log.

```
x=           99
x=          100
x= 09/12/89
x= abc
```

Example: Generate Statistics from Table Data

Example Overview: Generate Statistics from Table Data

This example analyzes the price movement of a simulated stock table. The first DS2 program creates the stock price table. There are two weeks of the open, high, low, and close prices for the stock.

The second program performs the analysis. A 2–day moving average of closing prices is created by assigning the previous two days' closing prices to elements of the CY array, averaging them by using the MEAN function, and then sending them to output as C_MA. The INIT method initializes the array CY and C_MA variables to the first price in the STOCK table. The RUN method assigns values to the array as it loops through the table.

The CTR variable is incremented each time through the RUN method. The moving average is not valid until two days of prices have been averaged, so the output begins with the third record (that is, when CTR > 2). After a row has been written, the CY array is updated.

The following statistics are also calculated:

- the daily change in closing price, **Chng**, calculated by subtracting yesterday's close, CY[1], from today's close
- the change from open to the closing price, O_C

- the range of the day's prices, H_L, calculated by subtracting the low price from the high price

Example Code: Generate Statistics from Table Data

```
proc ds2;
data stock (overwrite=yes);
   dcl date d;
   dcl double o h l c;
   method init();
      d = date '2010-09-18' ; o =  20; h = 22.25; l = 18; c = 21.5; output;
      d = date '2010-09-19' ; o =  21; h = 23.5; l = 19.25; c = 22; output;
      d = date '2010-09-20' ; o =  22.25; h = 24.75; l = 20; c = 21; output;
      d = date '2010-09-21' ; o =  21; h = 21.5; l = 18.75; c = 19; output;
      d = date '2010-09-22' ; o =  18; h = 19; l = 17.25; c = 17; output;
      d = date '2010-09-25' ; o =  17; h = 18; l = 15.5; c = 17; output;
      d = date '2010-09-26' ; o =  17.5; h = 20; l = 16; c = 18; output;
      d = date '2010-09-27' ; o =  18.5; h = 21; l = 18; c = 20; output;
      d = date '2010-09-28' ; o =  20; h = 22.25; l = 19.5; c = 21; output;
      d = date '2010-09-29' ; o =  21; h = 23; l = 18.75; c = 21.5; output;
   end;
enddata;
run;
data;
   keep d o h l c Chng O_C H_L C_MA;
   dcl double cy[2] C_MA H_L Chng O_C;
   dcl int ctr;
   method init();
      set stock (locktable=share);
      c_ma = c;
      cy[1] = c;
      cy[2] = c;
   end;

   method run();
      set stock (locktable=share);
      ctr+1;
      C_MA = mean(cy[1], cy[2]);
      H_L = h - l;
      O_C = o - c;
      Chng = c - cy[1];
      if ctr > 2 then output;
         cy[2] = cy[1];
         cy[1] = c;
   end;
enddata;
run;
```

Example Output: Generate Statistics from Table Data

The SAS System

d	o	h	l	c	Chng	O_C	H_L	C_MA
20SEP2010	22.25	24.75	20	21	-1	1.25	4.75	21.75
21SEP2010	21	21.5	18.75	19	-2	2	2.75	21.5
22SEP2010	18	19	17.25	17	-2	1	1.75	20
25SEP2010	17	18	15.5	17	0	0	2.5	18
26SEP2010	17.5	20	16	18	1	-0.5	4	17
27SEP2010	18.5	21	18	20	2	-1.5	3	17.5
28SEP2010	20	22.25	19.5	21	1	-1	2.75	19
29SEP2010	21	23	18.75	21.5	0.5	-0.5	4.25	20.5

Example: Matrices and Non-linear Equations

Example Overview: Matrices and Non-linear Equations

This example solves a system of non-linear equations using Newton's iterative process.

In this example, the root (to within a given epsilon) is found after five iterations of the loop, and is equal to (.5, 0, –.5236). Zero appears as an extremely small number.

One interesting feature to note in Newton's example is how the result matrix can be reused. In a simple method calculation, `r=m.mult(m2);` the result matrix instance is automatically created. However, if the result instance already exists, and its size is correct, the method reuses the result matrix for efficiency purposes. Note how this is done in Newton's computation loop. When the matrix is created the first time, the

left side result is reused in computations such as `ji=jm.inverse();`. Otherwise, you would have to free the old matrix and allocate a new one each time.

Example Code: Matrices and Non-linear Equations

```
proc ds2;
/*
 * Solve the system of equations
 *
 *   3*x1 + cos(x2*x3) - .5                = 0
 *   x1^2 - 81(x2 + .1)^2 + sin(x3) + 1.06 = 0
 *   e^(-x1*x2) + 20*x3 + (10pi-3)/3       = 0
 *
 * using Newton's method:
 *
 *   X_m = X_m0 - J^-1(X_m0) * F(X_m0)
 *
 */
data _null_;

   /* Infinity norm */

   method norm(double x[3]) returns double;
      return max(abs(x[3]), max(abs(x[1]), abs(x[2])));
   end;

   /* Pi computation */

   method pi() returns double;
      return atan(1)*4;
   end;

   /*
    *  f1(x1, x2, x3)
    *  f2(x1, x2, x3)
    *  f3(x1, x2, x3)
    */

   method compute_f(double f[3,1], double x[3]);
      f[1,1] = 3*x[1] - cos(x[2]*x[3]) - .5;
      f[2,1] = x[1]**2 - 81*(x[2]+.1)**2 + sin(x[3]) + 1.06;
      f[3,1] = exp(-x[1]*x[2]) + 20*x[3] + (10*pi() - 3)/3.0;
   end;


   /*
    *      Jacobian array
    *
    *     @f1() @f1() @f1()
    *     ---   ---   ---
    *     @x1   @x2   @x3
    *
    *     @f2() @f2() @f2()
```

```
 *     ---   ---   ---
 *     @x1   @x2   @x3
 *
 *     @f3() @f3() @f3()
 *     ---   ---   ---
 *     @x1   @x2   @x3
 */

method compute_j(double j[3,3], double x[3]);
   j[1,1] = 3;
   j[1,2] = x[3]*sin(x[2]*x[3]);
   j[1,3] = x[2]*sin(x[2]*x[3]);
   j[2,1] = 2*x[1];
   j[2,2] = -162*(x[2]+.1);
   j[2,3] = cos(x[3]);
   j[3,1] = -x[2]*exp(-x[1]*x[2]);
   j[3,2] = -x[1]*exp(-x[1]*x[2]);
   j[3,3] = 20;
end;

method init();
   dcl double j[3,3];
   dcl double f[3,1];
   dcl double y[3,1];
   dcl double x[3];
   dcl double x0[3];
   dcl double d[3];
   dcl package matrix jm;
   dcl package matrix ji;
   dcl package matrix fm;
   dcl package matrix ym;
   dcl package matrix xm0;
   dcl package matrix xm;
   dcl package matrix diff;
   dcl int niter;
   dcl double eps;

  /* Instantiate matrices */

   jm = _new_ matrix(3, 3);
   fm = _new_ matrix(3, 1);
   xm0 = _new_ matrix(3, 1);

  /* Initial approximation */

   x0[1] = .1;
   x0[2] = .1;
   x0[3] = -.1;

  /* Start loop */

   eps = 1;
   niter = 0;

   put '(' x0[1] ',' x0[2]',' x0[3] ')';
```

```
      do while(eps > 10**-6 and niter < 10);

      /* Compute functions with current approximation : j(x_0), f(x_0)
*/

         compute_j(j, x0);
         compute_f(f, x0);

       /* Load arrays into matrices */

         jm.load(j);
         fm.load(f);
         xm0.load(x0);

       /* Find inverse of Jacobian matrix */

         ji = jm.inverse();

       /* Multiply by function vector */

         ym = ji.mult(fm);

       /* Compute next approximation */

         xm = xm0.sub(ym);

       /* Compute error term */

         xm.toarray(x);
         diff = xm.sub(xm0);
         diff.toarray(d);
         eps = norm(d);

         x0 := x;
         put '(' x[1] ',' x[2]',' x[3] ')';
         put eps=;
         niter + 1;
      end;

      put niter=;
      put 'root = ('x[1] ',' x[2]',' x[3] ')';
   end;
enddata;
run;
quit;
```

Example Output: Matrices and Non-linear Equations

The following results appear in the log:

```
( 0.1 , 0.1 , -0.1 )
( 0.49986967292642 , 0.01946684853741 , -0.52152047193583 )
eps=0.42152047193583
( 0.50001424016421 , 0.00158859137029 , -0.52355696434763 )
eps=0.01787825716712
( 0.50000011346783 , 0.00001244478332 , -0.52359845007288 )
eps=0.00157614658697
( 0.50000000000707 , 7.7578571058927E-10 , -0.523598775578 )
eps=0.00001244400753
( 0.5 , -2.1250842759061E-18 , -0.52359877559829 )
eps=7.7578571271436E-10
niter=5
root = ( 0.5 , -2.1250842759061E-18 , -0.52359877559829 )
```

Example: DS2 Matrices and Regression Analysis

Example Overview: DS2 Matrices and Regression Analysis

This example uses DS2 matrices to find the parameter estimate for a regression analysis.

Example Code: DS2 Matrices and Regression Analysis

```
libname z 'c:\mylib';

data _null_;
   dsid = open('z.hmeq4');
   nobs = attrn(dsid, 'nobs');
   call symput('nobs', nobs);
   hmnv = attrn(dsid, 'nvars');
   call symput ('hmnv', hmnv);
run;

/*
 * This does a regression approximation of the hmeq model variable
 * MORTDUE using the 'independent' variables CLAGE, CLNO, DELINQ,
 * DEROG, NINQ, VALUE and YOJ.
 *
 * We set up a matrix X with the data for the independent variables,
```

```
 * and a matrix Y with the dependent data, and then solve for the
linear
 * coefficients b in y = X * b:
 *
 *     X'*y = X'*X*b
 *    (X'*X)^-1 * X'*y = b
 */

proc ds2;
   data y(overwrite=yes);
      vararray double s[&hmnv] i clage clno delinq derog ninq value
yoj;
      vararray double m[1] mortdue;

      method init();
         dcl package matrix x ym y xtx ti xt bm;
         dcl double b[&hmnv];

         /* Create the hmeq and mortdue matrices */
         x = _new_ matrix(&nobs, &hmnv);
         y = _new_ matrix(&nobs, 1);

         /* Read the data from the hmeq table */
         do j = 1 to &nobs
            set z.hmeq4;
            x.in(s, j);
         end;

     /*
      * x now contains the rows for the variables in hmeq -
      * plus a column of 1's for the constant term in the
approximation:
      *
      *  i   clage     clno    delinq   derog     ninq     value
yoj
      *
      *  1   94.367   9.0000   0.0000  0.00000   1.0000   39025.00
10.5000
      *  1  121.833  14.0000   2.0000  0.00000   0.0000   68400.00
7.0000
      *  1  149.467  10.0000   0.0000  0.00000   1.0000   16700.00
4.0000
      *  1  179.766  21.2961   0.4494  0.25457   1.1861  101776.05
8.9223
      *
      *              etc.
      */

     /* Read the data from the mortdue table */
        do j = 1 to &nobs
           set z.mortdue;
           y.in(m, j);
        end;

     /*
      * y is now a column vector containing the known values of MORTDUE
```

```
      */

     /* Make sure the row count matches */
        xr = x.rows();
        er = y.rows();

        if (xr ne er) then do;
           put 'invalid data';
           stop;
        end;

     /* Compute ti = (X'*X)^-1 */
        xt = x.trans();
        xtx = xt.mult(x);
        ti = xtx.inverse();

     /* Compute ym = X'*y */
        ym = xt.mult(y);

     /* Compute b = (X'*X)^-1 * X'y */
        bm = ti.mult(ym);

     /*
      * Thus
      *
      *   MORTDUE = b_0 + b_1 * CLAGE + b_2 * CLNO +
      *             b_3 * DELINQ + b_4 * DEROG + b_5 * NINQ +
      *             b_6 * VALUE + b_7 * YOJ + eps
      *
      * The b vector is equivalent to the parameter estimate from
      *
      *   proc reg; model mortdue= clage clno delinq derog ninq value
yoj;run;
      *
      *   Variable     DF    Parameter Estimate   F Value 1434.92
      *
      *   Intercept     1          11473
      *   clage         1       -1.64128
      *   clno          1      466.74925
      *   delinq        1      -46.87948
      *   derog         1    -1371.68909
      *   ninq          1      544.05978
      *   value         1        0.56118
      *   yoj           1     -530.88585
      */

        bm.toarray(b);
        do i = 1 to &hmnv
           put b[i];
        end;
     end;
enddata;
run;
quit;
```

Example Output: DS2 Matrices and Regression Analysis

The following results appear in the log.

```
11472.5599166895
-1.64128448706739
466.749252257557
-46.8794774759862
-1371.68908534735
544.059779616903
0.56117547945582
-530.885851993411
```

Example: Use the SQLSTMT Package in Another Package

Example Overview: Use the SQLSTMT Package in Another Package

This example shows how to use an instance of a package SQLSTMT in another package. It also shows the use of the _NEW_ operator to delay construction of the SQLSTMT instance until after the creation of the table that is referenced by the FedSQL statement. If the SQLSTMT instance was constructed before the referenced table was created, then the prepare of the FedSQL statement fails in the constructor.

Example Code: Use the SQLSTMT Package in Another Package

```
package fibonnaci;
   declare int nMax;    /* maximum index in this series */

   method fibonnaci(int n);
      nMax = n;
```

```
   end;

   method output(char(100) tablename);
      declare int    n;    /* index of fib in Fibonacci series */
      declare double fib;  /* Fibonacci number */

      declare package sqlstmt stmt;
      /* note that stmt is not created */

      sqlexec('create table ' || tablename || '
         (n int, fib double)');

      /* Using _new_ allows delay of prepare of SQL statement until
after */
      /* output table is created */

      stmt = _new_ sqlstmt('insert into ' || tablename || '
         values (?, ?)');

      /* fibonnaci(0) = 0 */
      stmt.setInteger(1, 0);     /* column n */
      stmt.setDouble(2, 0);      /* column fib */
      stmt.execute();

      /* fibonnaci(1) = 1 */
      stmt.setInteger(1, 1);     /* column n */
      stmt.setDouble(2, 2);      /* column fib */
      stmt.execute();

      first = 0;
      second = 1;

      do n = 2 to nMax;
         fib = first + second;
         stmt.setInteger(1, n);
         stmt.setDouble(2, fib);
         stmt.execute();

         first = second;
         second = fib;
      end;
   end;
endpackage;
```

The following code block creates an instance of package FIBONACCI, and the Fibonacci instance is used to generate an output table of a Fibonacci series.

```
data _null_;
   method init();
      declare package fibonnaci fibseries(20);
      fibseries.output('fibdata');
   end;
enddata;
run;
```

Example: Update the Values of a Table By Using Two Databases

Example Overview: Update the Values of a Table By Using Two Databases

The following example illustrates how the SQLSTMT package can facilitate updating a table in one database based on the values in another table in a second database. In the example program, stmt1 queries for all the x and y columns from the ORACLE table db1.dataset1. Then for each rowset in the result set from db1.dataset1, stmt2 finds the rows in BASE table db2.dataset2 that have the same x column values as the x value read from db1.dataset1. Stmt2 then updates the BASE table db2.dataset2 y column values of the found rows to be the same as the y value read from db1.dataset1.

Example Code: Update the Values of a Table By Using Two Databases

```
libname db1 odbc user=XXXX pw=XXXX dsn=exadat;
libname db2 './base';

proc ds2;
data _null_;
   declare double x y;
   method run();
      dcl package sqlstmt stmt1('select x,y from db1.dataset1');
      dcl package sqlstmt stmt2('update db2.dataset2 set y=? where
x=?',
          [y x]);

      stmt1.execute();
      stmt1.bindResults([x y]);
      do while (stmt1.fetch() = 0);
         stmt2.execute();
      end;
   end;
enddata;
run;
quit;
```

Example: Store FedSQL Statements in a Hash Package

Example Overview: Store FedSQL Statements in a Hash Instance

The following example shows how to use a hash package to manage a set of FedSQL statements. A set of FedSQL statements is dynamically allocated and stored in a hash package. The hash package is indexed by an integer key that is used to retrieve the appropriate FedSQL statement from the hash package.

Example Code: Store FedSQL Statements in a Hash Package

```
proc ds2;
/* Create 5 tables testdata1...testdata5.
 * Each table has 3 double columns (x,y,z) and no rows. */
data _null_;
   method init();
      declare int i;
      declare int rc;

      do i = 1 to 5;
         rc = sqlexec('create table testdata' || i ||
            '(x double, y double, z double)');
         if (rc ne 0) then put 'TEST FAILED';
      end;
   end;
enddata;
run;
quit;

proc ds2;
data _null_;

   /* Create an SQLstmt reference.
    * Note: does NOT create an sqlstmt instance. */
   declare package sqlstmt s;
   /* Create a hash instance. */
   declare package hash h();
```

```
/* Hash key: sqlstmts are accessed by index 1..5. */
declare int  i;
/* Variables to bind to sqlstmt parameters. */
declare double u v w;


/* Create 5 sqlstmts to insert data in tables
 * testdata1...testdata5. Store the sqlstmts in hash
 * table h. */

method init();
   declare int rc;

   h.definekey('i');    /* Key is index 1..5. */
   h.definedata('s');   /* Data is an sqlstmt reference. */
   h.definedone();

   /* Dynamically create 5 sqlstmt instances and add
    * each sqlstmt to hash table. */
   do i = 1 to 5;

      /* Dynamcially create an sqlstmt.
       * Variables [u v w] are bound to the parameters of
       *the sqlstmt. */
      s = _new_ sqlstmt('insert into testdata'
         || i || ' values (?,?,?)', [u v w]);

      /* Add the sqlstmt to hash h with key i. */
      rc = h.add();
      if (rc ne 0) then put 'TEST FAILED';
    end;
 end;


/* Retrieve the sqlstmts from the hash table and execute
 * the sqlstmts to insert rows in the tables. */

method run();
   declare int j;
   declare int rc;

   /* For each of 5 data tables testdata1...testdata5, */
   /* insert 9 rows. */
   do j = 1 to 9;
      do i = 1 to 5;

         /* Find the sqlstmt in hash by index i. */
         rc = h.find();
         if (rc ne 0) then put 'TEST FAILED';

         /* Set values to insert into the table testdata i. */
         u = i;        /* data for 1st column (x) */
         v = -j;       /* data for 2nd column (y) */
         w = i*10+j;   /* data for 3rd column (z) */
```

```
            /* Execute the sqlstmt to insert the row. */
            rc = s.execute();
            if (rc ne 0) then put 'TEST FAILED';

            /* Explicitly close the result set to free resources.
             * If not explicitly closed, result set would be
             * automatically closed at next execute of the sqlstmt. */
            rc = s.closeResults();
            if (rc ne 0) then put 'TEST FAILED';

         end;
      end;
   end;


   /* Explicitly delete each sqlstmt instance.  Note if not
    * explicitly deleted, the sqlstmt instances would automatically
    * be deleted when the sqlstmt variables referencing the instances
    * were deleted. */

   method term();
      declare int rc;

      do i = 1 to 5;
         rc = h.find();
         if (rc ne 0) then put 'TEST FAILED';
         s.delete();
      end;
   end;
enddata;
run;
quit;


proc print data=testdata1; quit;
proc print data=testdata2; quit;
proc print data=testdata3; quit;
proc print data=testdata4; quit;
proc print data=testdata5; quit;
```

Example Output: Store FedSQL Statements in a Hash Package

Output A20.1 *Testdata Tables*

Obs	X	Y	Z
1	1	-1	11
2	1	-2	12
3	1	-3	13
4	1	-4	14
5	1	-5	15
6	1	-6	16
7	1	-7	17
8	1	-8	18
9	1	-9	19

Obs	X	Y	Z
1	2	-1	21
2	2	-2	22
3	2	-3	23
4	2	-4	24
5	2	-5	25
6	2	-6	26
7	2	-7	27
8	2	-8	28
9	2	-9	29

Obs	X	Y	Z
1	3	-1	31
2	3	-2	32
3	3	-3	33
4	3	-4	34
5	3	-5	35
6	3	-6	36
7	3	-7	37
8	3	-8	38
9	3	-9	39

Obs	X	Y	Z
1	4	-1	41
2	4	-2	42
3	4	-3	43
4	4	-4	44
5	4	-5	45
6	4	-6	46
7	4	-7	47
8	4	-8	48
9	4	-9	49

Obs	X	Y	Z
1	5	-1	51
2	5	-2	52
3	5	-3	53
4	5	-4	54
5	5	-5	55
6	5	-6	56
7	5	-7	57
8	5	-8	58
9	5	-9	59

Example: Run a DS2 Program from z/OS Batch

Example Overview: Run a DS2 Program from z/OS Batch

The following example illustrates how to run a basic DS2 program from z/OS batch.

Example Code: Run a DS2 Program from z/OS Batch

```
//YOUR JOB BATCH INFO HERE
//TIME=1,NOTIFY=
/*JOBPARM FETCH
//VALID EXEC SAS94
//WORK DD PATH='/u/myuserid/mywork/'
//SYSIN DD *
proc ds2;
data departure (overwrite=yes);
   dcl char(9) airline;
   dcl char(5) arrtime;
   method init();
      airline='Southwest'; arrtime='10:30'; output;
      airline='Delta'; arrtime='12:45'; output;
      airline='American'; arrtime='11:15'; output;
   end;
enddata;
run;
quit;
proc print data=departure;
run;
quit;
```

Example: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File

Example Overview: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File

This example uses an HTTP and JSON package to extract information from a REST formatted file.

Example Code: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File

```
/* DS2 program that uses a REST-based API.    */
/* Uses http package for API calls           */
/* and the JSON package to parse the result.  */

proc ds2;
thread work.json / overwrite = yes;
    /* Global package references             */
    dcl package json j();

    /* Keeping these variables for output    */
    dcl double timest having format datetime20.;
    dcl varchar(50) name;
    dcl int lat lng bikes free;

    /* these are temp variables */
    dcl varchar(65534) character set utf8 response;
    dcl int rc;
    drop response rc;

    method parseMessages();
      dcl int tokenType parseFlags;
      dcl nvarchar(128) token;
      rc=0;
```

```
  * iterate over all message entries;
  do while (rc=0);
    j.getNextToken( rc, token, tokenType, parseFlags);

        if (token eq 'timestamp') then
      do;
        j.getNextToken( rc, token, tokenType, parseFlags);
        timest=inputn(token,'anydtdtm26.');
      end;

    if (token eq 'name') then
      do;
        j.getNextToken( rc, token, tokenType, parseFlags);
        name=token;
      end;

    if (token eq 'lat') then
      do;
        j.getNextToken( rc, token, tokenType, parseFlags);
        lat=token;
      end;

    if (token eq 'lng') then
      do;
        j.getNextToken( rc, token, tokenType, parseFlags);
        lng=token;
      end;

    if (token eq 'bikes') then
      do;
        j.getNextToken( rc, token, tokenType, parseFlags);
        bikes=token;
      end;

     if (token eq 'free') then
      do;
        j.getNextToken( rc, token, tokenType, parseFlags);
        free=token;
        output;
      end;
  end;
  return;
end;

method init();
  dcl package http webQuery();
  dcl int rc tokenType parseFlags;
  dcl nvarchar(128) token;
  dcl integer i rc;

  /* create a GET call to the API                */
  /* 'sas_programming' covers all SAS programming */
  /* topics from communities                     */

  webQuery.createGetMethod(
     'http://api.citybik.es/capital-bikeshare.json');
```

```
      /* execute the GET                                */

      webQuery.executeMethod();

      /* retrieve the response body as a string       */

      webQuery.getResponseBodyAsString(response, rc);
      put response;
      rc = j.createParser( response );
      do while (rc = 0);
        j.getNextToken( rc, token, tokenType, parseFlags);
        if (token = '{') then
          parseMessages();
      end;
    end;

  method term();
    rc = j.destroyParser();
  end;
endthread;

data bikes (overwrite=yes);
declare thread work.json json;
method run();
   set from json threads=3;
end;
enddata;
run;
quit;
```

Example Output: Using an HTTP and JSON Package to Extract and Parse a REST-Formatted File

Here are a few of the lines that are written to the SAS log:

```
[
  {
    "bikes": 10,
    "name": "Jones Branch & Westbranch Dr",
    "idx": 0,
    "lat":
38931911,
    "timestamp": "2017-09-13T18:21:46.986000Z",
    "lng": -77219261,
    "id": 0,

   "free": 3,
    "number": 430
  },
  {
    "bikes": 7,
    "name": "Town Center Pkwy &
Bowman Towne Dr",
    "idx": 1,
    "lat": 38962524,
    "timestamp":
"2017-09-13T18:21:46.979000Z",
    "lng": -77361902,
    "id": 1,
    "free": 8,

"number": 434
  },
  {
    "bikes": 17,
    "name": "Potomac & M St NW",
    "idx": 2,

"lat": 38905368,
    "timestamp": "2017-09-13T18:21:47.384000Z",
    "lng": -77065149,

"id": 2,
    "free": 1,
    "number": 473
  },
  {
    "bikes": 7,
    "name": "22nd St &
Constitution Ave NW",
    "idx": 3,
    "lat": 38892441,
    "timestamp":
"2017-09-13T18:21:46.931000Z",
    "lng": -77048947,
    "id": 3,
    "free": 15,

"number": 443
  },
  {
    "bikes": 2,
    "name": "18th & Eads St.",
    "idx": 4,

"lat": 38857250,
    "timestamp": "2017-09-13T18:21:47.160000Z",
    "lng": -77053320,

"id": 4,
    "free": 8,
    "number": 2
  },
  {
    "bikes": 3,
    "name": "15th & Crystal
Dr",
    "idx": 5,
```

Example: Reading JSON Text from HTTP and Creating a SAS Data Set

Example Overview: Reading JSON Text from HTTP and Creating a SAS Data Set

This example uses the HTTP package to read some customer-supplied JSON text from http://httpbin.org/post. The example then uses the JSON package to parse the response into a SAS data set.

Example Code: Reading JSON Text from HTTP and Creating a SAS Data Set

```
proc ds2;
 package myhttp /overwrite=yes;
   method post( nvarchar(8192) url,
                nvarchar(67108864) payload,
                in_out nvarchar respbody,
                in_out int hstat, in_out int rc );
    dcl package http h();
    dcl package logger logr( 'App.TableServices.d2pkg.HTTP' );
    dcl nvarchar(8192) respHdrs;
    rc = h.createPostMethod( url );
    if rc ne 0 then goto Exit;
    rc = h.setRequestContentType( 'application/json' );
    if rc ne 0 then goto Exit;
    rc = h.addRequestHeader( 'Accept', 'application/json' );
    if rc ne 0 then goto Exit;
    rc = h.setRequestBodyAsString( payload );
    if rc ne 0 then goto Exit;
    rc = h.executeMethod();
    if rc ne 0 then goto Exit;
    hstat = h.getStatusCode();
    h.getResponseHeadersAsString( respHdrs, rc );
    logr.log( 3, '--- Beginning of response headers ---' );
    logr.log( 3, respHdrs );
    logr.log( 3, '--- End of response headers -----------' );
    if hstat lt 400 then h.getResponseBodyAsString( respbody, rc );
    else respbody = '';
    Exit:
```

```
    h.delete();
  end;
 endpackage;
run;
quit;

proc ds2;
 ds2_options sas;

 data work.Shots(keep=(ShotId text PName DPO _count));
  dcl package json j();
  dcl int ShotId;
  dcl varchar(256) text PName;
  dcl int DPO _count;

  method init();
    dcl package myhttp p();
    dcl nvarchar(10000) body;
    dcl nvarchar(100000) response;
    dcl nvarchar(256) token;
    dcl int rc hstat tokenType parseFlags;
    rc = 0; hstat = 0;
    body =
        '{"_count":882, "Date":"2016-02-11",
          "Shots":[{"ShotId":14162,"text":"E","PName":"NG",
                    "DPO":1455209,"_count":98},
                   {"ShotId":14163,"text":"EN","PName":"NGG",
                    "DPO":1455210,"_count":784} ],
          "Defs": [{"DT":"P","osure":"SHEET","rioType":"EUR",
                    "DId":"e2938c74","_count":98},
                   {"DT":"N","osure":"VITY","SCat":"NGE",
                    "rioType":"EUR","DId":"faaa8b91","_count":784}],
          "VIds": [{"VId":"e2938c74"}, {"VId":"faaa8b91"} ] }';

    *Bounce the JSON payload off of a test server.;
    p.post( 'http://httpbin.org/post', body, response, hstat, rc );
    if ( hstat ~= 200 ) then do;
        put rc= hstat=;
        put 'Error: JSON payload did not bouce off of the server.';
        return;
    end;
    put response=;
    rc = j.createParser( response );

    * Look for a JSON object in the response.;
    do while ( rc < 101 );
        j.getNextToken( rc, token, tokenType, parseFlags );
        *put token=;
        if ( j.isLabel(tokenType, parseFlags) and
             (token EQ 'json') ) then
            goto LookForShots;
    end;
    put 'Error: no "json" in response.';
    return;

   LookForShots:
```

```
do while ( rc < 101 );
    j.getNextToken( rc, token, tokenType, parseFlags );
    *put token=;
    if ( j.isLabel(tokenType, parseFlags) and
         (token EQ 'Shots') ) then
        goto ShotsArray;
end;
put 'Error: "Shots" not found.';
return;

ShotsArray:
j.getNextToken( rc, token, tokenType, parseFlags );
if ( not j.isLeftBracket( tokenType ) ) then do;
    put 'Error: Expecting Shots array.';
    return;
end;

*put '----- Shots array --------------';
j.getNextToken( rc, token, tokenType, parseFlags );
*put token=;
do while ( j.isLeftBrace( tokenType ) );
    *put '-------OBJECT---------';
    j.getNextToken( rc, token, tokenType, parseFlags );
    *put token=;
    do while ( not j.isRightBrace( tokenType ) );
        if ( token EQ 'ShotId' ) then do;
            j.getNextToken( rc, token, tokenType, parseFlags );
            ShotId = token;
        end;
        else if ( token EQ 'text' ) then do;
            j.getNextToken( rc, token, tokenType, parseFlags );
            text = token;
        end;
        else if ( token EQ 'PName' ) then do;
            j.getNextToken( rc, token, tokenType, parseFlags );
            PName = token;
        end;
        else if ( token EQ 'DPO' ) then do;
            j.getNextToken( rc, token, tokenType, parseFlags );
            DPO = token;
        end;
        else if ( token EQ '_count' ) then do;
            j.getNextToken( rc, token, tokenType, parseFlags );
            _count = token;
        end;
        else do;
            put 'Error: Unexpected column.';
            return;
        end;
        j.getNextToken( rc, token, tokenType, parseFlags );

    end; * End of column loop.;
    output;
    j.getNextToken( rc, token, tokenType, parseFlags );

end; * End of row loop.;
```

```
    rc = j.destroyParser();
  end; * End method init;
 enddata;
run;
quit; * End of PROC DS2 step;

proc print data=work.Shots;
run;
```

Example Output: Reading JSON Text from HTTP and Creating a SAS Data Set

The SAS System

Obs	ShotId	text	PName	DPO	_count
1	14162	E	NG	1455209	98
2	14163	EN	NGG	1455210	784