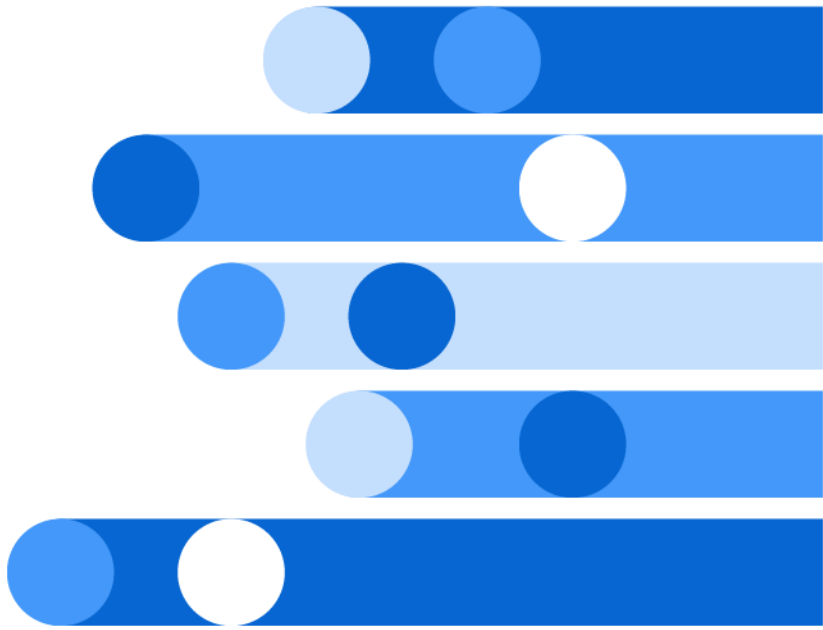
SAS® Functions and CALL Routines for SAS® Viya® Workbench



SAS® Documentation
April 14, 2026

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2024. *SAS® Functions and CALL Routines for SAS® Viya® Workbench*. Cary, NC: SAS Institute Inc.

SAS® Functions and CALL Routines for SAS® Viya® Workbench

Copyright © 2024, 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

April 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.

v_001-P1:vwblefunctionsref

Contents

About This Book . . . . . *v*

PART 1 About SAS Functions and CALL Routines 1

Chapter 1 / ***SAS Functions and CALL Routines*** **3**
Definitions of Functions and CALL Routines 5
Syntax of Functions and CALL Routines 5
Using Functions and CALL Routines 7
Function Compatibility Character Sets 19
Using Random-Number Functions and CALL Routines in the DATA Step 20
Using SYSRANDOM and SYSRANEND Macro Variables to Produce Random Number Streams 33
Hashing Functions and Hash-Based Message Authentication Code 36
Date and Time Intervals 40
Pattern Matching Using Perl Regular Expressions (PRX) 53
Using Perl Regular Expressions in the DATA Step 54
Writing Perl Debug Output to the SAS Log 65
Perl Artistic License Compliance 66
SAS Functions for Web Applications 67
Using Git Functions in SAS 67

PART 2 Functions and CALL Routines 81

Chapter 2 / ***Commonly Used Functions*** **83**
Most Commonly Used Functions 83

Chapter 3 / ***Dictionary of Functions and CALL Routines*** **87**
SAS Functions and CALL Routines Documented in Other SAS Publications 100
SAS Functions and CALL Routines by Category 100
Dictionary 139

PART 3 Appendixes 1595

Appendix 1 / ***Tables of Perl Regular Expression (PRX) Metacharacters*** **1597**
Tables of Perl Regular Expression (PRX) Metacharacters 1597

Appendix 2 / ***References*** **1609**

About This Book

Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

SAS uses standard conventions in the documentation of syntax for SAS language elements. These conventions enable you to easily identify the components of SAS syntax. The conventions can be divided into these parts:

- syntax components
- style conventions
- special characters
- references to SAS libraries and external files

Style Conventions

Style Conventions

The style conventions that are used in documenting SAS syntax include uppercase bold, uppercase, and italic:

UPPERCASE BOLD
identifies SAS keywords such as the names of functions or statements. In this example, the keyword ERROR is written in uppercase bold:

ERROR <*message*>;

UPPERCASE
identifies arguments that are literals.

In this example of the CMPMODEL= system option, the literals include BOTH, CATALOG, and XML:

CMPMODEL=BOTH | CATALOG | XML |

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

- nonliteral arguments. In this example of the LINK statement, the argument *label* is a user-supplied value and therefore appears in italic:

  LINK *label*;

- nonliteral values that are assigned to an argument.

  In this example of the FORMAT statement, the argument DEFAULT is assigned the variable *default-format*:

  FORMAT *variable(s)* <*format* > <DEFAULT = *default-format*>;

Special Characters

Special Characters

The syntax of SAS language elements can contain the following special characters:

=
an equal sign identifies a value for a literal in some language elements such as system options.

In this example of the MAPS system option, the equal sign sets the value of MAPS:

MAPS=*location-of-maps*

< >
angle brackets identify optional arguments. A required argument is not enclosed in angle brackets.

In this example of the CAT function, at least one item is required:

CAT (*item-1* <, *item-2*, ...>)

|
a vertical bar indicates that you can choose one value from a group of values. Values that are separated by the vertical bar are mutually exclusive.

In this example of the CMPMODEL= system option, you can choose only one of the arguments:

CMPMODEL=BOTH | CATALOG | XML

...
an ellipsis indicates that the argument can be repeated. If an argument and the ellipsis are enclosed in angle brackets, then the argument is optional. The repeated argument must contain punctuation if it appears before or after the argument.

In this example of the CAT function, multiple *item* arguments are allowed, and they must be separated by a comma:

CAT (*item-1* <, *item-2*, ...>)

'*value*' or "*value*"
indicates that an argument that is enclosed in single or double quotation marks must have a value that is also enclosed in single or double quotation marks.

In this example of the FOOTNOTE statement, the argument *text* is enclosed in quotation marks:

FOOTNOTE <*n*> <*ods-format-options* '*text*' | "*text*">;

;
a semicolon indicates the end of a statement or CALL routine.

In this example, each statement ends with a semicolon:

```
data namegame;
   length color name $8;
   color = 'black';
   name = 'jack';
   game = trim(color) || name;
run;
```

References to SAS Libraries and External Files

References to SAS Libraries and External Files

Many SAS statements and other language elements refer to SAS libraries and external files. You can choose whether to make the reference through a logical name (a libref or fileref) or use the physical filename enclosed in quotation marks.

If you use a logical name, you typically have a choice of using a SAS statement (LIBNAME or FILENAME) or the operating environment's control language to make the reference. Several methods of referring to SAS libraries and external files are available, and some of these methods depend on your operating environment.

In the examples that use external files, SAS documentation uses the italicized phrase *file-specification*. In the examples that use SAS libraries, SAS documentation uses the italicized phrase *SAS-library* enclosed in quotation marks:

```
infile file-specification obs = 100;
libname libref 'SAS-library';
```

PART 1

About SAS Functions and CALL Routines

Chapter 1
SAS Functions and CALL Routines 3

1

SAS Functions and CALL Routines

Definitions of Functions and CALL Routines ... 5

Syntax of Functions and CALL Routines ... 5

Using Functions and CALL Routines ... 7
- Restrictions That Affect Function Arguments ... 7
- Using the OF Operator with Temporary Arrays ... 8
- Characteristics of Target Variables ... 8
- About Descriptive Statistic Functions ... 9
- About Types of Financial Functions ... 9
- Using Pricing Functions ... 11
- Using DATA Step Functions within Macro Functions ... 11
- Invoking CALL Routines and the %SYSCALL Macro Statement ... 12
- Using Functions to Manipulate Files ... 13
- Using the VAR and V Functions ... 13

Function Compatibility Character Sets ... 19
- Overview ... 19
- I18N Level 0 ... 19
- I18N Level 1 ... 19
- I18N Level 2 ... 19

Using Random-Number Functions and CALL Routines in the DATA Step ... 20
- Overview ... 20
- Concepts ... 20
- Types of Random-Number Functions ... 21
- Pseudorandom Numbers ... 21
- Types of Random-Number Generators ... 22
- RNGs, Seeds, and Random Numbers from Distributions ... 23
- An RNG Generates Uniformly Distributed Values ... 25
- Seed Values and Reproducible Streams ... 25
- Initial Seed Values ... 26
- Are Some Seed Values Better Than Others? ... 27
- Cycles ... 27
- Overlap ... 28
- Multiple Independent Streams ... 28
- Hardware-Based RNGs ... 30
- Statistical Quality of RNGs ... 30

Statistical Quality of Mersenne Twister RNG with Certain Seeds ... 31
Summary of RNG Attributes ... 31

Using SYSRANDOM and SYSRANEND Macro Variables to Produce Random Number Streams ... **33**
Overview of the SYSRANDOM and SYSRANEND Macro Variables ... 33
The SYSRANDOM Macro Variable ... 33
The SYSRANEND Macro Variable ... 33
Example: Reproducing Results ... 34
Example: Creating a Reproducible Random Number Stream ... 35

Hashing Functions and Hash-Based Message Authentication Code ... **36**
Hashing Functions ... 36
MD5 ... 38
SHA256 ... 38
Hash-Based Message Authentication Code (HMAC) ... 39

Date and Time Intervals ... **40**
Definition of a Date and Time Interval ... 40
Interval Names and SAS Dates ... 40
Incrementing Dates and Times By Using Multipliers and By Shifting Intervals ... 40
Commonly Used Time Intervals ... 41
Retail Calendar Intervals: ISO 8601 Compliant ... 43
Custom Time Intervals ... 43
Best Practices for Custom Interval Names ... 50

Pattern Matching Using Perl Regular Expressions (PRX) ... **53**
Definition of Pattern Matching ... 53
Definition of Perl Regular Expression (PRX) Functions and CALL Routines ... 53
Benefits of Using Perl Regular Expressions in the DATA Step ... 54

Using Perl Regular Expressions in the DATA Step ... **54**
Syntax of Perl Regular Expressions ... 54
Example 1: Validating Data ... 57
Example 2: Matching and Replacing Text ... 59
Example 3: Extracting a Substring from a String ... 60
Example 4: Another Example of Extracting a Substring from a String ... 62
Example 5: Using PRX Functions to Match Capturing Groups ... 64

Writing Perl Debug Output to the SAS Log ... **65**

Perl Artistic License Compliance ... **66**

SAS Functions for Web Applications ... **67**

Using Git Functions in SAS ... **67**
Basic Workflow for SAS Git Functions ... 67
Example Git Workflow Scenarios ... 69
Advanced and Utility SAS Git Functions ... 74
Deprecated Git Functions ... 78

Definitions of Functions and CALL Routines

A *function* is a component of the SAS programming language that can accept arguments, perform a computation or other operation, and return a value. Functions return either numeric or character results. The value that is returned can be used in an assignment statement or elsewhere in expressions. Many functions are included with SAS, and you can write your own functions as well. You can use "FCMP Procedure" in *Base SAS Procedures for SAS Viya Workbench* to create customized functions.

Functions are used in DATA step programming statements, in a WHERE expression, in macro language statements, in PROC REPORT, and in Structured Query Language (SQL).

Some statistical procedures also use SAS functions. In addition, some other SAS software products offer functions that you can use in the DATA step. For more information about these functions, see the documentation that pertains to the specific SAS software product.

A *CALL routine* alters variable values or performs other system functions. CALL routines are similar to functions, but differ from functions in that you cannot use them in assignment statements or expressions.

All SAS CALL routines are invoked with CALL statements. That is, the name of the routine must appear after the keyword CALL in the CALL statement.

Syntax of Functions and CALL Routines

The syntax of a function has one of the following forms:

function-name (*argument-1*<, ...*argument-n*>)

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 SAS expression, including another function. The number and type of arguments that SAS 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), SAS writes a note to the log indicating that the argument is invalid, sets _ERROR_ to 1, and sets the result to a missing value. 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));`
- `dollars=int(cash);`
- `if sum(cash,credit)>1000 then    put 'Goal reached';`

variable-list
can be any form of a SAS 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);`

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 SAS to treat each element of the array as a separate argument.

The OF operator has been extended to accept temporary arrays. You can use temporary arrays in OF lists for most SAS functions just as you can use regular variable arrays, but there are some restrictions.

See For a list of these restrictions, see "Using the OF Operator with Temporary Arrays" on page 8.

The syntax of a CALL routine has one of the following forms:

CALL *routine-name* (*argument-1*<, ...*argument-n*>);

CALL *routine-name* (OF *variable-list*);

CALL *routine-name* (*argument-1* | OF *variable-list-1* <, ...*argument-n* | OF *variable-list-n*>);

routine-name
names a SAS CALL routine.

argument
can be a variable name, a constant, any SAS expression, an external module name, an array reference, or a function. Multiple arguments are separated by a comma. The number and type of arguments that are allowed are described with individual CALL routines in the dictionary section. Here are examples:

- `call prxsubstr(prx,string,position);`
- `call prxchange('/old/new',1+k,trim(string),result,length);`
- `call set(dsid);`
- `call ranbin(Seed_1,n,p,X1);`
- `call label(abc{j},lab);`
- `call cats(result,'abc',123);`

variable-list
can be any form of a SAS variable list, including variable names. If more than one variable list appears, separate them with a space or with a comma and another OF.

- `call cats(inventory, of y1-y15, of z1-z15);`
- `call catt(of item17-item23 pack17-pack23);`

Using Functions and CALL Routines

Restrictions That Affect Function Arguments

If the value of an argument is invalid, SAS writes a note or error message to the log and sets the result to a missing value. Here are some common restrictions for 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.
- When a numeric argument has a missing value, many functions write a note to the SAS log and return a missing value. Exceptions include some of the descriptive statistics functions and financial functions.
- For some functions, the allowed range of the arguments is platform-dependent, such as with the EXP function.

Using the OF Operator with Temporary Arrays

When you use the OF operator with temporary arrays, temporary arrays are passed to most functions whose arguments contain a varying number of parameters. You can use temporary arrays in OF lists in some functions, just as you can use temporary arrays in OF lists in regular variable arrays.

Here is an example:

```
data _null_;
   array y[10] _temporary_ (1,2,3,4,5,6,7,8,9,10);
   x = sum(of y{*});
   put x=;
run;
```

Example Code 1.1 *Log Output for the Example of Using Temporary Arrays*

```
x=55
```

These characteristics affect temporary arrays in OF lists:

- cannot be used as array indices
- can be used in functions where the number of parameters matches the number of elements in the OF list, as with regular variable arrays
- can be used in functions that take a varying number of parameters
- cannot be used with the DIF, LAG, SUBSTR, LENGTH, TRIM, or MISSING functions, nor with any of the variable information functions such as VLENGTH

Characteristics of Target Variables

Some character functions produce resulting variables, or target variables, with a default length of 200 bytes. Numeric target variables have a default length of 8 bytes. Character functions to which the default target variable lengths do not apply are shown in the following table. These functions obtain the length of the return argument based on the length of the first argument.

Table 1.1 *Functions Whose Return Argument Is Based on the Length of the First Argument*

Functions	
COMPBL	RIGHT
COMPRESS	STRIP

Functions	
DEQUOTE	SUBSTR
INPUTC	SUBSTRN
LEFT	TRANSLATE
LOWCASE	TRIM
PUTC	TRIMN
REVERSE	UPCASE

These functions show the length of the target variable if the target variable has not been assigned a length:

BYTE
: target variable is assigned a default length of 1.

INPUT
: length of the target variable is determined by the width of the informat.

PUT
: length of the target variable is determined by the width of the format.

VTYPE
: target variable is assigned a default length of 1.

VTYPEX
: target variable is assigned a default length of 1.

About Descriptive Statistic Functions

SAS provides functions that return descriptive statistics. Many of these functions correspond to the statistics produced by the MEANS and UNIVARIATE procedures. The computing method for each statistic is discussed in the elementary statistics procedures section of the *Base SAS Procedures Guide*. SAS calculates descriptive statistics for the nonmissing values of the arguments.

About Types of Financial Functions

SAS provides a group of functions that perform financial calculations. The functions are grouped into the following types:

Table 1.2 Types of Financial Functions

Function Type	Function	Description
Cash Flow	CONVX, CONVXP	Calculate convexities for cash flows
	DUR, DURP	Calculate modifies duration for cash flows
	PVP, YIELDP	Calculate present value and yield-to-maturity for a periodic cash flow
Parameter calculations	COMPOUND	Calculates compound interest parameters
	MORT	Calculates amortization parameters
Internal rate of return	INTRR, IRR	Calculate the internal rate of return
Net present and future value	NETPV, NPV	Calculate net present and future values
	SAVING	Calculates the future value of periodic saving
Depreciation	DACCxx	Calculates the accumulated depreciation up to the specified period
	DEPxxx	Calculates depreciation for a single period
Pricing	BLKSHCLPRC, BLKSHPTPRC	Calculate call prices and put prices for European options on stocks, based on the Black-Scholes model
	BLACKCLPRC , BLACKPTPRC	Calculate call prices and put prices for European options on futures, based on the Black model
	GARKHCLPRC, GARKHPTPRC	Calculate call prices and put prices for European options on stocks, based on the Garman-Kohlhagen model
	MARGRCLPRC, MARGRPTPRC	Calculate 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 are included in Table 1.2 on page 10.

Using DATA Step Functions within Macro Functions

The macro functions %SYSFUNC and %QSYSFUNC can call most DATA step functions to generate text in the macro facility. %SYSFUNC and %QSYSFUNC have one difference: %QSYSFUNC masks the result of the function and %SYSFUNC does not. %SYSFUNC also works with user-defined functions. For more information about these functions, see %QSYSFUNC and %SYSFUNC in *SAS Macro Language: Reference*.

%SYSFUNC arguments are a single DATA step function and an optional format, as shown in the following examples:

```
%sysfunc(date(),worddate.)
%sysfunc(attrn(&dsid,NOBS))
```

You cannot nest DATA step functions within %SYSFUNC. However, you can nest %SYSFUNC functions that call DATA step functions. For example:

```
%sysfunc(compress(%sysfunc(getoption(sasautos)),
```

```
%str(%)%(%')));
```

All arguments in DATA step functions within %SYSFUNC must be separated by commas. You cannot use argument lists that are preceded by the word OF.

Because %SYSFUNC is a macro function, you do not need to enclose character values in quotation marks as you do in DATA step functions. For example, the arguments to the OPEN function are enclosed in quotation marks when you use the function alone, but the arguments do not require quotation marks when used within %SYSFUNC.

```
dsid=open("sasuser.houses","i");
dsid=open("&mydata","&mode");
%let dsid=%sysfunc(open(sasuser.houses,i));
%let dsid=%sysfunc(open(&mydata,&mode));
```

Invoking CALL Routines and the %SYSCALL Macro Statement

When the %SYSCALL macro statement invokes a CALL routine, the value of each macro variable argument is retrieved and passed unresolved to the CALL routine. Upon completion of the CALL routine, the value for each argument is written back to the respective macro variable. If %SYSCALL encounters an error condition, the execution of the CALL routine terminates without updating the macro variable values and an error message is written to the log.

When %SYSCALL invokes a CALL routine, the argument value is passed unresolved to the CALL routine. The unresolved argument value might have been quoted using macro quoting functions and might contain delta characters. The argument value in its quoted form can cause unpredictable results when character values are compared. Some CALL routines unquote their arguments when they are called by %SYSCALL and return the unquoted values. Other CALL routines do not need to unquote their arguments. Here is a list of CALL routines that unquote their arguments when called by %SYSCALL:

- “CALL COMPCOST Routine” on page 232
- “LEXCOMB Function” on page 1057
- “LEXPERK Function” on page 1063
- “CALL LEXPERM Routine” on page 266
- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXSUBSTR Routine” on page 292
- “CALL SCAN Routine” on page 326
- “CALL SORTC Routine” on page 342
- “CALL STDIZE Routine” on page 345

In comparison, %SYSCALL invokes a CALL routine and returns an unresolved value, which contains delta characters. %SYSFUNC invokes a function and returns a resolved value, which does not contain delta characters. For more information, see "Macro Quoting" in *SAS Macro Language Reference for SAS Viya Workbench*, "%SYSCALL Macro Statement" in *SAS Macro Language Reference for SAS Viya Workbench*, and "%SYSFUNC Macro Function" in *SAS Macro Language Reference for SAS Viya Workbench*.

Using Functions to Manipulate Files

SAS manipulates files in different ways, depending on whether you use functions or statements. If you use functions such as FOPEN, FGET, and FCLOSE, you have more opportunity to examine and manipulate your data than when you use statements such as INFILE, INPUT, and PUT.

When you use external files, the FOPEN function allocates a buffer called the File Data Buffer (FDB) and opens the external file for reading or updating. The FREAD function reads a record from the external file and copies the data into the FDB. The FGET function then moves the data to the DATA step variables. The function returns a value that you can check with statements or other functions in the DATA step to determine how to further process your data. After the records are processed, the FWRITE function writes the contents of the FDB to the external file, and the FCLOSE function closes the file.

When you use SAS data sets, the OPEN function opens the data set. The FETCH and FETCHOBS functions read observations from an open SAS data set into the Data Set Data Vector (DDV). The GETVARC and GETVARN functions then move the data to DATA step variables. The functions return a value that you can check with statements or other functions in the DATA step to determine how you want to further process your data. After the data is processed, the CLOSE function closes the data set.

For a complete listing of functions and CALL routines, see "SAS Functions and CALL Routines by Category" on page 100.For complete descriptions and examples, see the dictionary section of this book.

Using the VAR and V Functions

The VAR and V functions return variable attributes. For example, the VARFMT function returns the format that is assigned to a SAS data set variable. If a format has not been assigned to the variable, VARFMT returns a blank string. The VFORMATD function returns the decimal value of the format that is associated with the specified variable.

Here are common VAR functions:

Table 1.3 Common VAR Functions

VAR Function	Short Description
"VARFMT Function"	Returns the format that is assigned to a SAS data set variable.
"VARINFMT Function"	Returns the informat that is assigned to a SAS data set variable.
"VARLEN Function"	Returns the length of a SAS data set variable.
"VARNAME Function"	Returns the name of a SAS data set variable.
"VARTYPE Function"	Returns the data type of a SAS data set variable.

Here are some tips when using the VAR functions:

- Obtain a data set ID from the OPEN function.
- Pass the data set ID and the variable number to the VAR function.
- The VAR functions are supported with %SYSFUNC or %SYSCALL.
- The VAR functions are supported in the DATA step but not in PROC DS2.
- The VAR functions do not support VARCHAR.
- Use the CLOSE function to end the process. The CLOSE function efficiently ends processing.

Here are common V functions. For example, VFORMAT returns the format that is associated with the specified variable. If a format has not been assigned to the variable, VFORMAT returns BEST12. for numeric variables or $*w*. for character variables, where *w* is the length of the variable.

Table 1.4 Common V Functions

Function	Short Description
"CALL VNEXT Routine"	Returns the name, type, and length of a variable that is used in a DATA step.
"VFORMAT Function"	Returns the format that is associated with the specified variable.
"VFORMATD Function"	Returns the decimal value of the format that is associated with the specified variable.
"VFORMATN Function"	Returns the format name that is associated with the specified variable.

Function	Short Description
"VFORMATW Function"	Returns the format width that is associated with the specified variable.
"VINFORMAT Function"	Returns the informat that is associated with the specified variable.
"VINFORMATD Function"	Returns the decimal value of the informat that is associated with the specified variable.
"VINFORMATN Function"	Returns the informat name that is associated with the specified variable.
"VINFORMATW Function"	Returns the informat width that is associated with the specified variable.

Here are some tips when using the V functions:

- The name of the variable is passed during execution of the DATA step or PROC set.
- CALL VNEXT returns the variables in a DATA step.
- The V functions cannot be used with %SYSFUNC or %SYSCALL.
- The V functions can be used in the DATA step. Some of the V functions can be used in PROC DS2.
- The V functions support VARCHAR.
- VFORMATN, VFORMATW, and VFORMATD return the format name, width, and decimal places in the DATA step but are not in PROC DS2.

This example shows how to return the format name, width, and decimal places using VAR functions with PRX. The code uses these VAR functions: VARFMT, VARLEN, VARNAME, and VARTYPE.

```
%macro FORMAT_INFO( data=_last_ );        1
   %local d j m n p w;
   %local dsid fmt len nvars rx type var;
   %let rx = .;
   %let rc = 0;

   %let dsid = %sysfunc(open(&data,IS));
   %put FORMAT_INFO: Data set &data: open=&dsid
   %if &dsid = 0 %then %do;
      %put FORMAT_INFO: Unable to open data set &data
      %let rc = 1;
      %goto exit;
   %end;

   %let nvars = %sysfunc(attrn( &dsid, NVARS ));
   %put FORMAT_INFO: Data set &data has &nvars variables;
   %if &nvars <= 0 %then %do;
      %put FORMAT_INFO: Data set &data has no variables;
```

```
      %let rc = 2;
      %goto exit;
   %end;

   %let p  = %str(/^ *([\$_a-zA-Z]([_a-zA-Z0-9]*[_a-zA-Z])?)?([0-9]*)\.
([0-9]*) *$/);      2
   %let rx = %sysfunc(prxparse( &p ));
   %put %str(       )rx = &rx
   %if &rx = %str(.) %then %do;
      %let rc = 3;
      %goto exit;
   %end;

   %do j = 1 %to &nvars
   %put &j
      %let  var = %sysfunc(varname( &dsid, &j ));   %put
%str(  )varname = &var
      %let type = %sysfunc(vartype( &dsid, &j ));   %put
%str(  )vartype = &type
      %let  len = %sysfunc(varlen(  &dsid, &j ));   %put
%str(   )varlen = &len
      %let  fmt = %sysfunc(varfmt(  &dsid, &j ));   %put
%str(   )varfmt = &fmt

      %if %bquote(&fmt) ^= %then %do;
         %let m = %sysfunc(prxmatch(&rx,&fmt));
         %put %str(    )match = &m
         %if &m %then %do;
            %let n = %sysfunc(prxposn(&rx,1,&fmt));
            %put %str(   )format = &n
            %let w = %sysfunc(prxposn(&rx,3,&fmt));
            %put %str(    )width = &w
            %let d = %sysfunc(prxposn(&rx,4,&fmt));
            %put %str( )decimals = &d
         %end;
      %end;

   %end;


%exit:

   %if &rx ^= %str(.) %then %do;
      %put FORMAT_INFO: Freeing rx=&rx
      %syscall prxfree( rx );
   %end;

   %if &dsid ^= 0 %then %do;
       %put FORMAT_INFO: Closing data set &data
       %let j = %sysfunc(close(&dsid));
   %end;

   %put FORMAT_INFO: Finished, rc=&rc

%mend FORMAT_INFO;
```

```
data test;
   u = 'horse';      format u $2.;
   v = ' hockey ';   format v $char8.;
   w = '  polo  ';
   x = 123.456;      format x 7.3;
   y = 123456789;    format y comma12.;
   z = 0;
   put (_all_) (/=);
run;

%format_info()
```

The following items correspond to the lines that are numbered in the preceding DATA step.

1 This section obtains data types, formats, and other attribute information for variables in the DATA set.

2 This section uses a Perl regular expression to split a format specifier into name, width, and decimals.

SAS writes the following output to the log:

```
FORMAT_INFO: Data set _last_: open=1
FORMAT_INFO: Data set _last_ has 6 variables
rx = 1
1
  varname = u
  vartype = C
   varlen = 5
   varfmt = $2.
    match = 1
   format = $
    width = 2
decimals =
2
  varname = v
  vartype = C
   varlen = 8
   varfmt = $CHAR8.
    match = 1
   format = $CHAR
    width = 8
decimals =
3
  varname = w
  vartype = C
   varlen = 8
   varfmt =
4
  varname = x
  vartype = N
   varlen = 8
   varfmt = 7.3
    match = 1
   format =
    width = 7
decimals = 3
5
  varname = y
  vartype = N
   varlen = 8
   varfmt = COMMA12.
    match = 1
   format = COMMA
    width = 12
decimals =
6
  varname = z
  vartype = N
   varlen = 8
   varfmt =
FORMAT_INFO: Freeing rx=12
FORMAT_INFO: Closing data set _last_
FORMAT_INFO: Finished, rc=0
```

Function Compatibility Character Sets

Overview

SAS string functions and CALL routines can be categorized by level numbers that are used in internationalization. I18N is the abbreviation for internationalization, and indicates string functions that can be adapted to different languages and locales without program changes.

I18N recognizes the following three levels that identify the character sets that you can use:

- "I18N Level 0" on page 19
- "I18N Level 1" on page 19
- "I18N Level 2" on page 19

For more information about function compatibility, see "Internationalization Compatibility for SAS String Functions" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

I18N Level 0

I18N Level 0 functions are designed for use with Single-Byte Character Sets (SBCS) only.

I18N Level 1

I18N Level 1 functions should be avoided, if possible, if you are using a non-English language. The I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings under certain circumstances.

I18N Level 2

I18N Level 2 functions are designed for use with SBCS, DBCS, and MBCS (UTF8).

Using Random-Number Functions and CALL Routines in the DATA Step

Overview

Random-number generator (RNG) refers to an algorithm that generates pseudorandom numbers or a hardware-based random-number generator. The following sections describe random-number generators (RNGs) in SAS and how to use the RAND function to generate random numbers from probability distributions. Information about how to initialize an RNG and how to generate independent streams of random numbers is also included.

Concepts

Here are the main concepts:

- SAS supports three families of pseudorandom number generators: one Mersenne twister generator, which is a permuted congruential generator (PCG), and two ThreeFry generators.
- SAS supports a hardware-based RNG on certain chip sets.
- You can use the STREAMINIT subroutine to specify an RNG and a seed value to initialize the RNG. If you do not call the STREAMINIT subroutine, the first call to the RAND function causes SAS to generate a seed value and to initialize the default RNG.
- For a modern pseudorandom number generator, there is no reason to prefer one seed value over another. Two different seed values generate equally random streams.
- After an RNG is initialized, it remains in use until the end of the DATA step or procedure. You cannot switch RNGs in the middle of a DATA step.
- The RAND function generates random values from probability distributions by transforming one or more uniform variates from the current RNG.
- If multiple DATA steps are wrapped in a macro, you want each DATA step to generate an independent sequence of random numbers. The STREAM subroutine initializes an RNG based on a seed and a second value called a *key*. For the PCG and ThreeFry generators, every seed-key combination results in an independent stream of random values.

- The PCG RNG provides a good combination of statistical quality, speed, cycle length, and multiple independent streams. For a summary of strengths and weaknesses for the RNG choices in SAS, see "Summary of RNG Attributes" on page 31 .

Types of Random-Number Functions

Two types of random-number functions are available in SAS.

The recommended random-number function is the RAND function. The RAND function works with the STREAMINIT and STREAM subroutines. The STREAMINIT subroutine uses an initial-seed value to initialize a random-number generator. Subsequent calls to the RAND function generate a stream of pseudorandom variates from a specified probability distribution by using the specified RNG.

The legacy random-number functions (NORMAL, UNIFORM, RANBIN, RANCAU, RANEXP, RANGAM, RANNOR, RANPOI, RANTBL, RANTRI, and RANUNI) are still supported in the DATA step, but are not recommended for serious statistical analyses. Because the underlying algorithm is not appropriate for parallel computations, the legacy functions are not supported in newer SAS procedures that are designed to execute in parallel.

Pseudorandom Numbers

Except when using a hardware-based method, a random-number generator produces pseudorandom numbers from mathematical formulas. Pseudorandom numbers satisfy many of the statistical properties of random variables. Consequently, the sequence of numbers appears to be random even though the sequence is actually deterministic.

The STREAMINIT subroutine uses a positive seed to initialize an RNG. Subsequent calls do nothing. For each thread, the first call to the STREAMINIT subroutine initializes the RNG on a per-thread basis. Subsequent calls to the STREAMINIT function from the same thread are ignored. The RNG remains in effect for the remainder of the DATA step or procedure.

After an RNG is initialized, each call to the RAND function modifies the state of the RNG. The next call to the RAND function returns a new value and again modifies the RNG state. A *stream* is a sequence of random numbers from an RNG. In most situations, all random numbers in the same DATA step are produced by a single stream.

Under certain circumstances, you might need to generate multiple independent streams from the same RNG. You can use the STREAM subroutine to specify a key value. For the PCG and ThreeFry generators, every seed-key combination results in an independent stream of random values. The STREAM subroutine enables you to access multiple copies of an RNG. Every copy is initialized separately.

Types of Random-Number Generators

You can specify a random-number generator by using the STREAMINIT subroutine.

- Mersenne twister pseudorandom number generator (Matsumoto and Nishimura 1998, 2002, 2005). SAS supports four methods of the Mersenne twister algorithm:
  - The MT1998 method is the 1998 32-bit Mersenne twister algorithm. This method was the default RNG for the RAND function prior to SAS® 9.4M3. A small number of seed values for the MT1998 method do not produce a stream of numbers that is sufficiently random. Consequently, it is better to use one of the other Mersenne twister methods.
  - The MTHYBRID method is a hybrid method that incorporates some of the 2002 algorithm into the MT1998 method. This method is the default RNG, beginning with SAS® 9.4M3.
  - The MT32 method (also called MT2002) is the 2002 32-bit Mersenne twister algorithm.
  - The MT64 method is a 64-bit Mersenne twister algorithm.
- A permuted congruential generator (O'Neill 2015). You can choose the PCG (also called PCG64i) RNG to use the permuted congruential generator method.
- A counter-based RNG. The counter-based RNG in SAS is based on the Threefish encryption function in the Random123 library (Salmon et al. 2011). You can choose from these methods:
  - The TF2 method (also called THREEFRY2x64) selects a 2x64-bit counter-based RNG.
  - The TF4 method (also called THREEFRY4x64) selects a 4x64-bit counter-based RNG.
- A hardware-based RNG. The hardware-based RNG generates random values from thermal noise in the chip. Currently, the only supported hardware-based RNG requires an Intel processor that supports the RdRand instruction. You can choose from these methods:
  - The HARDWARE method selects the default generator on the current CPU.
  - The RDRAND method, which requires an Intel processor that supports the RdRand instruction (Ivy Bridge and later processors).

Here is a summary of RNG choices:

RNG	Description
MTHybrid	Hybrid 1998/2002 32-bit Mersenne twister (default method).
MT1998	1998 32-bit Mersenne twister. Deprecated.

RNG	Description
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 Threefish encryption function in the Random123 library.
HARDWARE	Any supported hardware-based random-number generator.
RDRAND	Intel hardware-based RdRand instructions.

If you do not call the STREAMINIT routine or do not supply a method, the default RNG is the MTHYBRID method.

The first call to STREAMINIT sets the RNG and seed for each DATA step. Subsequent calls to STREAMINIT on the same thread are ignored. For more information, see "CALL STREAMINIT Routine" on page 352. Hardware-based RNGs do not use seeds. For more information, see "Hardware-Based RNGs" on page 30 .

Each RNG has advantages and disadvantages. The various RNGs are compared in the section "Summary of RNG Attributes" on page 31. Based on extensive testing, the section concludes that the PCG method is a fast algorithm that provides a good combination of desirable properties.

RNGs, Seeds, and Random Numbers from Distributions

This section explains how to use the STREAMINIT subroutine and the RAND function to generate random values from two probability distributions: the uniform distribution and the normal distribution. The example contains four DATA steps.

- In the first DATA step, the STREAMINIT call specifies the seed 12345, but does not specify an RNG. Consequently, the default RNG, which is MTHYBRID, is used by subsequent calls to the RAND function. The RANDMT data set contains two variables. The U1 variable is a random sample from the uniform distribution. The N1 variable is a random sample from the standard normal distribution.

- The second DATA step specifies the PCG RNG and the same seed 12345. Although the U2 and N2 variables are random samples from the uniform and standard normal distributions, the values in the RANDPCG data set are different from the values in the RANDMT data set because the numbers were generated by different RNGs.
- The third DATA step calls the PCG RNG again, but uses a different seed 54321. Again, the U3 and N3 variables are random samples from the uniform and standard normal distributions. The values in the RANDPCG2 data set are different from the values in the RANDMT data set because the values were generated by different RNGs. The values in the RANDPCG2 data set are different from the values in the RANDPCG data set because they were generated by a different seed.
- In the fourth DATA step, the RNG stream is identical to the stream in the third DATA step, but the RAND function is called in a different order. The calls to RAND('NORMAL') use an indeterminate number of values from the stream. Therefore, the RANDPCG2 and RANDPCG3 data sets are different for most observations.

```
data RandMT;
call streaminit(12345);        /* use default method */
do i = 1 to 8;
   u1 = rand('uniform');
   n1 = rand('normal');
   output;
end;
run;

data RandPCG;
call streaminit('PCG', 12345); /* PCG method, same seed */
do i = 1 to 8;
   u2 = rand('uniform');
   n2 = rand('normal');
   output;
end;
run;

data RandPCG2;
call streaminit('PCG', 54321); /* PCG method, different seed */
do i = 1 to 8;
   u3 = rand('uniform');
   n3 = rand('normal');
   output;
end;
run;
data RandPCG3;
call streaminit('PCG', 54321); /* same stream for PCG method */
do i = 1 to 8;
   n4 = rand('normal');        /* different order */
   u4 = rand('uniform');
   output;
end;
run;

data Rand(drop=i);
merge RandMT RandPCG RandPCG2 RandPCG3;
```

```
run;

proc print data=Rand; var u1-u4 n1-n4; run;
```

The output shows that random numbers depend on the RNG, the seed value, the distribution that is specified in the RAND function, and the order in which the RAND function is called.

Obs	u1	u2	u3	u4	n1	n2	n3	n4
1	0.58330	0.82524	0.16685	0.13491	0.27605	-0.68327	0.86974	-0.15737
2	0.28057	0.99923	0.82576	0.88551	0.25073	-0.75390	-0.28423	0.17183
3	0.64740	0.09084	0.96562	0.96562	-2.13902	-2.02323	-0.44223	-0.43210
4	0.87578	0.03156	0.94708	0.13485	-0.92463	0.42060	0.34340	1.11729
5	0.51838	0.72888	0.02538	0.83922	0.75687	-0.96208	1.47808	0.06267
6	0.93354	0.33863	0.94750	0.85702	-0.48949	1.58769	0.22598	0.64905
7	0.18739	0.24491	0.83922	0.74307	-0.26104	1.67247	0.79432	0.76911
8	0.74119	0.15313	0.76464	0.51714	-0.79348	-1.06984	0.43582	1.03904

An RNG Generates Uniformly Distributed Values

An RNG generates random values from the uniform distribution. The RAND function transforms one or more uniform variates from the RNG to construct a value from a probability distribution. Each time the RAND function uses a value from the stream, the function updates the internal state of the RNG.

For some distributions, such as the Bernoulli and exponential distributions, the RAND function uses one value from the RNG stream to generate its output. For other distributions, the RAND function might use two or more uniform variates. Some distributions (such as the normal and hypergeometric) require an indeterminate number of variates. Consequently, one call to the RAND function might use many uniform values from the RNG.

Seed Values and Reproducible Streams

It is important to obtain the same pseudorandom streams each time you run a program. This ability is useful when you are debugging a program or creating demonstrations. If you are conducting scientific research, obtaining the same streams enables other scientists to reproduce your results.

To create reproducible streams of pseudorandom numbers, call the STREAMINIT routine with a positive seed value before calling the RAND function, as explained in "RNGs, Seeds, and Random Numbers from Distributions" on page 23. To reproduce the same streams of pseudorandom numbers, use the same seed value and key values each time you run the DATA step, and specify the calls to the RAND

function in the same order. Key values are discussed in the section "Multiple Independent Streams" on page 28.

Hardware-based methods never produce reproducible streams.

You can specify a seed or let the software choose the seed. The following seed values are valid:

- For the MTHYBRID, MT32, and MT1998 methods, the seed value can be a positive integer less than $2^{32}-1$ (4,294,967,295).
- For the MT64, PCG, TF2, and TF4 methods, the seed value can be a positive integer less than $2^{64}-1025$ (18,446,744,073,709,549,568). Be aware that some very large integers do not have an exact representation in double-precision floating-point numbers. In SAS, all integers less than or equal to K have an exact representation, where K=CONSTANT('EXACTINT'). On most CPUs, $K=2^{53}$.

If you specify a positive seed, you can replicate a stream of pseudorandom numbers by rerunning the DATA step.

Using a random number function in a WHERE statement might generate a different result set from that in a subsetting IF statement. This different result set can be caused by how the criteria are optimized internally by SAS and is the expected behavior.

Initial Seed Values

You can specify an initial seed value by calling the STREAMINIT subroutine with a positive integer before calling the RAND function.

For a non-hardware RNG, SAS automatically generates an initial seed value if you make any of these calls:

- Call STREAMINIT with a missing, zero, or negative argument.
- Call STREAMINIT without specifying a seed.
- Call RAND before you call STREAMINIT.

SAS generates the seed value in one of two ways:

- From a hardware-based method, if one is available.
- From the current date, time, and possibly other factors. In this case, the same seed value is not generated again for at least several months.

The seed value that initializes the RNG is stored in the SAS macro variable, &SYSRANDOM. To display the seed value in the SAS log, you can run the `%put &sysrandom;` statement.

Are Some Seed Values Better Than Others?

You can initialize a pseudorandom-number generator by calling the STREAMINIT subroutine with a positive integer. Seeds do not generate streams that are more random than other seeds. For example, the stream for the seed 12345 is not less random than the seed for the nine-digit prime number 937162211. For a modern pseudorandom-number generator, streams from different seeds have similar statistical properties. Some programmers use a birthdate or a phone number as a seed. Other programmers use a numeric sequence such as 9876. Still others prefer to enter numbers at random. All of these methods are valid, and there is no intrinsic reason to prefer one seed over another.

If you do not want to choose a seed yourself, you can let SAS choose the seed:

1. Run a DATA step that calls the STREAMINIT subroutine with the value 0. This call generates a seed and stores it in the SYSRANDOM system macro variable.
2. Use the %PUT statement to display the value of the SYSRANDOM macro variable in the SAS log.
3. Copy the value from the SAS log into the SAS program that simulates data.

If you use this technique often, you can wrap the first two steps in a SAS macro:

```
%macro GenerateSeed();
   data _NULL_;
   call streaminit(0);
   run;
   %put &=SYSRANDOM;
%mend;

%GenerateSeed;   /* displays a value such as 441655494 */

data Simulate;
call streaminit(441655494); /* use the generated value */
do i = 1 to 5;
   x = rand("Uniform");
   output;
end;
run;
```

Cycles

You can call the RAND function multiple times within a single DATA step. For the non-hardware RNGs, the result is a sequence of pseudorandom numbers. Theoretically, if you call the RAND function enough times, the sequence of values returned by RAND eventually repeats. The number of values returned by RAND(‘uniform’) before the cycle repeats is called the cycle length or period of the RNG.

All of the pseudorandom methods supported by the RAND function provide very long cycle lengths and do not repeat in practice. The shortest cycle length of any supported method is approximately 2^{64}. Using a typical contemporary computer with eight cores, it would take about 1,000 years to generate that many pseudorandom numbers.

Overlap

Two DATA steps that use the same RNG but different seed values produce different sequences of pseudorandom numbers. *Overlap* occurs when a second DATA step generates k initial values and then generates the same sequence that appeared in the first DATA step.

If you are using a Mersenne twister or a PCG generator, it is theoretically possible (but not probable) that the two streams will overlap. If an overlap exists, the two streams in the two DATA steps are not statistically independent, so it is important to estimate the probability of an overlap.

Assume that a DATA step uses a Mersenne twister or a PCG generator to compute a stream of L uniform pseudorandom numbers. Suppose you run the DATA step R times, each time with a different randomly chosen initial seed. Let P be the period of the generator. The probability that any two streams will overlap is approximately R^2 L/P (Vigna 2015).

For example, the RNG that has the shortest period is the PCG method, which has a period of 2^{64}. If a DATA step generates a billion values ($L=10^9$) and you randomly choose 100 different seeds, the probability of overlap for the PCG method is approximately $(100^2)(10^9) / 2^{64}$ = 5.42e-7, or less than one in a million. For the Mersenne twister family of RNG, which has a period of 2^{19937}, the probability of an overlap is extremely small.

If you use the Threefry RNG, the probability of overlap is 0. Two streams that are initialized with different seeds or different keys never overlap.

Multiple Independent Streams

For most applications, you can use a single stream to generate random numbers. However, it is theoretically possible that two different DATA steps could produce streams that overlap. For this and other reasons, SAS supports the STREAM subroutine in the DATA step.

You can generate multiple streams of pseudorandom numbers within a single DATA step by calling the STREAM routine to specify a key for each stream. A *key* is a nonnegative integer value. Within a DATA step, you can specify multiple keys, thereby generating multiple streams.

The STREAM subroutine enables you to create and access multiple copies of an RNG, each of which was initialized separately. After you call the STREAM

subroutine, subsequent calls to the RAND function use that stream until a later call to the STREAM subroutine changes the key.

Different streams can be considered statistically independent for most purposes, especially if you are using the PCG or Threefry methods. Different streams are computationally independent, which means that generating pseudorandom numbers from one stream have no effect on pseudorandom numbers in another stream.

Calling the STREAM subroutine with a key value of zero yields the same stream that would be obtained if STREAM were not called.

The effect of CALL STREAM depends on the RNG method:

- For a hardware-based method, CALL STREAM has no effect. A warning message in the log alerts you that the call was ignored.
- For a Mersenne twister RNG, if you do not call STREAM, the stream is initialized by using only the seed value from STREAMINIT. If you call STREAM with a positive key value, the stream is initialized using both the seed value from STREAMINIT and the key value from STREAM. It is extremely unlikely that two Mersenne twister streams with different key values would overlap or be statistically dependent.
- For a permuted congruential generator (PCG), each distinct key value defines a distinct stream. Two streams with different key values never overlap. They are statistically independent for most purposes. The internal computations of the STREAM subroutine use the key value to compute an increment for the linear congruential generator used inside the PCG.
- For a Threefry generator, each distinct key value defines a distinct stream. Two streams with different key values never overlap. They are statistically independent for most purposes. In two DATA steps with different seed values, none of the streams in one DATA step overlap with any stream in the other DATA step, even if the key values are the same. All streams in the two DATA steps are statistically independent for most purposes. The internal computations of the STREAM subroutine use the key value and the seed value to compute a cryptographic-like key used inside the Threefry generator.

For most applications, you can use a single stream and never call the STREAM subroutine. One application that is helpful to have for multiple streams is a macro that calls multiple DATA steps, each of which generates random numbers. (Equivalently, the macro contains a macro loop that calls a DATA step multiple times.) Although you can easily pass a seed value into the macro, that action alone does not enable you to generate independent streams. However, you can use the STREAM subroutine to guarantee independent streams. For example, you can use the iterator in the macro loop as the key to ensure independent streams, as shown in this example:

```
%macro RepeatRand(N=, Repl=, seed=);
%do k = 1 %to &Repl;
   data U&k;
      call streaminit('PCG', &seed);
      call stream(&k);
      do i = 1 to &N;
         u = rand("uniform");
         output;
```

```
      end;
   run;
%end;
%mend;

%RepeatRand(N=8, Repl=2, seed=36457);
```

Because the STREAM subroutine uses the iteration number as a key value, each DATA step generates an independent stream of random numbers.

Another application of the STREAM subroutine occurs when you are distributing a program to run in parallel on multiple nodes in a computational grid. For example, if you want to generate 400,000 observations and you have access to a grid that has four nodes, you can generate 100,000 observations on each node. If your program uses the node ID as the key value in the STREAM call, then the random number streams that are generated on each node are independent.

Hardware-Based RNGs

If you use the RDRAND method, the RAND function uses an Intel RdRand hardware-based generator if your computer supports it. RdRand is designed to generate numbers by using a pseudorandom-number generator that is occasionally reseeded by a hardware device that is calibrated by a complicated feedback loop.

The HARDWARE method causes the RAND function to use the default hardware-based generator that is available on your computer. Currently, the Intel RdRand generator is the only hardware-based method that is supported by SAS. However, SAS might support additional generators on future chips other than Intel chips. If you use the HARDWARE method, your program is potentially more portable. If a SAS program uses a hardware-based method, you will get different random numbers every time you run the program. That is, random numbers from the hardware methods are not reproducible.

Hardware-based methods do not use initial seed values. Also, hardware streams never cycle or overlap. Hardware generators do not support multiple streams.

Statistical Quality of RNGs

An important consideration of pseudo-RNGs is how well the pseudorandom stream mimics statistical properties of truly random processes. The most thorough collection of numerical tests of randomness is called TESTU01 (L'Ecuyer and Simard 2007), which contains three suites of tests called Small Crush, Crush, and Big Crush. A generator that passes all three Crush suites when called with a wide variety of seeds is *Crush-resistant*. A stream that is generated by a Crush-resistant RNG has excellent statistical properties.

Mersenne twister generators are not Crush-resistant because they systematically fail linear complexity tests (Vigna 2015). However, linear complexity tests have less

practical importance than many other tests that pertain directly to the randomness assumptions in most statistical models.

If you do not require the extremely long period of the Mersenne twister, the 64-bit permuted congruential generator (PCG) is fast and is Crush-resistant (O'Neill 2015).

The Threefry generators in Random123 (Salmon, Moraes, Dror, and Shaw 2011) are also Crush-resistant. Threefry2x64 and Threefry4x64 are somewhat slower than the PCG method, but have longer periods.

Statistical Quality of Mersenne Twister RNG with Certain Seeds

Prior to SAS® 9.4M3, the default algorithm for RAND was the MT1998 method, which is the 1998 32-bit Mersenne twister RNG by Matsumoto and Nishimura (1998). However, the MT1998 initialization algorithm performs poorly when the seed is exactly divisible by 8192 and a positive key is not specified in the CALL STREAM routine. More generally, MT1998 can get stuck for a long time in internal states that have a high proportion of 0-bits (Panneton, L'Ecuyer, and Matsumoto 2006).

Matsumoto and Nishimura (2002) provided an improved initialization algorithm that is used in MT32. The MT1998 and MT32 methods use the same algorithm for generating pseudorandom numbers after the initialization is performed. Theoretically, if you run the MT32 method long enough, you can encounter the same low-quality substreams that are sometimes produced by the MT1998 initialization algorithm, but the chance of this happening is negligible.

In SAS® 9.4M3, the default method for RAND was changed to MTHybrid, which uses MT32 when the seed is divisible by 8192. Otherwise, MTHybrid uses MT1998. Therefore, MTHybrid yields the same results as MT1998 for the vast majority of cases, but does not generate low-quality streams when the seed is divisible by 8192.

The 64-bit Mersenne Twister (Matsumoto 2005) provides higher precision than the 32-bit versions and possibly better randomness.

Summary of RNG Attributes

When choosing a random-number generator, there are several factors that might influence your decision. Choosing one RNG over another involves trade-offs between speed, cycle length, statistical quality, and whether the RNG supports independent streams. The following table summarizes the attributes of the RNGs that are supported by the RAND function.

In this table, the entries in the Relative Performance column are based on 10 runs of the DATA step, each generating one billion uniform variates. The column shows

that the 32-bit Mersenne twister RNG is relatively fast, followed by the PCG. The hardware-based RDRAND method is the slowest.

The Quality column indicates the statistical properties of the RNG. The RNGs that are Crush-resistant are marked Excellent, as is the hardware-based method. The MT1998 method is judged Fair because the initialization algorithm performs poorly when the seed is exactly divisible by 8192. The MT32 and MT64 methods do not have that problem, but are not Crush-resistant.

The table indicates that the PCG method provides a good combination of statistical quality, speed, cycle length, and multiple independent streams.

Method	Period	Relative Performance		Quality	Independent Streams
		Linux	Windows		
MTHYBRID	2^{19937}	1.0	1.0	Good	No
MT1998	2^{19937}	1.0	1.0	Fair	No
MT32	2^{19937}	1.0	1.0	Very Good	No
MT64	2^{19937}	1.3	1.3	Very Good	No
PCG	2^{64}	1.1	1.2	Excellent	2^{63}
TF2	2^{129}	1.5	1.5	Excellent	2^{128}
TF4	2^{258}	1.8	1.6	Excellent	2^{256}
RDRAND	Infinite	9.6	4.0	Excellent	Not Applicable

Using SYSRANDOM and SYSRANEND Macro Variables to Produce Random Number Streams

Overview of the SYSRANDOM and SYSRANEND Macro Variables

Many SAS procedures (for example, FREQ, GLM, MCMC, OPTEX, and PLAN) use random number streams. These procedures use the same random number functions and CALL routines that you use in SAS DATA steps. SAS procedures use a SEED= option to provide the seed that initializes the random number stream.

SAS procedures with random number seeds create two macro variables, SYSRANDOM and SYSRANEND. You can use these macro variables to produce reproducible random number streams across procedures.

The SYSRANDOM Macro Variable

The SYSRANDOM macro variable stores the random number seed from the most recent procedure. The macro variable corresponds to the integer that is specified in the SEED= option. Many procedures, such as FREQ, GLM, MI, MCMC, OPTEX, PHREG, and PLAN, have a SEED= option. The SEED= option specifies the integer that is used to start the random number stream. Positive seed specifications are used as specified. If a seed is not specified in the SEED= option, or if the seed is less than or equal to zero, the procedure generates a seed from the clock time. You can use the SYSRANDOM macro variable to recover both directly specified and internally generated seeds.

The SYSRANEND Macro Variable

The SYSRANEND macro variable stores a seed that can be used to start the next step in your program. In some cases, this seed captures the state of the random number process when a procedure is completed. Your program can contain multiple steps and can control the random number sequence without specifying an explicit seed for each procedure. You can start the simulation with one seed, and use the

SYSRANEND macro variable to provide the seed in all subsequent procedures. In some cases, you can also use the SYSRANEND macro variable to stop and restart the random number generators (RNGs) that are continuing the same stream.

There are two types of RNGs in SAS procedures. The older RNG, which is used by the RANUNI function, starts the pseudo-random number stream with a single seed, and the state of the process can be captured in a new seed. The GLM, GLIMMIX, MI, OPTEX, PLAN, and other procedures use this older RNG. When the procedure exits, the value that is stored in SYSRANEND is the new seed. You can stop and restart the generator from its stopping point by using the SYSRANEND macro variable.

Other procedures, such as MCMC, GENMOD, LIFEREG, and PHREG, use the newer Mersenne-Twister RNG. This RNG is also used in the RAND function and does not propagate the state of the stream through a single seed. Some procedures use one RNG for some computations and the other RNG for other computations. You can use the SYSRANEND macro variable from these procedures to make a sequence of procedure runs reproducible, but the random streams is not equal to a single, long, procedure run.

Example: Reproducing Results

The following example shows how to recover the seed and use it to reproduce a set of results. The MCMC procedure generates samples from a posterior distribution. The following statements produce posterior samples from a linear regression model:

```
title 'Bayesian Linear Regression';

proc mcmc data=sashelp.class seed=0 outpost=out1;
   parms beta0 0 beta1 0;
   prior beta0 beta1 ~ normal(mean=0, var=1e6);
   mu=beta0 + beta1*height;
   model weight ~ n(mu, var=137);
run;
```

Because SEED=0 was specified, a random number seed is automatically generated from the clock time. This seed is stored in the SYSRANDOM macro variable. You can use a %PUT statement to display its value:

```
%put sysrandom=&sysrandom;
```

The following step creates the same results as the previous step by using the same seed:

```
proc mcmc data=sashelp.class seed=&sysrandom outpost=out2;
   parms beta0 0 beta1 0;
   prior beta0 beta1 ~ normal(mean=0, var=1e6);
   mu=beta0 + beta1*height;
   model weight ~ n(mu, var=137);
run;
```

Submit the following step to see that the two PROC MCMC executions produce identical samples:

```
proc compare data=out1 compare=out2;
```

```
run;
```

Example: Creating a Reproducible Random Number Stream

The PLAN procedure constructs and randomizes full factorial experimental designs. The OPTEX procedure searches a set of candidate design points for an optimal experimental design. Both procedures have a SEED= option. You can use the SYSRANEND macro variable to make a sequence of steps reproducible, as shown in the following example:

```
proc plan seed=17;
   factors x1=4 x2=4 x3=2 x4=2 x5=3 x6=3 x7=2 x8=2 / noprint;
   output out=cand;
run;
quit;

%put sysranend=&sysranend;

proc optex data=cand seed=&sysranend;
   class x1-x8;
   model x1-x8;
   generate n=26 iter=10 method=m_federov;
   output out=des;
run;
quit;
```

You can call PROC OPTEX multiple times and stop when a design with an efficiency (the measure of how good the design is) greater than 98% is found. You can use the SYSRANDOM and SYSRANEND macro variables to do this. The following statements call PROC OPTEX to create 100 designs and generate the best *D*-efficiency:

```
proc plan seed=17;
   factors x1=4 x2=4 x3=2 x4=2 x5=3 x6=3 x7=2 x8=2 / noprint;
   output out=cand;
run;
quit;

%macro design;
   ODS EXCLUDE ALL;
   %do %until(%sysevalf(&eff > 98));
      proc optex data=Cand seed=&sysranend;
         class x1-x8;
         model x1-x8;
         generate n=26 iter=100 keep=1 method=m_federov;
         ods output efficiencies=e1;
      run;
      quit;

      data _null_;
         set e1;
         call symputx('eff', dcriterion, 'L');
```

```
      run;
      %end;
   ODS EXCLUDE NONE;

   proc optex data=Cand seed=&sysrandom;
      class x1-x8;
      model x1-x8;
      generate n=26 iter=100 keep=1 method=m_federov;
      output out=des;
   run;
   quit;
%mend;

%design;
```

The *D*-efficiency is stored in a macro variable, and iteration stops when *D*-efficiency is greater than 98. The seed from the last step is used to reproduce and display the final results.

Hashing Functions and Hash-Based Message Authentication Code

Hashing Functions

Hashing is the transformation of data (known as a *message*) into a short fixed-length value (known as a *digest*) that represents the original string. Hashing is used to index and retrieve items in a database because it is faster to find the item by using the hashed key than by using the original value. Hashing functions are useful when you are developing shell scripts for software installation, file comparison, and detection of file corruption and tampering. You can use several types of algorithms in the hashing process. SAS supports these algorithms:

- MD5
- SHA1
- SHA256
- SHA384
- SHA512
- CRC32

SAS supports these hashing functions:

HASH_FAST_HEX
: returns a message digest as a 16-byte hexadecimal string for a key value and a message consisting of one or more character strings or numbers, using the fast-hash algorithm.

HASH_XX_HEX
: returns a message digest as a 16-byte hexadecimal string for a key value and a message consisting of one or more character strings or numbers.

HASHING
: transforms a message into a digest in hexadecimal representation.

HASHING_FILE
: transforms the entire contents of a file into a digest and returns the digest in hexadecimal representation.

HASHING_HMAC
: transforms a message into a digest in hexadecimal representation by using the HMAC algorithm, which uses the specified algorithm and a key value.

HASHING_HMAC_FILE
: transforms the entire contents of a file into a digest by using the HMAC algorithm, which uses the specified algorithm and a provided key value.

HASHING_HMAC_INIT
: initializes a running HMAC hash.

HASHING_INIT
: initializes a running hash.

HASHING_PART
: provides a portion of the message for the running hash.

HASHING_TERM
: returns the final digest in hexadecimal representation for the running hash.

MD5
: returns the MD5 digest for a specified message string.

SHA256
: returns the SHA256 digest for a specified message string.

SHA256HEX
: returns the SHA256 digest for a specified message, and the digest is provided in hexadecimal representation.

SHA256HMACHEX
: returns the result of the message digest (digital signature) of a specified string by using the HMAC algorithm.

This example transforms a string of characters into a shorter fixed-length value that represents the original string by using the HASHING function. The MD5, SHA1, and SHA256 algorithms are used to generate the values that represent the character string.

```
data null;
   message = "The quick brown fox jumps over the lazy dog";
   md5 = hashing('md5',message);
   sha1 = hashing('sha1',message);
   sha256 = hashing('sha256',message);
   put md5= / sha1= / sha256=;
```

```
run;
```

```
md5=9E107D9D372BB6826BD81D3542A419D6
sha1=2FD4E1C67A2D28FCED849EE1BB76E7391B93EB12
sha256=D7A8FBB307D7809469CA9ABCB0082E4F8D5651E46D3CDB762D02D0BF37C9E592
```

MD5

The MD5 function returns the MD5 digest for a specified message. The MD5 function converts a string, based on the MD5 algorithm, to a 128-bit hash value. This hash value is referred to as a message digest (digital signature), which is unique for each string that is passed to the function string.

The MD5 function does not format its own output. Use the $BINARY*w*. or $HEX*w*. formats to view readable results.

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 unique result.

In a DATA step, if the MD5 function returns a value to a variable that has not previously been assigned a length, that variable is assigned a length of 200 bytes.

You can use the MD5 function to track changes in your data sets. The MD5 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 MD5 function can be useful when you are developing shell scripts for software installation, file comparison, and 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.

SHA256

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.

Hash-Based Message Authentication Code (HMAC)

HMAC is a message authentication code obtained by running a cryptographic hash function such as MD5, SHA1, and SHA256 over data that you want to authenticate and create a shared secret key. HMAC processing is performed with a hashing algorithm. This dual processing strengthens the integrity and authentication of the message string.

These are the HASHING functions that have HMAC support:

- “HASHING_HMAC Function” on page 898
- “HASHING_HMAC_FILE Function” on page 900
- “HASHING_HMAC_INIT Function” on page 901
- “SHA256HMACHEX Function” on page 1387

Here is an example of Hashing functionality:

```
data abc;
   input x $ y $ z $;
datalines;
abc . .
ab c .
a bc .
a b c
;

data digests;
   set abc;
   drop handle ignore;
   fast = hash_fast_hex( 12345, trimn(x), trimn(y), trimn(z) );
   handle = hashing_init( 'md5' );
   ignore = hashing_part( handle, trimn(x) );
   ignore = hashing_part( handle, trimn(y) );
   ignore = hashing_part( handle, trimn(z));
   md5 = hashing_term( handle );
   handle = hashing_hmac_init( 'md5', '12345' );
   ignore = hashing_part( handle, trimn(x) );
```

```
   ignore = hashing_part( handle, trimn(y) );
   ignore = hashing_part( handle, trimn(z));
   hmac = hashing_term( handle );
run;

proc print data=digests;
run;
```

Date and Time Intervals

Definition of a Date and Time Interval

An *interval* is a unit of measurement that SAS counts within an elapsed period of time, such as days, months or hours. SAS determines date and time intervals based on fixed points on the calendar or clock. The starting point of an interval calculation defaults to the beginning of the period in which the beginning value falls, which might not be the actual beginning value that is specified. For example, if you are using the INTCK function to count the months between two dates, regardless of the actual day of the month that is specified by the date in the beginning value, SAS treats the beginning value as the first day of that month.

Interval Names and SAS Dates

Specific interval names are used with SAS date values, and other interval names are used with SAS time and datetime values. The interval names that are used with SAS date values are YEAR, SEMIYEAR, QTR, MONTH, SEMIMONTH, TENDAY, WEEK, WEEKDAY, and DAY. The interval names that are used with SAS time and datetime values are HOUR, MINUTE, and SECOND.

Interval names that are used with SAS date values can be prefixed with 'DT' to construct interval names for use with SAS datetime values. The interval names DTYEAR, DTSEMIYEAR, DTQTR, DTMONTH, DTSEMIMONTH, DTTENDAY, DTWEEK, DTWEEKDAY, and DTDAY are used with SAS time or datetime values.

Incrementing Dates and Times By Using Multipliers and By Shifting Intervals

SAS provides date, time, and datetime intervals for counting different periods of elapsed time. By using multipliers and shift indexes, you can create multiples of

intervals and shift their starting point to construct more complex interval specifications.

The general form of an interval name is

name<multiplier><.shift-index>

Both the argument *multiplier* and the *shift–index* argument are optional and default to 1. For example, YEAR, YEAR1, YEAR.1, and YEAR1.1 are all equivalent ways of specifying ordinary calendar years that begin in January. If you specify other values for *multiplier* and for *shift-index*, you can create multiple intervals that begin in different parts of the year. For example, the interval WEEK6.11 specifies six-week intervals starting on second Wednesdays.

Commonly Used Time Intervals

Time intervals that do not nest within years or days are aligned relative to the SAS date or datetime value 0. SAS uses the arbitrary reference time of midnight on January 1, 1960, as the origin for non-shifted intervals. Shifted intervals are defined relative to January 1, 1960.

For example, MONTH13 defines the intervals January 1, 1960, February 1, 1961, March 1, 1962, and so on, and the intervals December 1, 1958, November 1, 1957, and so on, before the base date January 1, 1960.

As another example, the interval specification WEEK6.13 defines six-week periods starting on second Fridays. The convention of alignment relative to the period that contains January 1, 1960, determines where to start counting to determine which dates correspond to the second Fridays of six-week intervals.

The following table lists time intervals that are commonly used.

Table 1.5 *Commonly Used Intervals with Optional Multiplier and Shift Indexes*

Interval	Description
DAY3	Three-day intervals
WEEK	Weekly intervals starting on Sundays
WEEK.7	Weekly intervals starting on Saturdays
WEEK6.13	Six-week intervals starting on second Fridays
WEEK2	Biweekly intervals starting on first Sundays
WEEK1.1	Same as WEEK
WEEK.2	Weekly intervals starting on Mondays
WEEK6.3	Six-week intervals starting on first Tuesdays

Interval	Description
WEEK6.11	Six-week intervals starting on second Wednesdays
WEEK4	Four-week intervals starting on first Sundays
WEEKDAY	Five-day work week with a Saturday-Sunday weekend
WEEKDAY1W	Six-day week with Sunday as a weekend day
WEEKDAY35W	Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)
WEEKDAY17W	Same as WEEKDAY
WEEKDAY67W	Five-day week with Friday and Saturday as weekend days
WEEKDAY3.2	Three-weekday intervals with Saturday and Sunday as weekend days (The intervals are aligned with respect to Jan. 1, 1960. For intervals that nest within a year, it is not necessary to go back to Jan. 1, 1960 to determine the alignment.)
TENDAY4.2	Four ten-day periods starting at the second TENDAY period
SEMIMONTH2.2	Intervals from the 16th of one month through the 15th of the next month
MONTH2.2	February–March, April–May, June–July, August–September, October–November, and December–January of the following year
MONTH2	January–February, March–April, May–June, July–August, September–October, November–December
QTR3.2	Nine-month intervals starting on February 1, 1960, November 1, 1960, August 1, 1961, May 1, 1962, and so on.
SEMIYEAR.3	Six-month intervals, March–August, and September–February
YEAR.10	Fiscal years starting in October
YEAR2.7	Biennial intervals starting in July of even years

Interval	Description
YEAR2.19	Biennial intervals starting in July of odd years
YEAR4.11	Four-year intervals starting in November of leap years (frequency of U.S. presidential elections)
YEAR4.35	Four-year intervals starting in November of even years between leap years (frequency of U.S. midterm elections)
DTMONTH13	Thirteen-month intervals starting at midnight of January 1, 1960, such as November 1, 1957, December 1, 1958, January 1, 1960, February 1, 1961, and March 1, 1962
HOUR8.7	Eight-hour intervals starting at 6 a.m., 2 p.m., and 10 p.m. (might be used for work shifts)

Retail Calendar Intervals: ISO 8601 Compliant

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 months of each period, respectively.

The intervals that are created from the formats can be used in any of the following functions: INTCINDEX, INTCK, INTCYCLE, INTFIT, INTFMT, INTGET, INTINDEX, INTNX, INTSEAS, INTSHIFT, and INTTEST.

Custom Time Intervals

Reasons for Using Custom Time Intervals

Standard time intervals (for example, QTR, MONTH, WEEK, and so on) do not always fit the data. Also, some time series are measured at standard intervals where there are gaps in the data. For example, you might want to use fiscal months that begin on the 10th day of each month. In this case, using the MONTH interval is not appropriate because the MONTH interval begins on the first day of each month. You can use a custom interval to model data at a frequency that is familiar to the business and to eliminate gaps in the data by compressing the data. The intervals

must be listed in ascending order. There cannot be gaps between intervals, and intervals cannot overlap.

As another example, you might want to collect data hourly for a business that is closed at night. In this case, using the DTHOUR interval results in gaps in the data that can cause problems in standard time series analysis. You might also want to calculate the number of business days between dates, excluding holidays and weekends, but holidays are counted when you use the INTCK function with the WEEKDAY interval. These are cases in which custom intervals can be used effectively.

Using Custom Time Intervals in a SAS Program

You can define custom intervals in a data set within a SAS program. Using a custom interval requires that you follow two steps for each interval:

1 Associate a data set name with a custom interval name by using the INTERVALDS= system option in an OPTIONS statement.

   Here is an example of the arguments in an INTERVALDS= system option. The example associates the data set StoreHoursDS with the custom interval StoreHours:

   ```
   options intervalds=(StoreHours, StoreHoursDS);
   ```

   For more information, see "INTERVALDS= System Option" in *SAS System Options for SAS Viya Workbench*.

2 Create a data set that describes the custom interval.

   The data set must contain the *begin* variable; it can also contain *end* and *season* variables. In your SAS program, include a FORMAT statement that is associated with the *begin* variable that specifies a SAS date, datetime, or numeric format that matches the *begin* variable data. If an *end* variable is present, include it in the FORMAT statement. A numeric format that is not a SAS date or SAS datetime format indicates that the values are observation numbers. If the *end* variable is not present, the implied value of *end* at each observation is one less than the value of *begin* at the next observation.

   Include in the span of the custom interval data set any dates or times that are necessary for performing calculations on the time series, including backcasting, forecasting, and other operations that might extend beyond the series.

   Note: Omitting the *end* variable ensures that no gaps exist in the data definition. Gaps in the data can cause errors in processing, if the END variable is included. The *end* variable must be properly defined to avoid gaps.

After you define custom intervals by using the preceding steps, the custom interval can be specified in SAS procedures and functions in places where a standard time interval can be specified.

The following examples show how to use some of the date and time functions. You can also find additional examples in the SAS Press book excerpt, SAS Functions by

Example, by Ron Cody. This SAS Press book contains information for approximately 200 functions. Also, Ron includes many examples and details concerning optional arguments.

Example 1: Creating Store Hours for a Business Using the INTNX Function

The following DATA step creates the StoreHoursDS data set for a business that is open from 9:00 AM to 6:00 PM Monday through Friday, and Saturday from 9:00 AM to 1:00 PM. The example uses the "INTNX Function" on page 1004, which increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value. In this example, StoreHours is the interval, and StoreHoursDS is the data set that contains user-supplied holidays:

```
options intervalds=(StoreHours=StoreHoursDS);
data StoreHoursDS(keep=begin end);
   start = '01JAN2009'd;
   stop  = '31DEC2009'd;
   do date = start to stop;
      dow = weekday(date);
      datetime=dhms(date,0,0,0);
      if dow not in (1,7) then
         do hour = 9 to 17;
            begin=intnx('hour',datetime,hour,'b');
            end=intnx('hour',datetime,hour,'e');
            output;
         end;
      else if dow = 7 then
         do hour = 9 to 12;
            begin=intnx('hour',datetime,hour,'b');
            end=intnx('hour',datetime,hour,'e');
            output;
         end;
   end;
   format begin end datetime.;
run;
title 'Store Hours Custom Interval';

proc print data=StoreHoursDS (obs=18);
run;
```

The output shows the first 18 observations of the custom interval data set.

Figure 1.1 *A Custom Interval for Store Hours*

Store Hours Custom Interval

Obs	begin	end
1	01JAN09:09:00:00	01JAN09:09:59:59
2	01JAN09:10:00:00	01JAN09:10:59:59
3	01JAN09:11:00:00	01JAN09:11:59:59
4	01JAN09:12:00:00	01JAN09:12:59:59
5	01JAN09:13:00:00	01JAN09:13:59:59
6	01JAN09:14:00:00	01JAN09:14:59:59
7	01JAN09:15:00:00	01JAN09:15:59:59
8	01JAN09:16:00:00	01JAN09:16:59:59
9	01JAN09:17:00:00	01JAN09:17:59:59
10	02JAN09:09:00:00	02JAN09:09:59:59
11	02JAN09:10:00:00	02JAN09:10:59:59
12	02JAN09:11:00:00	02JAN09:11:59:59
13	02JAN09:12:00:00	02JAN09:12:59:59
14	02JAN09:13:00:00	02JAN09:13:59:59
15	02JAN09:14:00:00	02JAN09:14:59:59
16	02JAN09:15:00:00	02JAN09:15:59:59
17	02JAN09:16:00:00	02JAN09:16:59:59
18	02JAN09:17:00:00	02JAN09:17:59:59

Example 2: Creating the Fiscal Month Custom Interval Using the INTNX Function

The following DATA step creates the FMDS data set to define a custom interval, FiscalMonth. It is appropriate for a business that uses fiscal months that start on the 10th day of each month. The SAME alignment option of the INTNX function specifies that the dates that are generated by the INTNX function be the same day of the month as the date in the *start* variable. The MONTH function assigns the month of the *begin* variable to the *season* variable, which specifies monthly seasonality:

```
options intervalds=(FiscalMonth=FMDS);
data FMDS(keep=begin season);
   start = '10JAN1999'd;
   stop  = '10JAN2001'd;
   nmonths = intck('month',start,stop);
   do i=0 to nmonths;
      begin = intnx('month',start,i,'S');
```

```
      season = month(begin);
      output;
   end;
   format begin date9.;
run;

proc print data=FMDS;
   title 'Fiscal Month Data';
run;
```

Figure 1.2 *Fiscal Month Data*

Fiscal Month Data

Obs	begin	season
1	10JAN1999	1
2	10FEB1999	2
3	10MAR1999	3
4	10APR1999	4
5	10MAY1999	5
6	10JUN1999	6
7	10JUL1999	7
8	10AUG1999	8
9	10SEP1999	9
10	10OCT1999	10
11	10NOV1999	11
12	10DEC1999	12
13	10JAN2000	1
14	10FEB2000	2
15	10MAR2000	3
16	10APR2000	4
17	10MAY2000	5
18	10JUN2000	6
19	10JUL2000	7
20	10AUG2000	8
21	10SEP2000	9
22	10OCT2000	10
23	10NOV2000	11
24	10DEC2000	12
25	10JAN2001	1

The difference between the custom FiscalMonth interval and a standard interval is seen in the following example. The output from the program compares how the

data is accumulated. For the FiscalMonth interval, values in the first nine days of the month are accumulated with the interval that begins in the previous month. For the standard MONTH interval, values in the first nine days of the month are accumulated with the calendar month.

```
data sales(keep=date sales);
   do date = '01JAN2000'd to '31DEC2000'd;
      month = MONTH(date);
      dayofmonth = DAY(date);
      sales = 0;
      if (dayofmonth lt 10) then sales= month/9;
      output;
   end;
   format date monyy.;
run;

proc timeseries data=sales out=dataInFiscalMonths;
   id date interval=FiscalMonth accumulate=total;
   var sales;
run;

proc timeseries data=sales out=dataInStdMonths;
   id date interval=Month accumulate=total;
   var sales;
run;

data compare;
   merge dataInFiscalMonths(rename=(sales=FM_sales))
         dataInStdMonths(rename=(sales=SM_sales));
   by date;
run;

title 'Standard Monthly Data and Fiscal Month Data';

proc print data=compare;
run;
```

Figure 1.3 Comparison of Standard Monthly Data and Fiscal Month Data

Standard Monthly Data and Fiscal Month Data

Obs	date	FM_sales	SM_sales
1	10DEC1999	1	.
2	01JAN2000	.	1
3	10JAN2000	2	.
4	01FEB2000	.	2
5	10FEB2000	3	.
6	01MAR2000	.	3
7	10MAR2000	4	.
8	01APR2000	.	4
9	10APR2000	5	.
10	01MAY2000	.	5
11	10MAY2000	6	.
12	01JUN2000	.	6
13	10JUN2000	7	.
14	01JUL2000	.	7
15	10JUL2000	8	.
16	01AUG2000	.	8
17	10AUG2000	9	.
18	01SEP2000	.	9
19	10SEP2000	10	.
20	01OCT2000	.	10
21	10OCT2000	11	.
22	01NOV2000	.	11
23	10NOV2000	12	.
24	01DEC2000	.	12
25	10DEC2000	0	.

Example 3: Using Custom Intervals with the INTCK Function

The following example uses custom intervals in the INTCK function to omit holidays when counting business days:

```
options intervalds=(BankingDays=BankDayDS);
data BankDayDS(keep=begin);
   start = '15DEC1998'd;
```

```
   stop  = '15JAN2002'd;
   nwkdays = intck('weekday',start,stop);
   do i = 0 to nwkdays;
      begin = intnx('weekday',start,i);
      year = year(begin);
      if begin ne holiday('NEWYEAR',year) and
         begin ne holiday('MLK',year) and
         begin ne holiday('USPRESIDENTS',year) and
         begin ne holiday('MEMORIAL',year) and
         begin ne holiday('USINDEPENDENCE',year) and
         begin ne holiday('LABOR',year) and
         begin ne holiday('COLUMBUS',year) and
         begin ne holiday('VETERANS',year) and
         begin ne holiday('THANKSGIVING',year) and
         begin ne holiday('CHRISTMAS',year) then
      output;
   end;
   format begin date9.;
run;
data CountDays;
   start = '01JAN1999'd;
   stop  = '31DEC2001'd;
   ActualDays = intck('DAYS',start,stop);
   Weekdays   = intck('WEEKDAYS',start,stop);
   BankDays   = intck('BankingDays',start,stop);
   format start stop date9.;
run;

title 'Methods of Counting Days';

proc print data=CountDays;
run;
```

Figure 1.4 *Bank Days Custom Interval*

Methods of Counting Days

Obs	start	stop	ActualDays	Weekdays	BankDays
1	01JAN1999	31DEC2001	1095	781	757

Best Practices for Custom Interval Names

The following items list best practices to use when you are creating custom interval names:

- The interval name cannot contain national characters.

- Custom interval names should not conflict with existing SAS interval names. For example, if BASE is a SAS interval name, do not use the following formats for the name of a custom interval:
  - BASE
  - BASE*m*
  - BASE*m.n*
  - DTBASE
  - DTBASE*m*
  - DTBASE*m.n*

  The following paragraphs describe the variables:

  m
  specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type. For example, the interval YEAR2 consists of two-year, or biennial, periods.

  n
  specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For example, YEAR.3 specifies yearly periods that are shifted to start on the first day of March of each calendar year and end in February of the following year.

  If you define a custom interval such as CUSTBASE, then you can use CUSTBASE*m.n*.

  Because of these rules, do not begin a custom interval name with DT, and do not end the custom interval name with a number.

- To ensure that custom intervals work reliably, always include one of the following formats:

  date-format with beginning and ending values
  specifies intervals that are used with SAS date values.

  datetime-format with beginning and ending values
  specifies intervals that are used with SAS datetime values.

  number-format with beginning and ending values
  specifies intervals that are used with SAS observation numbers.

- Beginning and ending values should be of the same type. Both values should be date values, datetime values, or observation numbers.

- Calculations for custom intervals cannot be performed before the first begin value or after the last end value. If you use the begin variable only, then the last end value that you can calculate is the last begin value –1. If you forecast or backcast the time series, be sure to include time definitions for the forecast and backcast values.

- CUSTBASE*m*.2 is never able to calculate a beginning period for the first date value in a data set because, by definition, the beginning of the first interval starts before the data set begins (at the – (*m*– 2) th observation). For example, you might have an interval called CUSTBASE4.2 with the first interval beginning before the first observation:

```
OBS
-2    Start of partial CUSTBASE4.2 interval observation:  -(4-2) = -2.
-1
0
1     End of partial CUSTBASE4.2 interval observation: This is the first
       observation in the data set.
2     Start of first complete CUSTBASE4.2 interval.
3
4
5     End of first complete CUSTBASE4.2 interval.
6     Start of 2nd CUSTBASE4.2 interval.
```

If you execute the INTNX function, the result must return the date that is associated with OBS –2, which does not exist:

```
INTNX('CUSTBASE4.2', date-at-obs1, 0, 'B');
```

- Include a variable named *season* in the custom interval data set to define the seasonal index. This result is similar to the result of `INTINDEX ('interval', date);`

  In the following example, the data set is associated with the custom interval CUSTWEEK:

```
Obs        begin                    season
    1             27DEC59          52
    2             03JAN60            1
    3             10JAN60            2
    4             17JAN60            3
    5             24JAN60            4
    6             31JAN60            5
```

  The following examples show the results of using custom interval functions:

  `INTINDEX ('CUSTWEEK', '03JAN60'D);`
  returns a value of 1.

  `INTSEAS ('CUSTWEEK');`
  returns a value of 52, which is the largest value of the season.

  `INTCYCLE ('CUSTWEEK');`
  returns CUSTWEEK52, which is CUSTBASE*max*(*season*).

  `INTCINDEX ('CUSTWEEK', '27DEC59'D);`
  returns a value of 1.

  `INTCINDEX('CUSTWEEK', '03JAN60'D)`
  returns a value of 2.

  A new cycle begins when the season is less than the previous value of *season*.

- Seasonality occurs when seasons are identified, such as season1, season2, season3, and so on. If all seasons are identified as season1, then there is no seasonality. No seasonality is also called trivial seasonality.

  Only trivial seasonality is available for intervals of the form CUSTBASE*m*. If *season* is not included in the data set, then trivial seasonality is valid.

- If a format for the begin variable is included in a data set, then a message generated by INTFMT ('CUSTBASE', 'l') or INTFMT ('CUSTBASE', 's') appears. The message recommends a format based on the format that is specified in the data set.

- Executing `INTSHIFT ('CUSTBASE');` or `INTSHIFT ('CUSTBASE`*`m`*`.`*`s`*`');` returns the value of CUSTBASE.
- With INTNX, INTCK, and INTTEST, the intervals CUSTBASE, CUSTBASE*m*, and CUSTBASE*m.s* work as expected.

Pattern Matching Using Perl Regular Expressions (PRX)

Definition of Pattern Matching

Pattern matching enables you to search for and extract multiple matching patterns from a character string in one step. Pattern matching also enables you to make several substitutions in a string in one step. You do this by using the PRX functions and CALL routines in the DATA step.

For example, you can search for multiple occurrences of a string and replace those strings with another string. You can search for a string in your source file and return the position of the match. You can find words in your file that are doubled. See "Using Perl Regular Expressions in the DATA Step" for more information.

Definition of Perl Regular Expression (PRX) Functions and CALL Routines

Perl regular expression (PRX) functions and CALL routines refers to a group of functions and CALL routines that use a modified version of Perl as a pattern-matching language to parse character strings. You can perform the following tasks:

- search for a pattern of characters within a string
- extract a substring from a string
- search and replace text with other text
- parse large amounts of text, such as web logs or other text data

Perl regular expressions comprise the character string matching category for functions and CALL routines. For a short description of these functions and CALL routines, see Functions and CALL Routines by Category in the Dictionary section of this document.

Benefits of Using Perl Regular Expressions in the DATA Step

Using Perl regular expressions in the DATA step enhances search-and-replace options in text. You can use Perl regular expressions to perform the following tasks:

- validate data
- replace text
- extract a substring from a string

You can write SAS programs that do not use regular expressions to produce the same results as you do when you use Perl regular expressions. However, the code without the regular expressions requires more function calls to handle character positions in a string and to manipulate parts of the string.

Perl regular expressions combine most, if not all, of these steps into one expression. The resulting code is less prone to error, easier to maintain, and clearer to read.

Using Perl Regular Expressions in the DATA Step

Syntax of Perl Regular Expressions

The Components of a Perl Regular Expression

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. Using metacharacters enables SAS to perform special actions. These actions include forcing the match to begin in a particular location, and matching a particular set of characters. Paired forward slashes are the default delimiters.

The following two examples show metacharacters and the values that they match:

- If you use the metacharacter \d, SAS matches a digit between 0–9.
- Quantifiers are used when a portion of a pattern needs to match a certain number (or numbers) of times. If there is not a quantifier the number of times to

match is one. The maximun value for a quantifier is 2147483646 (which is 2^31) -1). See "Tables of Perl Regular Expression (PRX) Metacharacters" and "Example 5: Using PRX Functions to Match Capturing Groups"

- If you use /\d+/, SAS finds the digits in the string "Raleigh, NC 27506". Here is an example:

```
data _null_;
  re = prxparse('/\d+/');
  string = 'Raleigh, NC 27506, USA';
  x = prxmatch(re, string);
  put x=;
  num=prxposn(re, 0, string);
  put num=;
run;
```

SAS writes the following output to the log:

```
x=13
num=27506
```

You can see lists of PRX metacharacters in "Tables of Perl Regular Expression (PRX) Metacharacters" on page 1597. For a complete list of metacharacters, see the Perl documentation.

Basic Syntax for Finding a Match in a String

You use the PRXMATCH function to find the position of a matched value in a source string. PRXMATCH has the following general form:

```
PRXMATCH('/search-string/ ','source-string'
```

The following example uses the PRXMATCH function to find the position of *search-string* in *source-string*:

```
prxmatch('/world/', 'Hello world!');
```

The result of PRXMATCH is the value 7, because *world* occurs in the seventh position of the string *Hello world!*.

Basic Syntax for Searching and Replacing Text

The basic syntax for searching and replacing text has the following form:

```
s/regular-expression/replacement-string/
```

The following example uses the PRXCHANGE function to show how substitution is performed:

```
prxchange('s/world/planet/', 1, 'Hello world!');
```

Arguments

s
: specifies the metacharacter for substitution.

world
specifies the regular expression.

planet
specifies the replacement value for *world*.

1
specifies that the search ends when one match is found.

Hello world!
specifies the source string to be searched.

The result of the substitution is **`Hello planet`**.

Another Example of Using Basic Syntax for Searching and Replacing Text

Another example of using the PRXCHANGE function changes the value *Jones, Fred* to *Fred Jones*:

```
prxchange('s/(\w+), (\w+)/$2 $1/',-1, 'Jones, Fred');
```

In this example, the Perl regular expression is `s/(\w+), (\w+)/$2 $1`. The number of times to search for a match is –1. The source string is 'Jones, Fred'. The value –1 specifies that matching patterns continue to be replaced until the end of the source is reached.

The Perl regular expression can be divided into its elements:

s
specifies a substitution regular expression.

(\w+)
matches one or more word characters (alphanumeric and underscore). The parentheses indicate that the value is stored in capture buffer 1.

,<space>
matches a comma and a space.

(\w+)
matches one or more word characters (alphanumeric and underscore). The parentheses indicate that the value is stored in capture buffer 2.

/
separator between the regular expression and the replacement string.

$2
part of the replacement string that substitutes the value in capture buffer 2, which in this case is the word after the comma, puts the substitution in the results.

<space>
puts a space in the result.

$1
puts capture buffer 1 into the result. In this case, it is the word before the comma.

Replacing Text

The following example uses the \u and \L metacharacters to replace the second character in MCLAUREN with a lowercase letter:

```
data _null_;
   x = 'MCLAUREN';
   x = prxchange("s/(MC)/\u\L$1/i", -1, x);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=McLAUREN
```

Example 1: Validating Data

You can test for a pattern of characters within a string. For example, you can examine a string to determine whether it contains a correctly formatted telephone number. This type of test is called data validation.

The following example validates a list of phone numbers. To be valid, a phone number must have one of the following forms: `(XXX) XXX-XXXX` or `XXX-XXX-XXXX`.

```
   data _null_;  1
      if _N_ = 1 then
         do;
            paren = "\([2-9]\d\d\) ?[2-9]\d\d-\d\d\d\d";  2
            dash = "[2-9]\d\d-[2-9]\d\d-\d\d\d\d";  3
            expression = "/(" || paren || ")|(" || dash || ")/";  4
            retain re;
            re = prxparse(expression);  5
            if missing(re) then  6
               do;
                  putlog "ERROR: Invalid expression " expression;  7
                  stop;
               end;
         end;

   length first last home business $ 16;
   input first last home business;

      if ^prxmatch(re, home) then  8
         putlog "NOTE: Invalid home phone number for " first last home;

      if ^prxmatch(re, business) then  9
         putlog "NOTE: Invalid business phone number for " first last
   business;

      datalines;
   Jerome Johnson (919)319-1677 (919)846-2198
```

```
Romeo Montague 800-899-2164 360-973-6201
Imani Rashid (508)852-2146 (508)366-9821
Palinor Kent . 919-782-3199
Ruby Archuleta . .
Takei Ito 7042982145 .
Tom Joad 209/963/2764 2099-66-8474
;
run;
```

The following items correspond to the lines that are numbered in the preceding DATA step.

1 Create a DATA step.

2 Build a Perl regular expression to identify a phone number that matches (XXX)XXX-XXXX, and assign the variable PAREN to hold the result. Use the following syntax elements to build the Perl regular expression:

\(	matches the open parenthesis in the area code.
[2–9]	matches the digits 2–9, which is the first number in the area code.
\d	matches a digit, which is the second number in the area code.
\d	matches a digit, which is the third number in the area code.
\)	matches the closed parenthesis in the area code.
<space>?	matches the space (which is the preceding subexpression) zero or one time. Spaces are significant in Perl regular expressions. They match a space in the text that you are searching. If a space precedes the question mark metacharacter (as it does in this case), the pattern matches either zero spaces or one space in this position in the phone number.

3 Build a Perl regular expression to identify a phone number that matches XXX-XXX-XXXX, and assign the variable DASH to hold the result.

4 Build a Perl regular expression that concatenates the regular expressions for (XXX)XXX-XXXX and XXX—XXX—XXXX. The concatenation enables you to search for both phone number formats from one regular expression.

The PAREN and DASH regular expressions are placed within parentheses. The bar metacharacter (|) that is located between PAREN and DASH instructs the compiler to match either pattern. The slashes around the entire pattern tell the compiler where the start and end of the regular expression is located.

5 Pass the Perl regular expression to PRXPARSE and compile the expression. PRXPARSE returns a value to the compiled pattern. Using the value with other Perl regular expression functions and CALL routines enables SAS to perform operations with the compiled Perl regular expression.

6 Use the MISSING function to check whether the regular expression was successfully compiled.

7 Use the PUTLOG statement to write an error message to the SAS log if the regular expression did not compile.

8 Search for a valid home phone number. PRXMATCH uses the value from PRXPARSE along with the search text and returns the position where the

regular expression was found in the search text. If there is no match for the home phone number, the PUTLOG statement writes a note to the SAS log.

9 Search for a valid business phone number. PRXMATCH uses the value from PRXPARSE along with the search text and returns the position where the regular expression was found in the search text. If there is no match for the business phone number, the PUTLOG statement writes a note to the SAS log.

Example Code 1.2 *Output from Validating Data*

```
NOTE: Invalid home phone number for Palinor Kent
NOTE: Invalid home phone number for Ruby Archuleta
NOTE: Invalid business phone number for Ruby Archuleta
NOTE: Invalid home phone number for Takei Ito 7042982145
NOTE: Invalid business phone number for Takei Ito
NOTE: Invalid home phone number for Tom Joad 209/963/2764
NOTE: Invalid business phone number for Tom Joad 2099-66-8474
```

Example 2: Matching and Replacing Text

This example uses a Perl regular expression to find a match and replace the matching characters with other characters. PRXPARSE compiles the regular expression and uses PRXCHANGE to find the match and perform the replacement. The example replaces all occurrences of a less than sign with `<`, a common substitution when converting text to HTML.

```
data _null_;  1
   input;  2
   _infile_ = prxchange('s/</&lt;/', -1, _infile_);  3
   put _infile_;  4
   datalines;  5
x + y < 15
x < 10 < y
y < 11
;
run;
```

The following items correspond to the numbered lines in the preceding DATA step.

1 Create a DATA step.

2 Bring an input data record into the input buffer without creating any SAS variables.

3 Call the PRXCHANGE routine to perform the pattern exchange. The format for the regular expression is `s/regular-expression/replacement-text/`. The `s` before the regular expression signifies that this is a substitution regular expression. The –1 is a special value that is passed to PRXCHANGE and indicates that all possible replacements should be made.

4 Write the current output line to the log by using the _INFILE_ option with the PUT statement.

5 Identify the input file.

Example Code 1.3 *Output from Replacing Text*

```
x + y &lt; 15
x &lt; 10 &lt; y
y &lt; 11
```

The ability to pass a regular expression to PRXCHANGE and return a result enables calling PRXCHANGE from a PROC SQL query. The following query produces a column with the same character substitution as in the preceding example. From the input table the query reads `text_lines`, changes the text for the column `line`, and places the results in a column named `html_line`:

```
proc sql;
   select prxchange('s/</&lt;/', -1, line)
   as html_line
   from text_lines;
quit;
```

Example 3: Extracting a Substring from a String

You can use Perl regular expressions to find and easily extract text from a string. In this example, the DATA step creates a subset of North Carolina business phone numbers. The program extracts the area code and checks it against a list of area codes for North Carolina.

```
data _null_;  1
   if _N_ = 1 then
      do;
         paren = "\(([2-9]\d\d)\) ?[2-9]\d\d-\d\d\d\d";  2
              dash = "([2-9]\d\d)-[2-9]\d\d-\d\d\d\d";  3
         regexp = "/(" || paren || ")|(" || dash || ")/";  4
         retain re;
         re = prxparse(regexp);  5
         if missing(re) then  6
            do;
               putlog "ERROR: Invalid regexp " regexp;  7
               stop;
            end;

         retain areacode_re;
         areacode_re = prxparse("/828|336|704|910|919|252/");  8
         if missing(areacode_re) then
            do;
               putlog "ERROR: Invalid area code regexp";
               stop;
            end;
      end;

   length first last home business $ 25;
   length areacode $ 3;
   input first last home business;
```

```
   if ^prxmatch(re, home) then
      putlog "NOTE: Invalid home phone number for " first last home;

   if prxmatch(re, business) then  9
      do;
         which_format = prxparen(re);  10
         call prxposn(re, which_format, pos, len);  11
         areacode = substr(business, pos, len);
         if prxmatch(areacode_re, areacode) then  12
            put "In North Carolina: " first last business;
      end;
      else
         putlog "NOTE: Invalid business phone number for " first last
business;
   datalines;
Jerome Johnson (919)319-1677 (919)846-2198
Romeo Montague 800-899-2164 360-973-6201
Imani Rashid (508)852-2146 (508)366-9821
Palinor Kent 704-782-4673 704-782-3199
Ruby Archuleta 905-384-2839 905-328-3892
Takei Ito 704-298-2145 704-298-4738
Tom Joad 515-372-4829 515-389-2838
;
```

1 Create a DATA step.

2 Build a Perl regular expression to identify a phone number that matches (XXX)XXX-XXXX, and assign the variable PAREN to hold the result. Use the following syntax elements to build the Perl regular expression:

\(	matches the open parenthesis in the area code. The open parenthesis marks the start of the submatch.
[2–9]	matches the digits 2–9.
\d	matches a digit, which is the second number in the area code.
\d	matches a digit, which is the third number in the area code.
\)	matches the closed parenthesis in the area code. The closed parenthesis marks the end of the submatch.
?	matches the space (which is the preceding subexpression) zero or one time. Spaces are significant in Perl regular expressions. They match a space in the text that you are searching. If a space precedes the question mark metacharacter (as it does in this case), the pattern matches either zero spaces or one space in this position in the phone number.

3 Build a Perl regular expression to identify a phone number that matches XXX-XXX-XXXX, and assign the variable DASH to hold the result.

4 Build a Perl regular expression that concatenates the regular expressions for (XXX)XXX-XXXX and XXX—XXX—XXXX. The concatenation enables you to search for both phone number formats from one regular expression.

The PAREN and DASH regular expressions are placed within parentheses. The bar metacharacter (|) that is located between PAREN and DASH instructs the

compiler to match either pattern. The slashes around the entire pattern tell the compiler where the start and end of the regular expression is located.

5 Pass the Perl regular expression to PRXPARSE and compile the expression. PRXPARSE returns a value to the compiled pattern. Using the value with other Perl regular expression functions and CALL routines enables SAS to perform operations with the compiled Perl regular expression.

6 Use the MISSING function to check whether the Perl regular expression compiled without error.

7 Use the PUTLOG statement to write an error message to the SAS log if the regular expression did not compile.

8 Compile a Perl regular expression that searches a string for a valid North Carolina area code.

9 Search for a valid business phone number.

10 Use the PRXPAREN function to determine which submatch to use. PRXPAREN returns the last submatch that was matched. If an area code matches the form (XXX), PRXPAREN returns the value 2. If an area code matches the form XXX, PRXPAREN returns the value 4.

11 Call the PRXPOSN routine to retrieve the position and length of the submatch.

12 Use the PRXMATCH function to determine whether the area code is a valid North Carolina area code, and write the observation to the log.

Example Code 1.4 *Output from Extracting a Substring from a String*

```
In North Carolina: Jerome Johnson (919)846-2198
In North Carolina: Palinor Kent 704-782-3199
In North Carolina: Takei Ito 704-298-4738
```

Example 4: Another Example of Extracting a Substring from a String

In this example, the PRXPOSN function is passed to the original search text instead of to the position and length variables. PRXPOSN returns the text that is matched.

```
data _null_;  1
   length first last phone $ 16;
   retain re;
   if _N_ = 1 then do;  2
      re=prxparse("/\(([2-9]\d\d)\) ?[2-9]\d\d-\d\d\d\d/");  3
   end;

   input first last phone & 16.;
   if prxmatch(re, phone) then do;  4
      area_code = prxposn(re, 1, phone);  5
      if area_code ^in ("828"
                                   "336"
```

```
                                    "704"
                                    "910"
                                    "919"
                                    "252") then
          putlog "NOTE: Not in North Carolina: "
                       first last phone;   6
      end;
     datalines;  7
Thomas Archer     (919)319-1677
Lucy Mallory       (800)899-2164
Tom Joad              (508)852-2146
Laurie Jorgensen  (252)352-7583
;
run;
```

The following items correspond to the numbered lines in the preceding DATA step.

1 Create a DATA step.

2 If this is the first record, find the value of *re*.

3 Build a Perl regular expression for pattern matching. Use the following syntax elements to build the Perl regular expression:

/	is the beginning delimiter for a regular expression.
\(	marks the next character entry as a character or a literal.
(	marks the start of the submatch.
[2–9]	matches the digits 2–9 and identifies the first number in the area code.
\d	matches a digit, which is the second number in the area code.
\d	matches a digit, which is the third number in the area code.
\)	matches the close parenthesis in the area code. The close parenthesis marks the end of the submatch.
?	matches the space (which is the preceding subexpression) zero or one time. Spaces are significant in Perl regular expressions. The spaces match a space in the text that you are searching. If a space precedes the question mark metacharacter (as it does in this case), the pattern matches either zero spaces or one space in this position in the phone number.
||	is the concatenation operator.
[2–9]	matches the digits 2–9 and identifies the first number in the seven-digit phone number.
\d	matches a digit, which is the second number in the seven-digit phone number.
\d	matches a digit, which is the third number in the seven-digit phone number.
–	is the hyphen between the first three and last four digits of the phone number after the area code.
\d	matches a digit, which is the fourth number in the seven-digit phone number.

\d matches a digit, which is the fifth number in the seven-digit phone number.

\d matches a digit, which is the sixth number in the seven-digit phone number.

\d matches a digit, which is the seventh number in the seven-digit phone number.

/ is the ending delimiter for a regular expression.

4 Return the position at which the string begins.

5 Identify the position at which the area code begins.

6 Search for an area code from the list. If the area code is not valid for North Carolina, use the PUTLOG statement to write a note to the SAS log.

7 Identify the input file.

Example Code 1.5 *Output from Extracting a Substring from a String*

```
NOTE: Not in North Carolina: Lucy Mallory (800)899-2164
NOTE: Not in North Carolina: Tom Joad (508)852-2146
```

Example 5: Using PRX Functions to Match Capturing Groups

In this example PRXPARSE, PRXMATCH, and PRXPOSN functions match capturing groups. Perl 5.40.0 is used.

```
data _null_;                          1
   regex = '/((foo)|(bar))*/';        2
   re  = prxparse(regex);             3
   str = 'foobar';                    4
   if prxmatch(re, str) then          5
      do;
         s1=prxposn(re, 1, str);
         s2=prxposn(re, 2, str);
         s3=prxposn(re, 3, str);
         put s1= / s2= / s3=;         6
      end;
run;
```

1 Create a DATA step.

2 Assign '/((foo)|(bar))*/' to the variable *regex*.

3 Use PRXPARSE to parse the value of *regex*

4 Assign the value, foobar, to the variable *str*.

5 Using the PRXMATCH and PRXPOSN functions, determine if the values of *re* and *str* match.

The expression, "/((foo)|(bar))*/" contains three capturing groups:

- Group 1 - ((foo)|(bar))
- Group 2 - (foo)
- Group 3 - (bar)

Repeated capturing groups capture the latest iteration, so "bar" is matched in Group 1 and Group 3. In Group 2 "foo" is not a match, so the output is blank.

6 Output the value of *s1, s2, s3*.

The following output is displayed. Notice that *s2* is blank.

```
s1=bar
s2=
s3=bar
```

Writing Perl Debug Output to the SAS Log

The DATA step provides debugging support with the CALL PRXDEBUG routine. CALL PRXDEBUG enables you to turn on and off Perl debug output messages that are sent to the SAS log.

The following example writes Perl debug output to the SAS log.

```
data _null_;

      /* CALL PRXDEBUG(1) turns on Perl debug output. */
   call prxdebug(1);
   putlog 'PRXPARSE: ';
   re = prxparse('/[bc]d(ef*g)+h[ij]k$/');
   putlog 'PRXMATCH: ';
   pos = prxmatch(re, 'abcdefg_gh_');

      /* CALL PRXDEBUG(0) turns off Perl debug output. */
   call prxdebug(0);
run;
```

SAS writes the following output to the log.

Example Code 1.6 *SAS Debugging Output*

```
PRXPARSE:
Compiling REx '[bc]d(ef*g)+h[ij]k$'
size 41 first at 1
rarest char g at 0
rarest char d at 0
   1: ANYOF[bc](10)
  10: EXACT <d>(12)
  12: CURLYX[0] {1,32767}(26)
  14:   OPEN1(16)
  16:     EXACT <e>(18)
  18:     STAR(21)
  19:       EXACT <f>(0)
  21:     EXACT <g>(23)
  23:   CLOSE1(25)
  25:   WHILEM[1/1](0)
  26: NOTHING(27)
  27: EXACT <h>(29)
  29: ANYOF[ij](38)
  38: EXACT <k>(40)
  40: EOL(41)
  41: END(0)
anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating) stclass
'ANYOF[bc]' minlen 7

PRXMATCH:
Guessing start of match, REx '[bc]d(ef*g)+h[ij]k$' against 'abcdefg_gh_'...
Did not find floating substr 'gh'...
Match rejected by optimizer
```

For a detailed explanation of Perl debug output, see "CALL PRXDEBUG Routine" on page 283.

Perl Artistic License Compliance

Perl regular expressions are supported beginning with SAS® 9.

Perl 5.40 is used by PRX functions on LAX.

Perl is compiled into a library for use with SAS. This library is shipped with SAS® 9. The Perl files are freely available in a ZIP file from the Technical Support Web site. The ZIP file is provided to comply with the Perl Artistic License and is not required in order to use the PRX functions. Each of the modified files has a comment block at the top of the file describing how and when the file was changed. The executables were given nonstandard Perl names. The standard version of Perl can be obtained from the Perl website.

Only Perl regular expressions are accessible from the PRX functions. Other parts of the Perl language are not accessible. The modified version of Perl regular expressions does not support the following items:

- Perl variables (except the capture buffer variables $1 - $n, which are supported).

- The regular expression options /c and /g, and the /e option with substitutions.
- The regular expression option /o in SAS® 9. (It is supported in SAS 9.1 and later.)
- Named characters, which use the \N{name} syntax.
- The metacharacters \pP, \PP, and \X.
- Executing Perl code within a regular expression, which includes the syntax (?{code}), (??{code}), and (?p{code}).
- Unicode pattern matching.
- Using ?PATTERN?. ? is treated like an ordinary regular expression start and end delimiter.
- The metacharacter \G.
- Perl comments between a pattern and replacement text. For example: s{regexp} # perl comment {replacement} is not supported.
- Matching backslashes with m/\\\\/. Instead, use m/\\/ to match a backslash.

SAS Functions for Web Applications

Four functions that manipulate web-related content are available in Base SAS software. HTMLENCODE and URLENCODE return encoded strings. HTMLDECODE and URLDECODE return decoded strings.

Using Git Functions in SAS

Basic Workflow for SAS Git Functions

The SAS Git functions enable you to interact with a remote Git repository from SAS. Here is a basic workflow for using the Git functions.

Creating a Local Repository

To clone a repository, use the GIT_CLONE function. Cloning a remote repository to the SAS server creates a local copy of a Git repository in a directory on the SAS server. The cloned repository appears and functions like a directory on the SAS server, but with the added benefits of Git version control and change tracking. You only need to clone the repository the first time you connect to the repository. The

local repository persists when you close your SAS session and is available to use during your next session. If you want to create a new local repository without immediately pulling in files from a remote repository, use the GIT_INIT_REPO function.

Retrieve Updates from a Remote Repository

To pull and merge updates from the remote repository, use the GIT_PULL function. To fetch updates from the remote repository without automatically merging updates, use the GIT_FETCH function. After a repository has been cloned, any committed updates from other users to the remote repository do not appear in your local repository. If there are changes to a file, SAS automatically attempts to merge remote updates with your local repository when you use the GIT_PULL function. If there are new files in the remote repository, they are copied into your local repository. If there are conflicts, the pull fails. An error is returned to the log and the conflicts must be resolved before you can successfully pull the updates.

Stage Changes in a Local Repository

To stage or unstage changes that you have made in your local repository, use the GIT_INDEX_ADD function and the GIT_INDEX_REMOVE function. When working in your local repository, you might have files that are ready to be committed and other files that you do not want to commit. The default state of files in the local repository is unstaged. To determine which files are ready to be committed, stage the files. Staging files must be completed by using the GIT_INDEX_ADD function. If you decide a staged file is not ready to be committed, unstage the file using the GIT_INDEX_REMOVE function.

Commit Changes to a Local Repository

To commit changes to your local repository, use the GIT_COMMIT function. Staged files can be found by using the GIT_STATUS function and inspecting the staged indicator. A successful commit creates a unique save of the current state of the local directory. Each commit has a unique commit ID that can be reverted to at a later time if needed by using the GIT_RESET function. Committing files does not automatically transfer changes to the remote repository.

Push Changes to a Remote Repository

To push committed changes from your local repository to the remote repository, use the GIT_PUSH function. Pushing to the remote repository enables other users to see changes that you have made in your local repository. Your local repository is not altered by pushing to the remote repository. If there are conflicts between the

local repository and remote repository, the push fails. An error is returned to the log and must be resolved before you can successfully push your committed changes.

SAS LOCKDOWN for the SAS GIT Functions

By default, SAS LOCKDOWN does not disable the SAS GIT functions. Starting in 2025.07, SAS administrators can disable GIT functions by specifying VIYA_LOCKDOWN_USER_DISABLED_METHODS=GIT, which removes the GIT access method from the lockdown list.

Additional Git Resources

- Understanding Git Integration in SAS Studio.
- For help that is specific to GitHub, see *GitHub Help*.

Example Git Workflow Scenarios

Basic Git Workflow

For this example, you are asked to work on a project that stores SAS programs in a Git repository. You need to make minor changes to one file, and then add a new file to the project.

In this sample Git scenario, complete these steps:

1 Clone the remote repository to your local Compute Server.

   a Locate and copy the URI for the remote repository that you want to clone.

   b Use the GIT_CLONE function to clone the repository to your local workspace server. Specify an empty directory on your SAS server for your local repository. Use a PUT statement to see the SAS return code in the log to determine whether the clone was successful.

   This code example shows how to clone with SSH keys:

```
data _null_;
 rc = git_clone(
  "ssh-remote-repository-url",
  "target-directory",
  "ssh-user-name",
  "ssh-password",
  "ssh-public-key",
  "ssh-private-key");
 put rc=;
```

```
run;
```

2 Pull updates from the remote repository to your local repository.

   a The GIT_PULL function is the easiest way to synchronize your local repository with the remote repository. If you are working in your local repository immediately after cloning, there might not be any updates to pull. However, as a best practice, pull your updates before starting your work to avoid conflicts.

   b If there are conflicts between the remote repository and the local repository that cannot automatically be merged, the pull fails. An error message from Git is returned to the log with details about the conflicts. When a pull is successful, a note is returned to the SAS log stating that the pull was successful.

   This example code shows how to pull using SSH keys:

```
data _null_;
 rc= git_pull(
  "your-local-repository",
  "ssh-user-name",
  "ssh-password",
  "ssh-public-key",
  "ssh-private-key");
run;
```

3 Create a file and make changes to an existing file in your local repository.

   a Create a file and save it to the local repository.

   b Open your existing file to edit the code, and then save the file to the local repository.

4 Stage and commit your changes to your local repository.

   a When you are finished working in your local repository, stage and commit your changes to save your current repository.

   b To stage a file, you need the name and status attributes from the file. You can retrieve those attributes using the GIT_STATUS function. When the function runs successfully, a note is returned to the SAS log that contains the name, staged indicator, and status of every new or altered file in the local repository.

   This example code shows how to run the GIT_STATUS function:

```
data _null_;
 rc= git_status("your-local-repository");
 put rc=;
run;
```

   c Use the attributes that are returned from the GIT_STATUS function to stage up to *n* files with the GIT_INDEX_ADD function. Here, *n* is the number of status objects that are returned by the GIT_STATUS function. The first argument in the GIT_INDEX_ADD function is the local repository. After the first argument, stage the files by specifying the name attribute, and then specify the status attribute for each file. The name attribute is the relative file path for the file inside your local repository. You can repeat the "*name*",

"status" pattern up to *n* times to selectively stage the files in your local repository. The order of the files in the function has no effect on staging the files. If you want to unstage a file, use the GIT_INDEX_REMOVE function.

This example code shows how to stage files:

```
data _null_;
 rc= git_index_add(
  "your-local-repository",
  "your-file-path-1",
  "your-file-status-1",
  "your-file-path-n",
  "your-file-status-n");
run;
```

d After you stage all of the files, commit the local repository. To see which files will be committed, use the GIT_STATUS function and inspect the value of the staged indicator. Files that are marked TRUE are committed, and files that are marked FALSE are not committed. Verify that the files' staged indicators are correct, and then use the GIT_COMMIT function to commit the changes to the local repository. To commit to the head of the currently checked out branch, specify "HEAD" for *"your-update-reference"*.

Note: Committing to the local repository does not affect files in the remote repository.

This example code shows how to commit changes to the local repository:

```
data _null_;
 rc= git_commit(
  "your-local-repository",
  "HEAD",
  "your-commit-author",
  "your-author-email",
  "your-commit-message");
run;
```

5 Pull the remote repository and then push your changes to the remote repository.

a To share your local repository updates with the remote repository, use the GIT_PUSH function. Use the GIT_PULL function to synchronize updates from the remote repository before you push the updates. Conflicts must be resolved before you can successfully push to the remote repository.

b When there are no conflicts between your local repository and the remote repository, use the GIT_PUSH function to push your updates to the remote repository. The current committed contents of your local repository are shared with the remote repository. Existing files are updated and new files are added. A successful push returns a note to the log, stating that the push was successful. Use your Git account credentials or your SSH keys to authenticate your push.

This example code shows how to push updates using SSH keys:

```
data _null_;
 rc= git_pull(
   "your-local-repository",
```

```
      "ssh-user-name",
      "ssh-password",
      "ssh-public-key",
      "ssh-private-key");
    rc= git_push(
      "your-local-repository",
      "ssh-user-name",
      "ssh-password",
      "ssh-public-key",
      "ssh-private-key");
   run;
```

Programming Using Git Functions

Here is a typical program using the Git functions:

```
data _null_;
 rc= git_clone(                                    /* 1 */
  "your-repository-uri",
  "your-target-directory",
  "your-ssh-user",
  "your-ssh-password",
  "your-public-ssh-key",
  "your-ssh-private-key");
 put rc=;
run;


data _null_;
 rc= git_pull(                                     /* 2 */
  "your-local-repository",
  "you-ssh-user",
  "your-ssh-password",
  "your-public-ssh-key",
  "your-ssh-private-key");
run;


data _null_;
    n = git_status("your-local-repository");      /* 3 */
    put n=;
run;

data _null_;
 rc= git_index_add(                                 /* 4 */
  "your-local-repository",
  "demo_file.sas",
  "modified",
  "new_file.sas",
  "new");
run;

data _null_;
 rc= git_commit(                                   /* 5 */
```

```
  "your-local-repository",
  "HEAD",
  "your-name",
  "your-email",
  "your-commit-message");
run;

data _null_;
 rc= git_pull(                                    /* 6 */
  "your-local-repository",
  "your-ssh-user",
  "your-ssh-password",
  "your-public-ssh-key",
  "your-ssh-private-key");
run;

data _null_;
 rc= git_push(                                    /* 7 */
  "your-local-repository",
  "your-ssh-user",
  "your-ssh-password",
  "your-public-ssh-key",
  "your-private-ssh-key");
run;
```

1 Clone the remote repository to a local directory on your SAS server.

2 Check for updates from the remote repository using the GIT_PULL function before making your own changes.

3 After making your own changes and creating a new file, use the GIT_STATUS function to see the status objects in your local repository. The log returns a note stating that there are two objects, "demo_file.sas" and “new_file.sas”.

4 Use the GIT_INDEX_ADD function to stage the modified file "demo_file.sas" and the new file “new_file.sas”.

5 Commit the changes to "demo_file.sas" and the new file “new_file.sas” to the local repository using the GIT_COMMIT function Specify "HEAD" to commit to the head of the currently checked out branch.

6 Synchronize your local repository with the remote repository using the GIT_PULL function before pushing to avoid conflicts.

7 Push changes from your local repository to the remote repository with the GIT_PUSH function.

Advanced and Utility SAS Git Functions

Using Branches in SAS

Branches are created by using the GIT_BRANCH_NEW function. Switch between branches using the GIT_BRANCH_CHKOUT function. When you clone a repository, the default main branch in the local repository is identified by "main". If you want to work outside the default main branch, create a branch, and then check out that branch.

To merge branches use the GIT_BRANCH_MERGE function. Merging pulls updates from a specified branch into your currently checked-out branch. After the branches are merged, you can delete the branch by using the GIT_BRANCH_DELETE function. Alternatively, you can also rebase a branch to a specified commit ID instead of using the GIT_BRANCH_MERGE function by using the GIT_REBASE function.

To share the new branch with other users on the remote repository, push the branch to the remote repository. To push a new branch, use the same steps as when pushing the main branch: stage the files, commit the files to the local repository, and then push the files to the remote repository. The new branch appears in the remote repository with the same name, and is available to other users. Users can use a fetch all to fetch all of the remote branches, including the new branch, to their local repository. However, pull and merge requests for branches in the remote repository cannot be performed through SAS and must be done through other means (for example, on GitHub).

Resetting and Deleting Files and Repositories

When working in your local repository, you can reset a file or the entire repository. Single files are reset to the last committed state. The local repository can be reset to any commit ID.

To reset a single file to the last committed version, use the GIT_RESET_FILE function. Any changes made since the last commit are permanently discarded and cannot be restored. If you are unsure what changes are uncommitted in a file, use the GIT_DIFF_FILE_IDX function. The GIT_DIFF_FILE_IDX function returns the changes made to a staged file compared to the last committed version in the local repository.

To reset a repository to a different commit ID, use the GIT_RESET function. Contents of the working directory and index can be kept or discarded by specifying the type of reset. Untracked files, or new files that have not been staged, are not affected by THE GIT_RESET function. There are three different types of resets:

Table 1.6 *Reset Types*

Reset Type	Description
Hard	Resets the current branch to the prior commit, resets your staged changes (the index), and discards all changes in your working directory. You cannot undo a hard reset.
Mixed	Resets the current branch to the prior commit and resets your staged changes (the index). No changes are made to your working directory.
Soft	Resets the current branch to the prior commit. No changes are made to the staged changes (the index) or your working directory.

To delete a local repository from the SAS server, use the GIT_DELETE_REPO function. The repository, all of its branches, and all of its contents are deleted from the SAS server. However, you can clone the remote repository to the SAS server again at any time with the GIT_CLONE function.

Stashing Local Changes

When working in your local repository, you can store all of the changes that you have made since your last commit in a stash by using the GIT_STASH function. Stashing stores all of the changes in the stash stack and reverts your local repository to its last committed state. You can reapply the changes that are stored in your stash by using the GIT_STASH_APPLY function. You can delete the changes that are stored in your stash by using the GIT_STASH_DROP function. Multiple stashes can be stored at the same time. To apply the most recent stash and drop that stash at the same time, use the GIT_STASH_POP function.

Working with Record Objects

The SAS Git functions use the following record objects to store different classes of attributes for local repositories. The record object functions return *n*. Here *n* indicates the number of objects in the record. For example, if the GIT_STATUS function returns 5, then there are five different status record objects in the local repository. Each record object has a corresponding function that retrieves a specified attribute from the *n*th specified object in the record. The returned attribute is stored in an output variable. When you are finished working with a record object, clear the record object with its corresponding free function. It is important to clear record objects after using them to free system memory.

Table 1.7 *Attribute Retrieving Functions*

Create Record Function	Description	Retrieve Attribute Function	Valid Attributes	Clear Record Function
"GIT_STATUS Function"	Stores status information for files in the local repository	"GIT_STATUS_GET Function"	"path", "staged", "status"	"GIT_STATUS_FREE Function"
"GIT_COMMIT_LOG Function"	Stores information about commits that have been made to the local repository	"GIT_COMMIT_GET Function"	"author", "children_ids", "committer", "committer_email", "email", "id", "in_current_branch", "message", "parent_ids", "stash", "time"	"GIT_COMMIT_FREE Function"
"GIT_DIFF Function"	Stores information about the differences between two different commit IDs	"GIT_DIFF_GET Function"	"diff_content", "diff_type", "file"	"GIT_DIFF_FREE Function"

Because these functions store the specified attribute in an output variable, you can use them to programmatically insert file attributes as arguments into other functions. For example, using the GIT_STATUS_GET function, you can store the path and status attributes for a file in variables, and then stage the attribute values using the GIT_INDEX_ADD function. Using this staging method eliminates the need to manually enter attribute values as arguments in the function. The following example shows a method of cycling through the status record object attributes in a local repository.

1 In this example, the repository has two changes on two files: *your-file-1* is an existing file that has been modified, and *your-file-2* is a new file. Neither file has been staged. Use the GIT_STATUS function on your repository to return the number of status objects in your repository:

```
data _null_;
```

```
    n = git_status("Your-Repository");
    put n=;
run;
```

The preceding statements produce these results:

```
NOTE: Entry: your-file-1. Staged: False. Status: Modified.
 NOTE: Entry: your-file-2. Staged: False. Status: New.
n=2
```

2 This example code shows how to cycle through the attributes of each status object to determine their values and use them in other functions:

```
data _null_;

    length path $ 1024;
    length status $ 1024;
    length staged $ 32;

    path="";
    status="";
    staged="";
     do i=1 to n;
       rc=git_status_get(i,"Your-Repository","PATH",path);
       put rc=;
       put path=;
       rc=git_status_get(i,"Your-Repository","STATUS",status);
       put rc=;
       put status=;
       rc=git_status_get(i,"Your-Repository","STAGED",staged);
       put rc=;
       put staged=;
     end;
run;
```

The preceding statements produce these results:

```
path=your-file-1
 rc=0
 status=modified
 rc=0
 staged=FALSE
 rc=0
 path=your-file-2
 rc=0
 status=new
 rc=0
 staged=FALSE
```

Exporting Large Diffs to Text Files

The maximum length for a character variable in SAS is 32,767 bytes. Because diff content for large files and repositories can exceed the maximum character variable length, the diff functions in SAS can also export diff content to a text file without

the 32,767-byte size limit. To export the diff content for an individual file in the index, use the "GIT_DIFF_FILE_IDX Function" and specify a file path for the optional argument "*output-file-path*". To export diff content to a text file for an entire repository between two commit IDs, use the GIT_DIFF_TO_FILE function.

Checking the Version of the Libgit2 Package

The SAS Git functions use the libgit2 package. Use the GIT_VERSION function to view the current version of the libgit2 package.

Deprecated Git Functions

The GITFN_ functions were deprecated and replaced with the GIT_ functions. If you previously used the GITFN_ functions, use this table to find the GITFN function that corresponds to the GIT_ function that you want to use. If a function is not listed in the New GIT_ Function Names table, then that function does not have a corresponding GITFN_ function.

Table 1.8 *New GIT_ Function Names*

GIT_ Functions	Deprecated GITFN_ Functions
"GIT_BRANCH_CHKOUT Function"	"GITFN_CO_BRANCH Function"
"GIT_BRANCH_DELETE Function"	"GITFN_DEL_BRANCH Function"
"GIT_BRANCH_MERGE Function"	"GITFN_MRG_BRANCH Function"
"GIT_BRANCH_NEW Function"	"GITFN_NEW_BRANCH Function"
"GIT_CLONE Function"	"GITFN_CLONE Function"
"GIT_COMMIT_FREE Function"	"GITFN_COMMITFREE Function"
"GIT_COMMIT Function"	"GITFN_COMMIT Function"
"GIT_COMMIT_GET Function"	"GITFN_COMMIT_GET Function"
"GIT_COMMIT_LOG Function"	"GITFN_COMMIT_LOG Function"
"GIT_DELETE_REPO Function"	"GITFN_DEL_REPO Function"
"GIT_DIFF_FILE_IDX Function"	"GITFN_DIFF_IDX_F Function"
"GIT_DIFF_FREE Function"	"GITFN_DIFF_FREE Function"
"GIT_DIFF Function"	"GITFN_DIFF Function"

“GIT_DIFF_GET Function”	“GITFN_DIFF_GET Function”
“GIT_INDEX_ADD Function”	“GITFN_IDX_ADD Function”
“GIT_INDEX_REMOVE Function”	“GITFN_IDX_REMOVE Function”
“GIT_PULL Function”	“GITFN_PULL Function”
“GIT_PUSH Function”	“GITFN_PUSH Function”
“GIT_RESET_FILE Function”	“GITFN_RESET_FILE Function”
“GIT_RESET Function”	“GITFN_RESET Function”
“GIT_STATUS_FREE Function”	“GITFN_STATUSFREE Function”
“GIT_STATUS Function”	“GITFN_STATUS Function”
“GIT_STATUS_GET Function”	“GITFN_STATUS_GET Function”
“GIT_VERSION Function”	“GITFN_VERSION Function”

PART 2

Functions and CALL Routines

Chapter 2
Commonly Used Functions **83**

Chapter 3
Dictionary of Functions and CALL Routines **87**

2

Commonly Used Functions

Most Commonly Used Functions 83

Most Commonly Used Functions

This table lists the most commonly used functions.

Functions	Short Description
"CALL SYMPUTX Routine" on page 366	Assigns a value to a macro variable, and removes both leading and trailing blanks.
"CALL SYMPUT Routine" on page 361	Assigns DATA step information to a macro variable.
"CATS Function" on page 383	Removes leading and trailing blanks, and returns a concatenated character string.
"CATX Function" on page 388	Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.
"COMPRESS Function" on page 467	Returns a character string with specified characters removed from the original string.
"COUNTW Function" on page 495	Counts the number of words in a character string.
"DATDIF Function" on page 513	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.

Functions	Short Description
"DATEPART Function" on page 519	Extracts the date from a SAS datetime value.
"DAY Function" on page 521	Returns the day of the month from a SAS date value.
"DOSUBL Function" on page 561	Imports macro variables from the calling environment, and exports macro variables back to the calling environment.
"EXIST Function" on page 582	Verifies the existence of a SAS library member.
"FILEEXIST Function" on page 609	Verifies the existence of an external file by its physical name.
"INDEX Function" on page 947	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.
"INPUT Function" on page 956	Returns the value that is produced when SAS converts an expression by using the specified informat.
"INT Function" on page 963	Returns the integer value, fuzzed to avoid unexpected floating-point results.
"INTNX Function" on page 1004	Increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value.
"LAG Function" on page 1033	Returns values from a queue.
"LENGTH Function" on page 1050	Returns the length of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string.
"LOWCASE Function" on page 1089	Converts all uppercase single-width English alphabet letters in an argument to lowercase.
"MDY Function" on page 1102	Returns a SAS date value from month, day, and year values.
"MONTH Function" on page 1120	Returns the month from a SAS date value.

Functions	Short Description
"PUT Function" on page 1281	Returns a value using a specified format.
"QTR Function" on page 1292	Returns the quarter of the year from a SAS date value.
"ROUND Function" on page 1348	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.
"SCAN Function" on page 1366	Returns the *n*th word from a character string.
"STRIP Function" on page 1416	Returns a character string with all leading and trailing blanks removed.
"SUBSTR (left of =) Function" on page 1420	Replaces character value contents.
"SUBSTR (right of =) Function" on page 1422	Extracts a substring from an argument.
"SUM Function" on page 1428	Returns the sum of the nonmissing arguments.
"TIME Function" on page 1450	Returns the current time of day as a numeric SAS time value.
"TIMEPART Function" on page 1451	Extracts a time value from a SAS datetime value.
"TRANWRD Function" on page 1465	Replaces all occurrences of a substring in a character string.
"TRANSLATE Function" on page 1458	Replaces specific characters in a character expression.
"TODAY Function" on page 1457	Returns the current date as a numeric SAS date value.
"UPCASE Function" on page 1485	Converts all lowercase single-width English alphabet letters in an argument to uppercase.
"YEAR Function" on page 1573	Returns the year from a SAS date value.

Functions	Short Description
"YRDIF Function" on page 1576	Returns the difference in years between two dates according to specified day count conventions; returns a person's age.
"ZIPCITYDISTANCE Function" on page 1583	Returns the geodetic distance between two ZIP code locations.

3

Dictionary of Functions and CALL Routines

SAS Functions and CALL Routines Documented in Other SAS Publications **100**

SAS Functions and CALL Routines by Category **100**

Dictionary **139**

ABS Function 139
ADDRLONG Function 140
AIRY Function 141
ALLCOMB Function 142
ALLPERM Function 144
ANYALNUM Function 147
ANYALPHA Function 150
ANYCNTRL Function 152
ANYDIGIT Function 154
ANYFIRST Function 156
ANYGRAPH Function 158
ANYLOWER Function 161
ANYNAME Function 163
ANYPRINT Function 165
ANYPUNCT Function 168
ANYSPACE Function 170
ANYUPPER Function 173
ANYXDIGIT Function 175
ARCOS Function 177
ARCOSH Function 177
ARSIN Function 179
ARSINH Function 180
ARTANH Function 181
ATAN Function 182
ATAN2 Function 183
ATTRC Function 185
ATTRN Function 188
BAND Function 193
BETA Function 194
BETAINV Function 195

BHAMMING_32 Function ... 196
BHAMMING_HEX Function ... 198
BLACKCLPRC Function ... 200
BLACKPTPRC Function ... 202
BLKSHCLPRC Function ... 204
BLKSHPTPRC Function ... 206
BLSHIFT Function ... 208
BNOT Function ... 209
BOR Function ... 210
BRSHIFT Function ... 211
BXOR Function ... 212
BYTE Function ... 214
CALL ALLCOMB Routine ... 215
CALL ALLCOMBI Routine ... 219
CALL ALLPERM Routine ... 222
CALL CATS Routine ... 225
CALL CATT Routine ... 228
CALL CATX Routine ... 230
CALL COMPCOST Routine ... 232
CALL EXECUTE Routine ... 235
CALL GRAYCODE Routine ... 236
CALL IS8601_CONVERT Routine ... 241
CALL LABEL Routine ... 253
CALL LEXCOMB Routine ... 255
CALL LEXCOMBI Routine ... 259
CALL LEXPERK Routine ... 262
CALL LEXPERM Routine ... 266
CALL LOGISTIC Routine ... 271
CALL MISSING Routine ... 272
CALL MODULE Routine ... 274
CALL POKELONG Routine ... 279
CALL PRXCHANGE Routine ... 280
CALL PRXDEBUG Routine ... 283
CALL PRXFREE Routine ... 285
CALL PRXNEXT Routine ... 287
CALL PRXPOSN Routine ... 289
CALL PRXSUBSTR Routine ... 292
CALL RANBIN Routine ... 295
CALL RANCAU Routine ... 298
CALL RANCOMB Routine ... 301
CALL RANEXP Routine ... 304
CALL RANGAM Routine ... 306
CALL RANNOR Routine ... 309
CALL RANPERK Routine ... 312
CALL RANPERM Routine ... 314
CALL RANPOI Routine ... 316
CALL RANTBL Routine ... 319
CALL RANTRI Routine ... 321
CALL RANUNI Routine ... 324
CALL SCAN Routine ... 326
CALL SET Routine ... 336
CALL SLEEP Routine ... 338
CALL SOFTMAX Routine ... 340
CALL SORT Routine ... 341

CALL SORTC Routine 342
CALL SORTN Routine 344
CALL STDIZE Routine 345
CALL STREAM Routine 349
CALL STREAMINIT Routine 352
CALL STREAMREWIND Routine 358
CALL SYMPUT Routine 361
CALL SYMPUTX Routine 366
CALL SYSTEM Routine 369
CALL TANH Routine 370
CALL VNAME Routine 371
CALL VNEXT Routine 373
CAT Function 375
CATQ Function 378
CATS Function 383
CATT Function 385
CATX Function 388
CDF Function 392
CDF Bernoulli Distribution Function 394
CDF Beta Distribution Function 395
CDF Binomial Distribution Function 397
CDF Cauchy Distribution Function 398
CDF Chi Square Distribution Function 399
CDF Conway-Maxwell-Poisson Distribution Function 401
CDF Exponential Distribution Function 402
CDF F Distribution Function 403
CDF Gamma Distribution Function 404
CDF Generalized Poisson Distribution Function 405
CDF Geometric Distribution Function 406
CDF Hypergeometric Distribution Function 407
CDF Laplace Distribution Function 409
CDF Logistic Distribution Function 410
CDF Lognormal Distribution Function 411
CDF Negative Binomial Distribution Function 412
CDF Normal Distribution Function 413
CDF Normal Mixture Distribution Function 415
CDF Pareto Distribution Function 416
CDF Poisson Distribution Function 417
CDF T Distribution Function 418
CDF Tweedie Distribution Function 419
CDF Uniform Distribution Function 421
CDF Wald Distribution Function 422
CDF Weibull Distribution Function 423
CEIL Function 424
CEILZ Function 426
CEXIST Function 427
CHAR Function 429
CHOOSEC Function 430
CHOOSEN Function 432
CINV Function 433
CLOSE Function 435
CMISS Function 436
CNONCT Function 437
COALESCE Function 439

COALESCEC Function ... 440
COLLATE Function ... 442
COMB Function ... 445
COMPARE Function ... 447
COMPBL Function ... 450
COMPFUZZ Function ... 452
COMPFUZZ_MISS Function ... 454
COMPGED Function ... 456
COMPLEV Function ... 463
COMPOUND Function ... 466
COMPRESS Function ... 467
COMPSRV_OVAL Function ... 472
COMPSRV_UNQUOTE2 Function ... 474
CONSTANT Function ... 476
CONVX Function ... 482
CONVXP Function ... 483
COS Function ... 485
COSH Function ... 486
COT Function ... 487
COUNT Function ... 488
COUNTC Function ... 491
COUNTW Function ... 495
CSC Function ... 499
CSS Function ... 500
CUMIPMT Function ... 501
CUMPRINC Function ... 502
CUROBS Function ... 504
CV Function ... 505
DACCDB Function ... 506
DACCDBSL Function ... 507
DACCSL Function ... 508
DACCSYD Function ... 510
DACCTAB Function ... 511
DAIRY Function ... 512
DATDIF Function ... 513
DATE Function ... 516
DATEJUL Function ... 518
DATEPART Function ... 519
DATETIME Function ... 520
DAY Function ... 521
DCLOSE Function ... 522
DCREATE Function ... 524
DEPDB Function ... 525
DEPDBSL Function ... 527
DEPSL Function ... 528
DEPSYD Function ... 529
DEPTAB Function ... 531
DEQUOTE Function ... 532
DEVIANCE Function ... 534
DHMS Function ... 540
DIF Function ... 542
DIGAMMA Function ... 544
DIM Function ... 545
DINFO Function ... 548

DIVIDE Function 551
DLGCDIR Function 552
DNUM Function 555
DOPEN Function 556
DOPTNAME Function 558
DOPTNUM Function 560
DOSUBL Function 561
DREAD Function 569
DROPNOTE Function 570
DSNAME Function 572
DUR Function 573
DURP Function 574
EFFRATE Function 576
ENVLEN Function 577
ERF Function 578
ERFC Function 579
EUCLID Function 580
EXIST Function 582
EXP Function 585
EXPM1 Function 586
FACT Function 587
FAPPEND Function 588
FCLOSE Function 591
FCOL Function 592
FCOPY Function 594
FDELETE Function 599
FETCH Function 600
FETCHOBS Function 602
FEXIST Function 605
FGET Function 607
FILEEXIST Function 609
FILENAME Function 611
FILEREF Function 616
FINANCE Function 618
FINANCE ACCRINT Function 623
FINANCE ACCRINTM Function 624
FINANCE AMORDEGRC Function 625
FINANCE AMORLINC Function 627
FINANCE COUPDAYBS Function 628
FINANCE COUPDAYS Function 629
FINANCE COUPDAYSNC Function 630
FINANCE COUPNCD Function 631
FINANCE COUPNUM Function 632
FINANCE COUPPCD Function 633
FINANCE CUMIPMT Function 634
FINANCE CUMPRINC Function 635
FINANCE DB Function 636
FINANCE DDB Function 637
FINANCE DISC Function 639
FINANCE DOLLARDE Function 640
FINANCE DOLLARFR Function 640
FINANCE DURATION Function 641
FINANCE EFFECT Function 642
FINANCE FV Function 643

FINANCE FVSCHEDULE Function 644
FINANCE INTRATE Function 645
FINANCE IPMT Function 646
FINANCE IRR Function 647
FINANCE ISPMT Function 648
FINANCE MDURATION Function 649
FINANCE MIRR Function 650
FINANCE NOMINAL Function 651
FINANCE NPER Function 652
FINANCE NPV Function 653
FINANCE ODDFPRICE Function 654
FINANCE ODDFYIELD Function 655
FINANCE ODDLPRICE Function 657
FINANCE ODDLYIELD Function 658
FINANCE PMT Function 659
FINANCE PPMT Function 661
FINANCE PRICE Function 662
FINANCE PRICEDISC Function 663
FINANCE PRICEMAT Function 664
FINANCE PV Function 665
FINANCE RATE Function 666
FINANCE RECEIVED Function 667
FINANCE SLN Function 668
FINANCE SYD Function 669
FINANCE TBILLEQ Function 670
FINANCE TBILLPRICE Function 671
FINANCE TBILLYIELD Function 672
FINANCE VDB Function 673
FINANCE XIRR Function 674
FINANCE XNPV Function 675
FINANCE YIELD Function 676
FINANCE YIELDDISC Function 677
FINANCE YIELDMAT Function 678
FIND Function 679
FINDC Function 683
FINDW Function 692
FINFO Function 699
FINV Function 702
FIPNAME Function 703
FIPNAMEL Function 705
FIPSTATE Function 707
FIRST Function 709
FLOOR Function 710
FLOORZ Function 712
FMTINFO Function 714
FNONCT Function 716
FNOTE Function 718
FOPEN Function 720
FOPTNAME Function 723
FOPTNUM Function 726
FPOINT Function 728
FPOS Function 731
FPUT Function 733
FREAD Function 735

FREWIND Function ... 737
FRLEN Function ... 739
FSEP Function ... 741
FUZZ Function ... 743
FWRITE Function ... 744
GAMINV Function ... 747
GAMMA Function ... 748
GARKHCLPRC Function ... 749
GARKHPTPRC Function ... 751
GCD Function ... 754
GEODIST Function ... 755
GEOMEAN Function ... 758
GEOMEANZ Function ... 759
GETOPTION Function ... 761
GETVARC Function ... 765
GETVARN Function ... 768
GIT_BRANCH_CHKOUT Function ... 770
GIT_BRANCH_DELETE Function ... 771
GIT_BRANCH_MERGE Function ... 773
GIT_BRANCH_NEW Function ... 775
GIT_CLONE Function ... 776
GIT_COMMIT_FREE Function ... 781
GIT_COMMIT Function ... 782
GIT_COMMIT_GET Function ... 784
GIT_COMMIT_LOG Function ... 787
GIT_DELETE_REPO Function ... 788
GIT_DIFF_FILE_IDX Function ... 789
GIT_DIFF_FREE Function ... 792
GIT_DIFF Function ... 794
GIT_DIFF_GET Function ... 795
GIT_DIFF_TO_FILE Function ... 797
GIT_FETCH Function ... 799
GIT_INDEX_ADD Function ... 803
GIT_INDEX_REMOVE Function ... 805
GIT_INIT_REPO Function ... 806
GIT_PULL Function ... 808
GIT_PUSH Function ... 812
GIT_REBASE Function ... 817
GIT_REBASE_OP Function ... 818
GIT_REF_FREE Function ... 820
GIT_REF_GET Function ... 821
GIT_REF_LIST Function ... 823
GIT_RESET_FILE Function ... 825
GIT_RESET Function ... 826
GIT_SET_URL Function ... 828
GIT_STASH_APPLY Function ... 829
GIT_STASH_DROP Function ... 831
GIT_STASH Function ... 832
GIT_STASH_POP Function ... 834
GIT_STATUS_FREE Function ... 835
GIT_STATUS Function ... 837
GIT_STATUS_GET Function ... 838
GIT_VERSION Function ... 841
GITFN_CLONE Function ... 842

GITFN_CO_BRANCH Function ... 845
GITFN_COMMIT_GET Function ... 847
GITFN_COMMITFREE Function ... 849
GITFN_COMMIT_LOG Function ... 850
GITFN_COMMIT Function ... 852
GITFN_DEL_BRANCH Function ... 854
GITFN_DEL_REPO Function ... 855
GITFN_DIFF_FREE Function ... 856
GITFN_DIFF_GET Function ... 858
GITFN_DIFF_IDX_F Function ... 860
GITFN_DIFF Function ... 862
GITFN_IDX_ADD Function ... 863
GITFN_IDX_REMOVE Function ... 865
GITFN_MRG_BRANCH Function ... 867
GITFN_NEW_BRANCH Function ... 868
GITFN_PULL Function ... 870
GITFN_PUSH Function ... 873
GITFN_RESET_FILE Function ... 876
GITFN_RESET Function ... 877
GITFN_STATUS Function ... 879
GITFN_STATUS_GET Function ... 880
GITFN_STATUSFREE Function ... 883
GITFN_VERSION Function ... 884
GRAYCODE Function ... 886
HARMEAN Function ... 889
HARMEANZ Function ... 891
HASH_FAST_HEX Function ... 892
HASH_XX_HEX Function ... 894
HASHING Function ... 895
HASHING_FILE Function ... 897
HASHING_HMAC Function ... 898
HASHING_HMAC_FILE Function ... 900
HASHING_HMAC_INIT Function ... 901
HASHING_INIT Function ... 903
HASHING_PART Function ... 904
HASHING_TERM Function ... 906
HBOUND Function ... 907
HMS Function ... 910
HOLIDAY Function ... 911
HOLIDAYCK Function ... 916
HOLIDAYCOUNT Function ... 920
HOLIDAYNAME Function ... 922
HOLIDAYNX Function ... 924
HOLIDAYNY Function ... 928
HOLIDAYTEST Function ... 931
HOUR Function ... 934
HTMLDECODE Function ... 936
HTMLENCODE Function ... 937
IBESSEL Function ... 940
IFC Function ... 942
IFN Function ... 944
INDEX Function ... 947
INDEXC Function ... 949
INDEXW Function ... 951

INPUT Function 956
INPUTC Function 959
INPUTN Function 962
INT Function 963
INTCINDEX Function 965
INTCK Function 968
INTCYCLE Function 978
INTFIT Function 983
INTFMT Function 987
INTGET Function 990
INTINDEX Function 993
INTNX Function 1004
INTRR Function 1011
INTSEAS Function 1013
INTSHIFT Function 1016
INTTEST Function 1018
INTZ Function 1020
IORCMSG Function 1022
IPMT Function 1023
IQR Function 1025
IRR Function 1026
JBESSEL Function 1028
JSONPP Function 1029
JULDATE Function 1030
JULDATE7 Function 1031
KURTOSIS Function 1032
LAG Function 1033
LARGEST Function 1042
LBOUND Function 1044
LCM Function 1046
LCOMB Function 1047
LEFT Function 1048
LENGTH Function 1050
LENGTHC Function 1052
LENGTHM Function 1054
LENGTHN Function 1056
LEXCOMB Function 1057
LEXCOMBI Function 1061
LEXPERK Function 1063
LEXPERM Function 1065
LFACT Function 1068
LGAMMA Function 1070
LIBNAME Function 1070
LIBREF Function 1074
LOG Function 1076
LOG10 Function 1076
LOG1PX Function 1077
LOG2 Function 1079
LOGBETA Function 1080
LOGCDF Function 1081
LOGISTIC Function 1083
LOGPDF Function 1084
LOGSDF Function 1087
LOWCASE Function 1089

LPERM Function ... 1090
LPNORM Function ... 1091
MAD Function ... 1093
MARGRCLPRC Function ... 1094
MARGRPTPRC Function ... 1097
MAX Function ... 1099
MD5 Function ... 1101
MDY Function ... 1102
MEAN Function ... 1103
MEDIAN Function ... 1104
MIN Function ... 1106
MINUTE Function ... 1107
MISSING Function ... 1108
MOD Function ... 1110
MODEXIST Function ... 1113
MODULE Function ... 1114
MODULEC Function ... 1116
MODULEN Function ... 1117
MODZ Function ... 1118
MONTH Function ... 1120
MOPEN Function ... 1121
MORT Function ... 1125
MSPLINT Function ... 1127
MVALID Function ... 1130
N Function ... 1133
NETPV Function ... 1134
NLITERAL Function ... 1136
NMISS Function ... 1139
NOMRATE Function ... 1140
NORMAL Function ... 1142
NOTALNUM Function ... 1142
NOTALPHA Function ... 1144
NOTCNTRL Function ... 1147
NOTDIGIT Function ... 1149
NOTE Function ... 1151
NOTFIRST Function ... 1154
NOTGRAPH Function ... 1156
NOTLOWER Function ... 1159
NOTNAME Function ... 1161
NOTPRINT Function ... 1163
NOTPUNCT Function ... 1167
NOTSPACE Function ... 1169
NOTUPPER Function ... 1172
NOTXDIGIT Function ... 1174
NPV Function ... 1176
NVALID Function ... 1178
NWKDOM Function ... 1180
OPEN Function ... 1183
ORDINAL Function ... 1187
PATHNAME Function ... 1188
PCTL Function ... 1190
PDF Function ... 1193
PEEKCLONG Function ... 1214
PEEKLONG Function ... 1216

PERM Function 1217
PMT Function 1219
POINT Function 1221
POISSON Function 1223
PPMT Function 1224
PROBBETA Function 1226
PROBBNML Function 1228
PROBBNRM Function 1229
PROBCHI Function 1231
PROBF Function 1232
PROBGAM Function 1234
PROBHYPR Function 1236
PROBIT Function 1237
PROBMC Function 1239
PROBMED Function 1253
PROBNEGB Function 1254
PROBNORM Function 1256
PROBT Function 1257
PROPCASE Function 1259
PRXCHANGE Function 1261
PRXMATCH Function 1267
PRXPAREN Function 1272
PRXPARSE Function 1274
PRXPOSN Function 1276
PTRLONGADD Function 1280
PUT Function 1281
PUTC Function 1284
PUTN Function 1288
PVP Function 1290
QTR Function 1292
QUANTILE Function 1293
QUOTE Function 1298
RANBIN Function 1301
RANCAU Function 1302
RAND Function 1304
RANEXP Function 1325
RANGAM Function 1327
RANGE Function 1328
RANK Function 1330
RANNOR Function 1331
RANPOI Function 1332
RANTBL Function 1334
RANTRI Function 1335
RANUNI Function 1337
RENAME Function 1338
REPEAT Function 1341
RESOLVE Function 1342
REVERSE Function 1343
REWIND Function 1344
RIGHT Function 1346
RMS Function 1347
ROUND Function 1348
ROUNDE Function 1356
ROUNDZ Function 1359

SAVING Function ... 1362
SAVINGS Function ... 1364
SCAN Function ... 1366
SDF Function ... 1376
SEC Function ... 1380
SECOND Function ... 1381
SHA256 Function ... 1383
SHA256HEX Function ... 1384
SHA256HMACHEX Function ... 1387
SIGN Function ... 1389
SIN Function ... 1390
SINH Function ... 1391
SKEWNESS Function ... 1392
SLEEP Function ... 1393
SMALLEST Function ... 1395
SORT Function ... 1397
SOUNDEX Function ... 1399
SPEDIS Function ... 1400
SQRT Function ... 1403
SQUANTILE Function ... 1404
STD Function ... 1408
STDERR Function ... 1409
STFIPS Function ... 1410
STNAME Function ... 1412
STNAMEL Function ... 1414
STRIP Function ... 1416
SUBPAD Function ... 1418
SUBSTR (left of =) Function ... 1420
SUBSTR (right of =) Function ... 1422
SUBSTRN Function ... 1424
SUM Function ... 1428
SUMABS Function ... 1430
SYMEXIST Function ... 1431
SYMGET Function ... 1432
SYMGLOBL Function ... 1433
SYMLOCAL Function ... 1434
SYSEXIST Function ... 1434
SYSGET Function ... 1436
SYSMSG Function ... 1437
SYSPARM Function ... 1439
SYSPROCESSID Function ... 1440
SYSPROCESSNAME Function ... 1441
SYSPROD Function ... 1443
SYSRC Function ... 1445
SYSTEM Function ... 1446
TAN Function ... 1447
TANH Function ... 1448
TIME Function ... 1450
TIMEPART Function ... 1451
TIMEVALUE Function ... 1452
TINV Function ... 1454
TNONCT Function ... 1455
TODAY Function ... 1457
TRANSLATE Function ... 1458

TRANSTRN Function 1461
TRANWRD Function 1465
TRIGAMMA Function 1469
TRIM Function 1470
TRIMN Function 1472
TRUNC Function 1474
TSLVL Function 1475
TYPEOF Function 1477
TZONEID Function 1478
TZONENAME Function 1479
TZONEOFF Function 1481
TZONES2U Function 1482
TZONEU2S Function 1484
UNIFORM Function 1485
UPCASE Function 1485
URLDECODE Function 1487
URLENCODE Function 1489
USS Function 1491
UUIDGEN Function 1492
VAR Function 1494
VARFMT Function 1495
VARINFMT Function 1497
VARLABEL Function 1499
VARLEN Function 1501
VARNAME Function 1503
VARNUM Function 1504
VARRAY Function 1505
VARRAYX Function 1507
VARTYPE Function 1508
VERIFY Function 1512
VFORMAT Function 1513
VFORMATD Function 1515
VFORMATDX Function 1517
VFORMATN Function 1519
VFORMATNX Function 1521
VFORMATW Function 1523
VFORMATWX Function 1525
VFORMATX Function 1527
VINARRAY Function 1529
VINARRAYX Function 1530
VINFORMAT Function 1532
VINFORMATD Function 1534
VINFORMATDX Function 1536
VINFORMATN Function 1537
VINFORMATNX Function 1539
VINFORMATW Function 1541
VINFORMATWX Function 1543
VINFORMATX Function 1545
VLABEL Function 1547
VLABELX Function 1548
VLENGTH Function 1550
VLENGTHX Function 1552
VNAME Function 1554
VNAMEX Function 1556

VTYPE Function ... 1557
VTYPEX Function ... 1559
VVALUE Function ... 1561
VVALUEX Function ... 1563
WEEK Function ... 1565
WEEKDAY Function ... 1569
WHICHC Function ... 1570
WHICHN Function ... 1571
YEAR Function ... 1573
YIELDP Function ... 1574
YRDIF Function ... 1576
YYQ Function ... 1579
ZIPCITY Function ... 1580
ZIPCITYDISTANCE Function ... 1583
ZIPFIPS Function ... 1584
ZIPNAME Function ... 1586
ZIPNAMEL Function ... 1588
ZIPSTATE Function ... 1590

SAS Functions and CALL Routines Documented in Other SAS Publications

Functions and CALL routines with related subject matter are also documented in the following publications.

- *SAS Data Quality: Language Reference*
- *SAS Logging: Configuration and Programming Reference*
- *SAS Macro Language: Reference*
- *SAS National Language Support (NLS): Reference Guide*

SAS Functions and CALL Routines by Category

The following table lists SAS functions and CALL routines according to category:

Table 3.1 Category and Definition Table

Category	Definition
Array	Returns information about arrays. See Array for a list of functions.
Binary Results	Returns binary results See Binary Results for a list of functions.
Bitwise Logical Operations	Returns the bitwise logical result for an argument. See Bitwise Logical Operations for a list of functions.
Character	Returns information based on character data. See Character for a list of functions.
Character String Matching	Returns information from Perl regular expressions. See Character String Matching for a list of functions.
Combinatorial	Generates combinations and permutations. See Combinatorial for a list of functions.
Compute Server	Performs actions that are passed to the compute server. See Compute Server for a list of functions.
Date and Time	Returns date and time values, including time intervals. See Date and Time for a list of functions.
Descriptive Statistics	Returns statistical values such as mean, median, and standard deviation. See Descriptive Statistics for a list of functions.
Distance	Returns the geodetic distance. See Distance for a list of functions.
External Files	Returns information that is associated with external files. See External Files for a list of functions.
External Routines	Returns a character or numeric value, or calls a routine without any return code. See External Routines for a list of functions.
Financial	Calculates financial values such as interest, periodic payments, depreciation, and prices for European options on stocks. See Financial for a list of functions.

Category	Definition
Git	Supports the libgit2 package for interacting with Git repositories. See Git for a list of functions.
Hyperbolic	Performs hyperbolic calculations such as sinh, cosh, and tanh. See Hyperbolic for a list of functions.
Macro	Assigns a value to a macro variable, returns the value of a macro variable, determines whether a macro variable is global or local in scope, and identifies whether a macro variable exists. See Macro for a list of functions.
Mathematical	Performs mathematical calculations such as factorials, absolute value, fuzzy comparisons, and logarithms. See Mathematical for a list of functions.
Missing Values	Returns missing values and arguments. See Missing Values for a list of functions.
Probability	Returns probability calculations such as from a chi-square or Poisson distribution. See Probability for a list of functions.
Quantile	Returns a quantile from specific distributions. See Quantile for a list of functions.
Random Number	Returns random variates from specific distributions. See Random Number for a list of functions.
Rounding and Truncation	Truncates numeric values and returns numeric values, often using fuzzing or zero fuzzing. See Rounding and Truncation for a list of functions.
SAS File I/O	Returns information about SAS files. See SAS File I/O for a list of functions.
Search	Searches for character or numeric values. See Search for a list of functions.
Sort	Sorts the values of character or numeric arguments. See Sort for a list of functions.
Special	Returns and stores memory addresses, writes a value directly into memory, suspends execution of a program, submits an operating-environment command for execution, returns the

Category	Definition
	value of a SAS or graphics option, specifies formats and informats at run time, returns the system return code, returns the UUID, determines whether a product is licensed, as well as returns other information about SAS processing. See Special for a list of functions.
State and ZIP Code	Returns ZIP codes, FIPS codes, state and city names, postal codes, and the geodetic distance between ZIP codes. See State and ZIP code for a list of functions.
Trigonometric	Returns trigonometric values such as sin, cos, and tan. See Trigonometric for a list of functions.
Variable Control	Assigns variable labels, links SAS data set variables to DATA step or macro variables, and assigns variable names. See Variable Control for a list of functions.
Variable Information	Returns a name, type, length, informat name, label, and other variable information. See Variable Information for a list of functions.
Web Tools	Encodes and decodes a string of data. See Web Tools for a list of functions.

Category	Language Elements	Description
Array	DIM Function (p. 545)	Returns the number of elements in an array.
	HBOUND Function (p. 907)	Returns the upper bound of an array.
	LBOUND Function (p. 1044)	Returns the lower bound of an array.
Binary Results	ADDRLONG Function (p. 140)	Returns the memory address of a variable on a 64-bit platform.
	MD5 Function (p. 1101)	Returns an MD5 message digest as a 16-byte binary string for a message consisting of a character string.
	PEEKCLONG Function (p. 1214)	Stores the contents of a memory address in a character variable on a 64-bit platform.
	PTRLONGADD Function (p. 1280)	Returns the pointer address as a character variable on a 64-bit platform.
	SHA256 Function (p. 1383)	Returns an SHA256 message digest as a 32-byte binary string for a message consisting of a character string.

Category	Language Elements	Description
Bitwise Logical Operations	BAND Function (p. 193)	Returns the bitwise logical AND of 32-bit integer values.
	BHAMMING_32 Function (p. 196)	Returns the bitwise Hamming distance between two 32-bit integer values.
	BLSHIFT Function (p. 208)	Returns the bitwise logical left shift of two 32-bit integer values.
	BNOT Function (p. 209)	Returns the bitwise logical NOT of 32-bit integer values.
	BOR Function (p. 210)	Returns the bitwise logical OR of 32-bit integer values.
	BRSHIFT Function (p. 211)	Returns the bitwise logical right shift of 32-bit integer values.
	BXOR Function (p. 212)	Returns the bitwise logical EXCLUSIVE OR of two 32-bit integer values.
Bitwise Operations	BHAMMING_HEX Function (p. 198)	Returns the bitwise Hamming distance between two hexadecimal bit strings.
Character	ANYALNUM Function (p. 147)	Searches a character string for an alphanumeric character, and returns the first position at which the character is found.
	ANYALPHA Function (p. 150)	Searches a character string for an alphabetic character, and returns the first position at which the character is found.
	ANYCNTRL Function (p. 152)	Searches a character string for a control character, and returns the first position at which that character is found.
	ANYDIGIT Function (p. 154)	Searches a character string for a digit, and returns the first position at which the digit is found.
	ANYFIRST Function (p. 156)	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 position at which that character is found.
	ANYGRAPH Function (p. 158)	Searches a character string for a graphical character, and returns the first position at which that character is found.
	ANYLOWER Function (p. 161)	Searches a character string for a lowercase letter, and returns the first position at which the letter is found.
	ANYNAME Function (p. 163)	Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.
	ANYPUNCT Function (p. 168)	Searches a character string for a punctuation character, and returns the first position at which that character is found.
	ANYSPACE Function (p. 170)	Searches a character string for a whitespace character (blank, horizontal or vertical tab, carriage return, line feed,

Category	Language Elements	Description
		and form feed), and returns the first position at which that character is found.
	ANYUPPER Function (p. 173)	Searches a character string for an uppercase letter, and returns the first position at which the letter is found.
	ANYXDIGIT Function (p. 175)	Searches a character string for a hexadecimal character that represents a digit, and returns the first position at which that character is found.
	BYTE Function (p. 214)	Returns one character in the ASCII collating sequence.
	CALL CATS Routine (p. 225)	Removes leading and trailing blanks, and returns a concatenated character string.
	CALL CATT Routine (p. 228)	Removes trailing blanks and returns a concatenated character string.
	CALL CATX Routine (p. 230)	Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.
	CALL COMPCOST Routine (p. 232)	Sets the costs of operations for later use by the COMPGED function
	CALL MISSING Routine (p. 272)	Assigns missing values to the specified character or numeric variables.
	CALL SCAN Routine (p. 326)	Returns the position and length of the nth word from a character string.
	CAT Function (p. 375)	Does not remove leading or trailing blanks and returns a concatenated character string.
	CATQ Function (p. 378)	Concatenates character and numeric values by using a delimiter to separate items and by adding quotation marks to strings that contain the delimiter.
	CATS Function (p. 383)	Removes leading and trailing blanks, and returns a concatenated character string.
	CATT Function (p. 385)	Removes trailing blanks, and returns a concatenated character string.
	CATX Function (p. 388)	Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.
	CHAR Function (p. 429)	Returns a single character from a specified position in a character string.
	CHOOSEC Function (p. 430)	Returns a character value that represents the results of choosing from a list of arguments.
	CHOOSEN Function (p. 432)	Returns a numeric value that represents the results of choosing from a list of arguments.

Category	Language Elements	Description
	COALESCEC Function (p. 440)	Returns the first nonmissing value from a list of character arguments.
	COLLATE Function (p. 442)	Returns a character string in the ASCII collating sequence.
	COMPARE Function (p. 447)	Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.
	COMPBL Function (p. 450)	Removes multiple blanks from a character string.
	COMPGED Function (p. 456)	Returns the generalized edit distance between two strings.
	COMPLEV Function (p. 463)	Returns the Levenshtein edit distance between two strings.
	COMPRESS Function (p. 467)	Returns a character string with specified characters removed from the original string.
	COUNT Function (p. 488)	Counts the number of times that a specified substring appears within a character string.
	COUNTC Function (p. 491)	Counts the number of characters that appear or do not appear in a list of characters.
	COUNTW Function (p. 495)	Counts the number of words in a character string.
	DEQUOTE Function (p. 532)	Removes matching quotation marks from a character string that begins with a quotation mark, and deletes all characters to the right of the closing quotation mark.
	FIND Function (p. 679)	Searches for a specific substring of characters within a character string.
	FINDC Function (p. 683)	Searches a string for any character in a list of characters.
	FINDW Function (p. 692)	Returns the character position of a word in a string, or returns the number of the word in a string.
	FIRST Function (p. 709)	Returns the first character in a character string.
	HASHING Function (p. 895)	Returns a message digest as a hexadecimal string for a message consisting of a character string, using different methods.
	HASHING_FILE Function (p. 897)	Returns a message digest as a hexadecimal string for a message consisting of an entire file, using different hashing methods.
	HASHING_HMAC Function (p. 898)	Returns a message digest as a hexadecimal string for a character HMAC key value and a message consisting of a character string, using various methods.
	HASHING_HMAC_FILE Function (p. 900)	Returns a message digest as a hexadecimal string for an HMAC key value consisting of a character string and a

Category	Language Elements	Description
		message consisting of an entire file, using different methods.
	HASHING_HMAC_INIT Function (p. 901)	Initializes a running HMAC hash and returns a numeric handle for use with HASHING_PART and HASHING_TERM.
	HASHING_INIT Function (p. 903)	Initializes a running hash and returns a numeric handle for use with HASHING_PART and HASHING_TERM.
	HASHING_PART Function (p. 904)	Provides part of a message for a running hash and returns a numeric value of 1 for a valid handle.
	HASHING_TERM Function (p. 906)	Returns the final digest in hexadecimal representation for the running hash.
	IFC Function (p. 942)	Returns a character value based on whether an expression is true, false, or missing.
	INDEX Function (p. 947)	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. 949)	Searches a character expression for any of the specified characters, and returns the position of that character.
	INDEXW Function (p. 951)	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. 1048)	Left-aligns a character string.
	LENGTH Function (p. 1050)	Returns the length of bytes of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string.
	LENGTHC Function (p. 1052)	Returns the length of a character string, including trailing blanks.
	LENGTHM Function (p. 1054)	Returns the amount of memory, in bytes, that is allocated for a character string.
	LENGTHN Function (p. 1056)	Returns the length of bytes in a character string, excluding trailing blanks.
	LOWCASE Function (p. 1089)	Converts all uppercase single-width English alphabet letters in an argument to lowercase.
	MD5 Function (p. 1101)	Returns an MD5 message digest as a 16-byte binary string for a message consisting of a character string.
	MISSING Function (p. 1108)	Returns a numeric result that indicates whether the argument contains a missing value.
	MVALID Function (p. 1130)	Checks the validity of a character string for use as a SAS member name.

Category	Language Elements	Description
	NLITERAL Function (p. 1136)	Converts a character string that you specify to a SAS name literal.
	NOTALNUM Function (p. 1142)	Searches a character string for a non-alphanumeric character, and returns the first position at which the character is found.
	NOTALPHA Function (p. 1144)	Searches a character string for a nonalphabetic character, and returns the first position at which the character is found.
	NOTCNTRL Function (p. 1147)	Searches a character string for a character that is not a control character, and returns the first position at which that character is found.
	NOTDIGIT Function (p. 1149)	Searches a character string for any character that is not a digit, and returns the first position at which that character is found.
	NOTFIRST Function (p. 1154)	Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.
	NOTGRAPH Function (p. 1156)	Searches a character string for a non-graphical character, and returns the first position at which that character is found.
	NOTLOWER Function (p. 1159)	Searches a character string for a character that is not a lowercase letter, and returns the first position at which that character is found.
	NOTNAME Function (p. 1161)	Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.
	NOTPRINT Function (p. 1163)	Searches a character string for a nonprintable character, and returns the first position at which that character is found.
	NOTPUNCT Function (p. 1167)	Searches a character string for a character that is not a punctuation character, and returns the first position at which that character is found.
	NOTSPACE Function (p. 1169)	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 position at which that character is found.
	NOTUPPER Function (p. 1172)	Searches a character string for a character that is not an uppercase letter, and returns the first position at which that character is found.
	NOTXDIGIT Function (p. 1174)	Searches a character string for a character that is not a hexadecimal character, and returns the first position at which that character is found.

Category	Language Elements	Description
	NVALID Function (p. 1178)	Checks the validity of a character string for use as a SAS variable name.
	PROPCASE Function (p. 1259)	Converts all words in an argument to proper case.
	QUOTE Function (p. 1298)	Adds double quotation marks to a character value.
	RANK Function (p. 1330)	Returns the position of a character in the ASCII collating sequence.
	REPEAT Function (p. 1341)	Returns a character value that consists of the first argument repeated n+1 times.
	REVERSE Function (p. 1343)	Reverses a character string.
	RIGHT Function (p. 1346)	Right aligns a character expression.
	SCAN Function (p. 1366)	Returns the nth word from a character string.
	SHA256 Function (p. 1383)	Returns an SHA256 message digest as a 32-byte binary string for a message consisting of a character string.
	SHA256HEX Function (p. 1384)	Returns the SHA256 digest for a specified message, and the digest is provided in hexadecimal representation.
	SHA256HMACHEX Function (p. 1387)	Returns the result of the message digest of a specified string using the HMAC algorithm.
	SORT Function (p. 1397)	Sorts a list of variables.
	SOUNDEX Function (p. 1399)	Encodes a string to facilitate searching.
	SPEDIS Function (p. 1400)	Determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words.
	STRIP Function (p. 1416)	Returns a character string with all leading and trailing blanks removed.
	SUBPAD Function (p. 1418)	Returns a substring that has a length that you specify, using blank padding if necessary.
	SUBSTR (left of =) Function (p. 1420)	Replaces character value contents.
	SUBSTR (right of =) Function (p. 1422)	Extracts a substring from an argument.
	SUBSTRN Function (p. 1424)	Returns a substring, allowing a result with a length of zero.
	TRANSLATE Function (p. 1458)	Replaces specific characters in a character expression.

Category	Language Elements	Description
	TRANSTRN Function (p. 1461)	Replaces or removes all occurrences of a substring in a character string.
	TRANWRD Function (p. 1465)	Replaces all occurrences of a substring in a character string.
	TRIM Function (p. 1470)	Removes trailing blanks from a character string and returns one blank if the string is missing.
	TRIMN Function (p. 1472)	Removes trailing blanks from character expressions and returns a string with a length of zero if the expression is missing.
	TYPEOF Function (p. 1477)	Returns a value that indicates whether the argument is character or numeric.
	UPCASE Function (p. 1485)	Converts all lowercase single-width English alphabet letters in an argument to uppercase.
	VERIFY Function (p. 1512)	Returns the position of the first character in a string that is not in specified data strings.
Character String Matching	CALL PRXCHANGE Routine (p. 280)	Performs a pattern-matching replacement.
	CALL PRXDEBUG Routine (p. 283)	Enables Perl regular expressions in a DATA step to send debugging output to the SAS log.
	CALL PRXFREE Routine (p. 285)	Frees memory that was allocated for a Perl regular expression.
	CALL PRXNEXT Routine (p. 287)	Returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string.
	CALL PRXPOSN Routine (p. 289)	Returns the start position and length for a capture buffer.
	CALL PRXSUBSTR Routine (p. 292)	Returns the position and length of a substring that matches a pattern.
	PRXCHANGE Function (p. 1261)	Performs a pattern-matching replacement.
	PRXMATCH Function (p. 1267)	Searches for a pattern match and returns the position at which the pattern is found.
	PRXPAREN Function (p. 1272)	Returns the last bracket match for which there is a match in a pattern.
	PRXPARSE Function (p. 1274)	Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.
	PRXPOSN Function (p. 1276)	Returns a character string that contains the value for a capture buffer.

Category	Language Elements	Description
Client Only	ADDRLONG Function (p. 140)	Returns the memory address of a variable on a 64-bit platform.
Combinatorial	ALLCOMB Function (p. 142)	Generates all combinations of the values of n variables taken k at a time in a minimal change order.
	ALLPERM Function (p. 144)	Generates all permutations of the values of several variables in a minimal change order.
	CALL ALLCOMB Routine (p. 215)	Generates all combinations of the values of n variables taken k at a time in a minimal change order.
	CALL ALLCOMBI Routine (p. 219)	Generates all combinations of the indices of n objects taken k at a time in a minimal change order.
	CALL ALLPERM Routine (p. 222)	Generates all permutations of the values of several variables in a minimal change order.
	CALL GRAYCODE Routine (p. 236)	Generates all subsets of n items in a minimal change order.
	CALL LEXCOMB Routine (p. 255)	Generates all distinct combinations of the nonmissing values of n variables taken k at a time in lexicographic order.
	CALL LEXPERK Routine (p. 262)	Generates all distinct permutations of the nonmissing values of n variables taken k at a time in lexicographic order.
	CALL LEXPERM Routine (p. 266)	Generates all distinct permutations of the nonmissing values of several variables in lexicographic order.
	CALL RANCOMB Routine (p. 301)	Permutes the values of the arguments and returns a random combination of k out of n values.
	CALL RANPERK Routine (p. 312)	Permutes the values of the arguments and returns a random permutation of k out of n values.
	CALL RANPERM Routine (p. 314)	Randomly permutes the values of the arguments.
	COMB Function (p. 445)	Computes the number of combinations of n elements taken r at a time.
	GRAYCODE Function (p. 886)	Generates all subsets of n items in a minimal change order.
	LCOMB Function (p. 1047)	Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of n objects taken r at a time.
	LEXCOMB Function (p. 1057)	Generates all distinct combinations of the nonmissing values of n variables taken k at a time in lexicographic order.
	LEXCOMBI Function (p. 1061)	Generates all combinations of the indices of n objects taken k at a time in lexicographic order.

Category	Language Elements	Description
	LEXPERK Function (p. 1063)	Generates all distinct permutations of the nonmissing values of n variables taken k at a time in lexicographic order.
	LEXPERM Function (p. 1065)	Generates all distinct permutations of the nonmissing values of several variables in lexicographic order.
	LFACT Function (p. 1068)	Computes the logarithm of the FACT (factorial) function.
	LPERM Function (p. 1090)	Computes the logarithm of the PERM function, which is the logarithm of the number of permutations of n objects, with the option of including r number of elements.
	PERM Function (p. 1217)	Computes the number of permutations of n items that are taken r at a time.
Compute Server	COMPSRV_OVAL Function (p. 472)	Returns the original, possibly unsafe, value of an input parameter or global macro variable that is passed into the Compute server.
	COMPSRV_UNQUOTE2 Function (p. 474)	Unmasks matched pairs of quotation marks in an input parameter or global macro variable.
Date and Time	CALL IS8601_CONVERT Routine (p. 241)	Converts an ISO 8601 interval to datetime and duration values, or converts datetime and duration values to an ISO 8601 interval.
	DATDIF Function (p. 513)	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.
	DATE Function (p. 516)	Returns the current date as a SAS date value.
	DATEJUL Function (p. 518)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 519)	Extracts the date from a SAS datetime value.
	DATETIME Function (p. 520)	Returns the current date and time of day as a SAS datetime value.
	DAY Function (p. 521)	Returns the day of the month from a SAS date value.
	DHMS Function (p. 540)	Returns a SAS datetime value from date, hour, minute, and second values.
	HMS Function (p. 910)	Returns a SAS time value from hour, minute, and second values.
	HOLIDAY Function (p. 911)	Returns a SAS date value of a specified holiday for a specified year.
	HOLIDAYCK Function (p. 916)	Returns the number of occurrences of the holiday value between date1 and date2.

Category	Language Elements	Description
	HOLIDAYCOUNT Function (p. 920)	Returns the number of holidays defined for a SAS date value.
	HOLIDAYNAME Function (p. 922)	Returns the name of the holiday that corresponds to the SAS date or a blank string if a holiday is not defined for the SAS date.
	HOLIDAYNX Function (p. 924)	Returns the nth occurrence of the holiday relative to the date argument.
	HOLIDAYNY Function (p. 928)	Returns the nth occurrence of the holiday for the year.
	HOLIDAYTEST Function (p. 931)	Returns 1 if the holiday occurs on the SAS date value.
	HOUR Function (p. 934)	Returns the hour from a SAS time or datetime value.
	INTCINDEX Function (p. 965)	Returns the cycle index when a date, time, or datetime interval and value are specified.
	INTCK Function (p. 968)	Returns the number of interval boundaries of a given kind that lie between two dates, times, or datetime values.
	INTCYCLE Function (p. 978)	Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.
	INTFIT Function (p. 983)	Returns a time interval that is aligned between two dates.
	INTFMT Function (p. 987)	Returns a recommended SAS format when a date, time, or datetime interval is specified.
	INTGET Function (p. 990)	Returns a time interval based on three date or datetime values.
	INTINDEX Function (p. 993)	Returns the seasonal index when a date, time, or datetime interval and value are specified.
	INTNX Function (p. 1004)	Increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value.
	INTSEAS Function (p. 1013)	Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.
	INTSHIFT Function (p. 1016)	Returns the shift interval that corresponds to the base interval.
	INTTEST Function (p. 1018)	Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.
	JULDATE Function (p. 1030)	Returns the Julian date from a SAS date value.
	JULDATE7 Function (p. 1031)	Returns a seven-digit Julian date from a SAS date value.

Category	Language Elements	Description
	MDY Function (p. 1102)	Returns a SAS date value from month, day, and year values.
	MINUTE Function (p. 1107)	Returns the minute from a SAS time or datetime value.
	MONTH Function (p. 1120)	Returns the month from a SAS date value.
	NWKDOM Function (p. 1180)	Returns the date for the nth occurrence of a weekday for the specified month and year.
	QTR Function (p. 1292)	Returns the quarter of the year from a SAS date value.
	SECOND Function (p. 1381)	Returns the seconds and milliseconds from a SAS time or datetime value.
	TIME Function (p. 1450)	Returns the current time of day as a numeric SAS time value.
	TIMEPART Function (p. 1451)	Extracts a time value from a SAS datetime value.
	TODAY Function (p. 1457)	Returns the current date as a numeric SAS date value.
	TZONEID Function (p. 1478)	Returns the current time zone ID.
	TZONENAME Function (p. 1479)	Returns the current standard or daylight savings time, time zone name.
	TZONEOFF Function (p. 1481)	Returns the user time zone offset.
	TZONES2U Function (p. 1482)	Converts a SAS date time value to a UTC date time value.
	TZONEU2S Function (p. 1484)	Converts a UTC date time value to a SAS date time value.
	WEEK Function (p. 1565)	Returns the week-number value.
	WEEKDAY Function (p. 1569)	From a SAS date value, returns an integer that corresponds to the day of the week.
	YEAR Function (p. 1573)	Returns the year from a SAS date value.
	YRDIF Function (p. 1576)	Returns the difference in years between two dates according to specified day count conventions; returns a person's age.
	YYQ Function (p. 1579)	Returns a SAS date value from year and quarter year values.
Descriptive Statistics	CMISS Function (p. 436)	Counts the number of missing arguments.
	CSS Function (p. 500)	Returns the corrected sum of squares.
	CV Function (p. 505)	Returns the coefficient of variation.
	EUCLID Function (p. 580)	Returns the Euclidean norm of the nonmissing arguments.

Category	Language Elements	Description
	GEOMEAN Function (p. 758)	Returns the geometric mean.
	GEOMEANZ Function (p. 759)	Returns the geometric mean, using zero fuzzing.
	HARMEAN Function (p. 889)	Returns the harmonic mean.
	HARMEANZ Function (p. 891)	Returns the harmonic mean, using zero fuzzing.
	IQR Function (p. 1025)	Returns the interquartile range.
	KURTOSIS Function (p. 1032)	Returns the kurtosis.
	LARGEST Function (p. 1042)	Returns the kth largest nonmissing value.
	LPNORM Function (p. 1091)	Returns the Lp norm of the second argument and subsequent nonmissing arguments.
	MAD Function (p. 1093)	Returns the median absolute deviation from the median.
	MAX Function (p. 1099)	Returns the largest value.
	MEAN Function (p. 1103)	Returns the arithmetic mean (average).
	MEDIAN Function (p. 1104)	Returns the median value.
	MIN Function (p. 1106)	Returns the smallest value.
	MISSING Function (p. 1108)	Returns a numeric result that indicates whether the argument contains a missing value.
	N Function (p. 1133)	Returns the number of nonmissing numeric values.
	NMISS Function (p. 1139)	Returns the number of missing numeric values.
	ORDINAL Function (p. 1187)	Returns the kth smallest of the missing and nonmissing values.
	PCTL Function (p. 1190)	Returns the percentile that corresponds to the percentage.
	RANGE Function (p. 1328)	Returns the range of the nonmissing values.
	RMS Function (p. 1347)	Returns the root mean square of the nonmissing arguments.
	SKEWNESS Function (p. 1392)	Returns the skewness of the nonmissing arguments.
	SMALLEST Function (p. 1395)	Returns the kth smallest nonmissing value.
	STD Function (p. 1408)	Returns the standard deviation of the nonmissing arguments.

Category	Language Elements	Description
	STDERR Function (p. 1409)	Returns the standard error of the mean of the nonmissing arguments.
	SUM Function (p. 1428)	Returns the sum of the nonmissing arguments.
	SUMABS Function (p. 1430)	Returns the sum of the absolute values of the nonmissing arguments.
	USS Function (p. 1491)	Returns the uncorrected sum of squares of the nonmissing arguments.
	VAR Function (p. 1494)	Returns the variance of the nonmissing arguments.
Distance	GEODIST Function (p. 755)	Returns the geodetic distance between two latitude and longitude coordinates.
	ZIPCITYDISTANCE Function (p. 1583)	Returns the geodetic distance between two ZIP code locations.
External Files	DCLOSE Function (p. 522)	Closes a directory that was opened by the DOPEN function.
	DCREATE Function (p. 524)	Returns the complete pathname of a new, external directory.
	DINFO Function (p. 548)	Returns information about a directory.
	DNUM Function (p. 555)	Returns the number of members in a directory.
	DOPEN Function (p. 556)	Opens a directory, and returns a directory identifier value.
	DOPTNAME Function (p. 558)	Returns directory attribute information.
	DOPTNUM Function (p. 560)	Returns the number of information items that are available for a directory.
	DREAD Function (p. 569)	Returns the name of a directory member.
	DROPNOTE Function (p. 570)	Deletes a note marker from a SAS data set or an external file.
	FAPPEND Function (p. 588)	Appends the current record to the end of an external file.
	FCLOSE Function (p. 591)	Closes an external file, directory, or directory member.
	FCOL Function (p. 592)	Returns the current column position in the File Data Buffer (FDB).
	FCOPY Function (p. 594)	Copies records from one fileref to another fileref, and returns a value that indicates whether the records were successfully copied.
	FDELETE Function (p. 599)	Deletes an external file or an empty directory.

Category	Language Elements	Description
	FEXIST Function (p. 605)	Verifies the existence of an external file that is associated with a fileref.
	FGET Function (p. 607)	Copies data from the File Data Buffer (FDB) into a variable.
	FILEEXIST Function (p. 609)	Verifies the existence of an external file by its physical name.
	FILENAME Function (p. 611)	Assigns or deassigns a fileref to an external file, directory, or output device.
	FILEREF Function (p. 616)	Verifies whether a fileref has been assigned for the current SAS session.
	FINFO Function (p. 699)	Returns the value of a file information item.
	FNOTE Function (p. 718)	Identifies the last record that was read, and returns a value that the FPOINT function can use.
	FOPEN Function (p. 720)	Opens an external file and returns a file identifier value.
	FOPTNAME Function (p. 723)	Returns the name of an item of information about an external file.
	FOPTNUM Function (p. 726)	Returns the number of information items, such as filename or record length, that are available for an external file.
	FPOINT Function (p. 728)	Positions the read pointer on the next record to be read.
	FPOS Function (p. 731)	Sets the position of the column pointer in the File Data Buffer (FDB).
	FPUT Function (p. 733)	Moves data to the File Data Buffer (FDB) of an external file, starting at the FDB's current column position.
	FREAD Function (p. 735)	Reads a record from an external file into the File Data Buffer (FDB).
	FREWIND Function (p. 737)	Positions the file pointer to the start of the file.
	FRLEN Function (p. 739)	Returns the size of the last record that was read, or, if the file is opened for output, returns the current record size.
	FSEP Function (p. 741)	Sets the token delimiters for the FGET function.
	FWRITE Function (p. 744)	Writes a record to an external file.
	MOPEN Function (p. 1121)	Opens a file by directory ID and member name, and returns either the file identifier or a 0.
	PATHNAME Function (p. 1188)	Returns the physical name of an external file or a SAS library, or returns a blank.

Category	Language Elements	Description
	RENAME Function (p. 1338)	Renames a member of a SAS library, an entry in a SAS catalog, an external file, or a directory.
	SYSMSG Function (p. 1437)	Returns error or warning message text from processing the last data set or external file function.
	SYSRC Function (p. 1445)	Returns a system error number.
External Routines	CALL MODULE Routine (p. 274)	Calls an external routine without any return code.
	MODEXIST Function (p. 1113)	Determines whether a software image exists in the version of SAS that you have installed.
	MODULE Function (p. 1114)	Calls a specific routine or module that resides in an external dynamic link library (DLL).
	MODULEC Function (p. 1116)	Calls an external routine and returns a character value.
	MODULEN Function (p. 1117)	Calls an external routine and returns a numeric value.
Financial	BLACKCLPRC Function (p. 200)	Calculates call prices for European options on futures, based on the Black model.
	BLACKPTPRC Function (p. 202)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 204)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	BLKSHPTPRC Function (p. 206)	Calculates put prices for European options on stocks, based on the Black-Scholes model.
	COMPOUND Function (p. 466)	Returns compound interest parameters.
	CONVX Function (p. 482)	Returns the convexity for an enumerated cash flow.
	CONVXP Function (p. 483)	Returns the convexity for a periodic cash flow stream such as a bond.
	CUMIPMT Function (p. 501)	Returns the cumulative interest paid on a loan between the start and end periods.
	CUMPRINC Function (p. 502)	Returns the cumulative principal paid on a loan between the start and end periods.
	DACCDB Function (p. 506)	Returns the accumulated declining balance depreciation.
	DACCDBSL Function (p. 507)	Returns the accumulated declining balance with conversion to a straight-line depreciation.
	DACCSL Function (p. 508)	Returns the accumulated straight-line depreciation.

Category	Language Elements	Description
	DACCSYD Function (p. 510)	Returns the accumulated sum-of-years-digits depreciation.
	DACCTAB Function (p. 511)	Returns the accumulated depreciation from specified tables.
	DEPDB Function (p. 525)	Returns the declining balance depreciation.
	DEPDBSL Function (p. 527)	Returns the declining balance with conversion to a straight-line depreciation.
	DEPSL Function (p. 528)	Returns the straight-line depreciation.
	DEPSYD Function (p. 529)	Returns the sum-of-years-digits depreciation.
	DEPTAB Function (p. 531)	Returns the depreciation from specified tables.
	DUR Function (p. 573)	Returns the modified duration for an enumerated cash flow.
	DURP Function (p. 574)	Returns the modified duration for a periodic cash flow stream, such as a bond.
	EFFRATE Function (p. 576)	Returns the effective annual interest rate.
	FINANCE Function (p. 618)	Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
	FINANCE ACCRINT Function (p. 623)	Computes the accrued interest for a security that pays periodic interest.
	FINANCE ACCRINTM Function (p. 624)	Computes the accrued interest for a security that pays interest at maturity.
	FINANCE AMORDEGRC Function (p. 625)	Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
	FINANCE AMORLINC Function (p. 627)	Computes the depreciation for each accounting period.
	FINANCE COUPDAYBS Function (p. 628)	Computes the number of days from the beginning of the coupon period to the settlement date.
	FINANCE COUPDAYS Function (p. 629)	Computes the number of days in the coupon period that contains the settlement date.
	FINANCE COUPDAYSNC Function (p. 630)	
	FINANCE COUPNCD Function (p. 631)	
	FINANCE COUPNUM Function (p. 632)	Computes the number of coupons that are payable between the settlement date and the maturity date.

Category	Language Elements	Description
	FINANCE COUPPCD Function (p. 633)	Computes the previous coupon date before the settlement date.
	FINANCE CUMIPMT Function (p. 634)	Computes the cumulative interest paid between two periods.
	FINANCE CUMPRINC Function (p. 635)	Computes the cumulative principal that is paid on a loan between two periods.
	FINANCE DB Function (p. 636)	Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
	FINANCE DDB Function (p. 637)	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. 639)	Computes the discount rate for a security.
	FINANCE DOLLARDE Function (p. 640)	Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
	FINANCE DOLLARFR Function (p. 640)	Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
	FINANCE DURATION Function (p. 641)	Computes the annual duration of a security with periodic interest payments.
	FINANCE EFFECT Function (p. 642)	Computes the effective annual interest rate.
	FINANCE FV Function (p. 643)	Computes the future value of an investment.
	FINANCE FVSCHEDULE Function (p. 644)	Computes the future value of the initial principal after applying a series of compound interest rates.
	FINANCE INTRATE Function (p. 645)	Computes the interest rate for a fully invested security.
	FINANCE IPMT Function (p. 646)	Computes the interest payment for an investment for a specified period.
	FINANCE IRR Function (p. 647)	Computes the internal rate of return for a series of cash flows.
	FINANCE ISPMT Function (p. 648)	Calculates the interest paid during a specific period of an investment.
	FINANCE MDURATION Function (p. 649)	Computes the Macaulay modified duration for a security with an assumed face value of $100.
	FINANCE MIRR Function (p. 650)	Computes the internal rate of return where positive and negative cash flows are financed at different rates.

Category	Language Elements	Description
	FINANCE NOMINAL Function (p. 651)	Computes the annual nominal interest rates.
	FINANCE NPER Function (p. 652)	Computes the number of periods for an investment.
	FINANCE NPV Function (p. 653)	Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
	FINANCE ODDFPRICE Function (p. 654)	Computes the price of a security per $100 face value with an odd first period.
	FINANCE ODDFYIELD Function (p. 655)	Computes the yield of a security with an odd first period.
	FINANCE ODDLPRICE Function (p. 657)	Computes the price of a security per $100 face value with an odd last period.
	FINANCE ODDLYIELD Function (p. 658)	Computes the yield of a security with an odd last period.
	FINANCE PMT Function (p. 659)	Computes the periodic payment of an annuity.
	FINANCE PPMT Function (p. 661)	Computes the payment on the principal for an investment for a specified period.
	FINANCE PRICE Function (p. 662)	Computes the price of a security per $100 face value that pays periodic interest.
	FINANCE PRICEDISC Function (p. 663)	Computes the price of a discounted security per $100 face value.
	FINANCE PRICEMAT Function (p. 664)	Computes the price of a security per $100 face value that pays interest at maturity.
	FINANCE PV Function (p. 665)	Computes the present value of an investment.
	FINANCE RATE Function (p. 666)	Computes the interest rate per period of an annuity.
	FINANCE RECEIVED Function (p. 667)	Computes the amount that is received at maturity for a fully invested security.
	FINANCE SLN Function (p. 668)	Computes the straight-line depreciation of an asset for one period.
	FINANCE SYD Function (p. 669)	Computes the sum-of-years digits depreciation of an asset for a specified period.
	FINANCE TBILLEQ Function (p. 670)	Computes the bond-equivalent yield for a treasury bill.

Category	Language Elements	Description
	FINANCE TBILLPRICE Function (p. 671)	Computes the price of a treasury bill per $100 face value.
	FINANCE TBILLYIELD Function (p. 672)	Computes the yield for a treasury bill.
	FINANCE VDB Function (p. 673)	Computes the depreciation of an asset for a specified or partial period by using a declining balance method.
	FINANCE XIRR Function (p. 674)	Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
	FINANCE XNPV Function (p. 675)	Computes the net present value for a schedule of cash flows that is not necessarily periodic.
	FINANCE YIELD Function (p. 676)	Computes the yield on a security that pays periodic interest.
	FINANCE YIELDDISC Function (p. 677)	Computes the annual yield for a discounted security (for example, a treasury bill).
	FINANCE YIELDMAT Function (p. 678)	Computes the annual yield of a security that pays interest at maturity.
	GARKHCLPRC Function (p. 749)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
	GARKHPTPRC Function (p. 751)	Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.
	INTRR Function (p. 1011)	Returns the internal rate of return as a fraction.
	IPMT Function (p. 1023)	Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.
	IRR Function (p. 1026)	Returns the internal rate of return as a percentage.
	MARGRCLPRC Function (p. 1094)	Calculates call prices for European options on stocks, based on the Margrabe model.
	MARGRPTPRC Function (p. 1097)	Calculates put prices for European options on stocks, based on the Margrabe model.
	MORT Function (p. 1125)	Returns amortization parameters.
	NETPV Function (p. 1134)	Returns the net present value as a percent.
	NOMRATE Function (p. 1140)	Returns the nominal annual interest rate.
	NPV Function (p. 1176)	Returns the net present value with the rate expressed as a percentage.

Category	Language Elements	Description
	PMT Function (p. 1219)	Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.
	PPMT Function (p. 1224)	Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.
	PVP Function (p. 1290)	Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.
	SAVING Function (p. 1362)	Returns the future value of a periodic saving.
	SAVINGS Function (p. 1364)	Returns the balance of a periodic savings by using variable interest rates.
	TIMEVALUE Function (p. 1452)	Returns the equivalent of a reference amount at a base date by using variable interest rates.
	YIELDP Function (p. 1574)	Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.
Git	GIT_BRANCH_CHKOUT Function (p. 770)	Check out a branch in a Git repository.
	GIT_BRANCH_DELETE Function (p. 771)	Deletes a Git branch in the repository.
	GIT_BRANCH_MERGE Function (p. 773)	Merges a Git branch into the currently checked-out branch.
	GIT_BRANCH_NEW Function (p. 775)	Creates a Git branch.
	GIT_CLONE Function (p. 776)	Clones a Git repository into a directory on the SAS server.
	GIT_COMMIT_FREE Function (p. 781)	Clears the commit record object that was created by GIT_COMMIT_LOG for the specified repository.
	GIT_COMMIT Function (p. 782)	Commits staged files to the local repository.
	GIT_COMMIT_GET Function (p. 784)	Returns the specified attribute of the nth commit object that is associated with the local repository.
	GIT_COMMIT_LOG Function (p. 787)	Returns the number of commit objects that are associated with the local repository.
	GIT_DELETE_REPO Function (p. 788)	deletes a local Git repository and all content within the repository.
	GIT_DIFF_FILE_IDX Function (p. 789)	Returns the diff for a file in the index.

Category	Language Elements	Description
	GIT_DIFF_FREE Function (p. 792)	Clears the diff record object associated with a local repository.
	GIT_DIFF Function (p. 794)	Returns the number of diffs between two commits in the local repository and creates a diff record object for the local repository.
	GIT_DIFF_GET Function (p. 795)	Returns the specified attribute of the nth diff object in the local repository.
	GIT_DIFF_TO_FILE Function (p. 797)	Writes the diff content to a file reference.
	GIT_INDEX_ADD Function (p. 803)	Stages 1 to n number of files to commit to the local repository.
	GIT_INDEX_REMOVE Function (p. 805)	Unstages 1 to n number of files to commit to the local repository.
	GIT_INIT_REPO Function (p. 806)	Initializes a new local Git repository.
	GIT_PULL Function (p. 808)	Pulls changes from the remote repository into the local repository.
	GIT_PUSH Function (p. 812)	Pushes the committed files in the local repository to the remote repository.
	GIT_REBASE Function (p. 817)	Rebases your current branch to a specified commit ID.
	GIT_REBASE_OP Function (p. 818)	Used to execute rebase operations when a conflict occurs.
	GIT_REF_FREE Function (p. 820)	Clears the reference objects that are associated with the local repository.
	GIT_REF_GET Function (p. 821)	Returns the specified attribute of a reference object in the local repository.
	GIT_REF_LIST Function (p. 823)	Returns the list of reference objects that are associated with the local repository.
	GIT_RESET_FILE Function (p. 825)	Resets a file in the index to the local repository version.
	GIT_RESET Function (p. 826)	Resets the local repository to a specified commit.
	GIT_SET_URL Function (p. 828)	Sets the remote repository URL for a local repository.
	GIT_STASH_APPLY Function (p. 829)	Applies file changes that are stored in a stash to the local repository.

Category	Language Elements	Description
	GIT_STASH_DROP Function (p. 831)	Drops the contents of the stash stack at the specified index.
	GIT_STASH Function (p. 832)	Stashes file changes that have not been committed.
	GIT_STASH_POP Function (p. 834)	Applies the changes that are stored in the stash, and then drops the contents of the stash.
	GIT_STATUS_FREE Function (p. 835)	Clears the status record object that was created by GIT_STATUS for the specified repository.
	GIT_STATUS Function (p. 837)	Returns the status objects for files in the local repository and creates a status record.
	GIT_STATUS_GET Function (p. 838)	Returns the specified attribute of the nth status object returned from GITFN_STATUS() in the local repository.
	GIT_VERSION Function (p. 841)	Specifies whether libgit2 is available and if available, specifies the version that is being used.
	GITFN_CLONE Function (p. 842)	Clones a Git repository into a directory on the SAS server.
	GITFN_CO_BRANCH Function (p. 845)	Check out a branch in a Git repository.
	GITFN_COMMIT_GET Function (p. 847)	Returns the specified attribute of the nth commit object that is associated with the local repository.
	GITFN_COMMITFREE Function (p. 849)	Clears the commit record object that was created by GITFN_COMMIT_LOG for the specified repository.
	GITFN_COMMIT_LOG Function (p. 850)	Returns the number of commit objects that are associated with the local repository.
	GITFN_COMMIT Function (p. 852)	Commits staged files to the local repository.
	GITFN_DEL_BRANCH Function (p. 854)	Deletes a Git branch in the repository.
	GITFN_DEL_REPO Function (p. 855)	Deletes a local Git repository and its contents.
	GITFN_DIFF_FREE Function (p. 856)	Clears the diff record object associated with a local repository.
	GITFN_DIFF_GET Function (p. 858)	Returns the specified attribute of the nth diff object in the local repository.
	GITFN_DIFF_IDX_F Function (p. 860)	Returns the changes for a file in the index.

Category	Language Elements	Description
	GITFN_DIFF Function (p. 862)	Returns the number of diffs between two commits in the local repository and creates a diff record object for the local repository.
	GITFN_IDX_ADD Function (p. 863)	Stages 1 to n number of files to commit to the local repository.
	GITFN_IDX_REMOVE Function (p. 865)	Unstages 1 to n number of files to commit to the local repository.
	GITFN_MRG_BRANCH Function (p. 867)	Merges a Git branch into the currently checked-out branch.
	GITFN_NEW_BRANCH Function (p. 868)	Creates a Git branch.
	GITFN_PULL Function (p. 870)	Pulls changes from the remote repository into the local repository.
	GITFN_PUSH Function (p. 873)	Pushes the committed files in the local repository to the remote repository.
	GITFN_RESET_FILE Function (p. 876)	Resets a file in the index to the local repository version.
	GITFN_RESET Function (p. 877)	Resets the local repository to a specified commit.
	GITFN_STATUS Function (p. 879)	Returns the status objects for files in the local repository and creates a status record.
	GITFN_STATUS_GET Function (p. 880)	Returns the specified attribute of the nth status object returned from GITFN_STATUS() in the local repository.
	GITFN_STATUSFREE Function (p. 883)	Clears the status record object that was created by GITFN_STATUS for the specified repository.
	GITFN_VERSION Function (p. 884)	Specifies whether libgit2 is available and if available, specifies the version that is being used.
Hyperbolic	ARCOSH Function (p. 177)	Returns the inverse hyperbolic cosine.
	ARSINH Function (p. 180)	Returns the inverse hyperbolic sine.
	ARTANH Function (p. 181)	Returns the inverse hyperbolic tangent.
	COSH Function (p. 486)	Returns the hyperbolic cosine.
	SINH Function (p. 1391)	Returns the hyperbolic sine.
	TANH Function (p. 1448)	Returns the hyperbolic tangent.
Macro	CALL EXECUTE Routine (p. 235)	Resolves the argument, and issues the resolved value for execution at the next step boundary.

Category	Language Elements	Description
	CALL SYMPUT Routine (p. 361)	Assigns a value produced in a DATA step to a macro variable.
	CALL SYMPUTX Routine (p. 366)	Assigns a value to a macro variable, and removes both leading and trailing blanks.
	DOSUBL Function (p. 561)	Enables the immediate execution of SAS code after a text string is passed.
	RESOLVE Function (p. 1342)	Returns the resolved value of the argument after the argument has been processed by the macro facility.
	SYMEXIST Function (p. 1431)	Returns an indication of the existence of a macro variable.
	SYMGET Function (p. 1432)	Returns the value of a macro variable during DATA step execution.
	SYMGLOBL Function (p. 1433)	Returns an indication of whether a macro variable is in global scope to the DATA step during DATA step execution.
	SYMLOCAL Function (p. 1434)	Returns an indication of whether a macro variable is in local scope to the DATA step during DATA step execution.
Mathematical	ABS Function (p. 139)	Returns the absolute value.
	AIRY Function (p. 141)	Returns the value of a differential equation.
	BETA Function (p. 194)	Returns the value of the beta function.
	CALL LOGISTIC Routine (p. 271)	Applies the logistic function to each argument.
	CALL SOFTMAX Routine (p. 340)	Returns the softmax value.
	CALL STDIZE Routine (p. 345)	Standardizes the values of one or more variables.
	CALL TANH Routine (p. 370)	Returns the hyperbolic tangent.
	CNONCT Function (p. 437)	Returns the noncentrality parameter from a chi-square distribution.
	COALESCE Function (p. 439)	Returns the first nonmissing value from a list of numeric arguments.
	COMPFUZZ Function (p. 452)	Performs a fuzzy comparison of two numeric values.
	COMPFUZZ_MISS Function (p. 454)	Performs comparisons of missing numeric values and fuzzy comparisons of nonmissing numeric values.
	CONSTANT Function (p. 476)	Computes machine and mathematical constants.

Category	Language Elements	Description
	DAIRY Function (p. 512)	Returns the derivative of the AIRY function.
	DEVIANCE Function (p. 534)	Returns the deviance based on a probability distribution.
	DIGAMMA Function (p. 544)	Returns the value of the digamma function.
	DIVIDE Function (p. 551)	Returns the result of a division that handles special missing values for ODS output.
	ERF Function (p. 578)	Returns the value of the (normal) error function.
	ERFC Function (p. 579)	Returns the value of the complementary (normal) error function.
	EXP Function (p. 585)	Returns the value of the exponential function.
	EXPM1 Function (p. 586)	Returns 1 less than the exponential function of the argument.
	FACT Function (p. 587)	Computes a factorial.
	FNONCT Function (p. 716)	Returns the value of the noncentrality parameter of an F distribution.
	GAMMA Function (p. 748)	Returns the value of the gamma function.
	GCD Function (p. 754)	Returns the greatest common divisor for one or more integers.
	IBESSEL Function (p. 940)	Returns the value of the modified Bessel function.
	IFN Function (p. 944)	Returns a numeric value based on whether an expression is true, false, or missing.
	JBESSEL Function (p. 1028)	Returns the value of the Bessel function.
	LCM Function (p. 1046)	Returns the least common multiple.
	LGAMMA Function (p. 1070)	Returns the natural logarithm of the Gamma function.
	LOG Function (p. 1076)	Returns the natural (base e) logarithm.
	LOG10 Function (p. 1076)	Returns the logarithm to the base 10.
	LOG1PX Function (p. 1077)	Returns the log of 1 plus the argument.
	LOG2 Function (p. 1079)	Returns the logarithm to the base 2.
	LOGBETA Function (p. 1080)	Returns the logarithm of the beta function.
	LOGISTIC Function (p. 1083)	Returns the logistic transformation of the argument.

Category	Language Elements	Description
	MOD Function (p. 1110)	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. 1118)	Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.
	MSPLINT Function (p. 1127)	Returns the ordinate of a monotonicity-preserving interpolating spline.
	SIGN Function (p. 1389)	Returns the sign of a value.
	SQRT Function (p. 1403)	Returns the square root of a value.
	TNONCT Function (p. 1455)	Returns the value of the noncentrality parameter from the Student's t distribution.
	TRIGAMMA Function (p. 1469)	Returns the value of the trigamma function.
Missing Values	CALL MISSING Routine (p. 272)	Assigns missing values to the specified character or numeric variables.
	CMISS Function (p. 436)	Counts the number of missing arguments.
	COALESCE Function (p. 439)	Returns the first nonmissing value from a list of numeric arguments.
	COALESCEC Function (p. 440)	Returns the first nonmissing value from a list of character arguments.
	MISSING Function (p. 1108)	Returns a numeric result that indicates whether the argument contains a missing value.
	N Function (p. 1133)	Returns the number of nonmissing numeric values.
	NMISS Function (p. 1139)	Returns the number of missing numeric values.
Probability	CDF Function (p. 392)	Returns a value from a cumulative probability distribution.
	CDF Bernoulli Distribution Function (p. 394)	Returns a value from the Bernoulli cumulative probability distribution.
	CDF Beta Distribution Function (p. 395)	Returns a value from the beta cumulative probability distribution.
	CDF Binomial Distribution Function (p. 397)	Returns a value from the binomial cumulative probability distribution.
	CDF Cauchy Distribution Function (p. 398)	Returns a value from the Cauchy cumulative probability distribution.
	CDF Chi Square Distribution Function (p. 399)	Returns a value from the Chi-square cumulative probability distribution.

Category	Language Elements	Description
	CDF Conway-Maxwell-Poisson Distribution Function (p. 401)	Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.
	CDF Exponential Distribution Function (p. 402)	Returns a value from the Exponential probability distribution.
	CDF F Distribution Function (p. 403)	Returns a value from the F distribution.
	LOGCDF Function (p. 1081)	Returns the logarithm of a left cumulative distribution function.
	LOGPDF Function (p. 1084)	Returns the logarithm of a probability density (mass) function.
	LOGSDF Function (p. 1087)	Returns the logarithm of a survival function.
	PDF Function (p. 1193)	Returns a value from a probability density (mass) distribution.
	POISSON Function (p. 1223)	Returns the probability from a Poisson distribution.
	PROBBETA Function (p. 1226)	Returns the probability from a beta distribution.
	PROBBNML Function (p. 1228)	Returns the probability from a binomial distribution.
	PROBBNRM Function (p. 1229)	Returns a probability from a bivariate normal distribution.
	PROBCHI Function (p. 1231)	Returns the probability from a chi-square distribution.
	PROBF Function (p. 1232)	Returns the probability from an F distribution.
	PROBGAM Function (p. 1234)	Returns the probability from a gamma distribution.
	PROBHYPR Function (p. 1236)	Returns the probability from a hypergeometric distribution.
	PROBMC Function (p. 1239)	Returns a probability or a quantile from various distributions for multiple comparisons of means.
	PROBMED Function (p. 1253)	Computes cumulative probabilities for the sample median.
	PROBNEGB Function (p. 1254)	Returns the probability from a negative binomial distribution.
	PROBNORM Function (p. 1256)	Returns the probability from the standard normal distribution.
	PROBT Function (p. 1257)	Returns the probability from a t distribution.

Category	Language Elements	Description
	SDF Function (p. 1376)	Returns a survival function.
Quantile	BETAINV Function (p. 195)	Returns a quantile from the beta distribution.
	CINV Function (p. 433)	Returns a quantile from the chi-square distribution.
	FINV Function (p. 702)	Returns a quantile from the F distribution.
	GAMINV Function (p. 747)	Returns a quantile from the gamma distribution.
	PROBIT Function (p. 1237)	Returns a quantile from the standard normal distribution.
	QUANTILE Function (p. 1293)	Returns the quantile from a distribution when you specify the left probability (CDF).
	SQUANTILE Function (p. 1404)	Returns the quantile from a distribution when you specify the right probability (SDF).
	TINV Function (p. 1454)	Returns a quantile from the t distribution.
Random Number	CALL RANBIN Routine (p. 295)	Returns a random variate from a binomial distribution.
	CALL RANCAU Routine (p. 298)	Returns a random variate from a Cauchy distribution.
	CALL RANEXP Routine (p. 304)	Returns a random variate from an exponential distribution.
	CALL RANGAM Routine (p. 306)	Returns a random variate from a gamma distribution.
	CALL RANNOR Routine (p. 309)	Returns a random variate from a normal distribution.
	CALL RANPOI Routine (p. 316)	Returns a random variate from a Poisson distribution.
	CALL RANTBL Routine (p. 319)	Returns a random variate from a tabled probability distribution.
	CALL RANTRI Routine (p. 321)	Returns a random variate from a triangular distribution.
	CALL RANUNI Routine (p. 324)	Returns a random variate from a uniform distribution.
	CALL STREAM Routine (p. 349)	Specifies a random-number stream to use for subsequent calls to the RAND function.
	CALL STREAMINIT Routine (p. 352)	Specifies a random-number generator and seed value for generating random numbers.

Category	Language Elements	Description
	CALL STREAMREWIND Routine (p. 358)	Rewinds a stream to its initial state for subsequent random-number generation.
	NORMAL Function (p. 1142)	Returns a random variate from a normal, or Gaussian, distribution.
	RANBIN Function (p. 1301)	Returns a random variate from a binomial distribution.
	RANCAU Function (p. 1302)	Returns a random variate from a Cauchy distribution.
	RAND Function (p. 1304)	Generates random numbers from a distribution that you specify.
	RANEXP Function (p. 1325)	Returns a random variate from an exponential distribution.
	RANGAM Function (p. 1327)	Returns a random variate from a gamma distribution.
	RANNOR Function (p. 1331)	Returns a random variate from a normal distribution.
	RANPOI Function (p. 1332)	Returns a random variate from a Poisson distribution.
	RANTBL Function (p. 1334)	Returns a random variate from a tabled probability distribution.
	RANTRI Function (p. 1335)	Returns a random variate from a triangular distribution.
	RANUNI Function (p. 1337)	Returns a random variate from a uniform distribution.
	UNIFORM Function (p. 1485)	Returns a random variate from a uniform distribution.
Rounding and Truncation	CEIL Function (p. 424)	Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results.
	CEILZ Function (p. 426)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	FLOOR Function (p. 710)	Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results.
	FLOORZ Function (p. 712)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
	FUZZ Function (p. 743)	Returns the nearest integer if the argument is within 1E−12 of that integer.
	INT Function (p. 963)	Returns the integer value, fuzzed to avoid unexpected floating-point results.
	INTZ Function (p. 1020)	Returns the integer portion of the argument, using zero fuzzing.

Category	Language Elements	Description
	ROUND Function (p. 1348)	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. 1356)	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. 1359)	Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.
	TRUNC Function (p. 1474)	Truncates a numeric value to a specified number of bytes.
SAS File I/O	ATTRC Function (p. 185)	Returns the value of a character attribute for a SAS data set.
	ATTRN Function (p. 188)	Returns the value of a numeric attribute for a SAS data set.
	CEXIST Function (p. 427)	Verifies the existence of a SAS catalog or SAS catalog entry.
	CUROBS Function (p. 504)	Returns the observation number of the current observation.
	DROPNOTE Function (p. 570)	Deletes a note marker from a SAS data set or an external file.
	DSNAME Function (p. 572)	Returns the SAS data set name that is associated with a data set identifier.
	ENVLEN Function (p. 577)	Returns the length of an environment variable.
	EXIST Function (p. 582)	Verifies the existence of a SAS library member within a currently assigned SAS data library.
	FCOPY Function (p. 594)	Copies records from one fileref to another fileref, and returns a value that indicates whether the records were successfully copied.
	FETCH Function (p. 600)	Reads the next non-deleted observation from a SAS data set into the Data Set Data Vector (DDV).
	FETCHOBS Function (p. 602)	Reads a specified observation from a SAS data set into the Data Set Data Vector (DDV).
	GETVARC Function (p. 765)	Returns the value of a SAS data set character variable.
	GETVARN Function (p. 768)	Returns the value of a SAS data set numeric variable.
	IORCMSG Function (p. 1022)	Returns a formatted error message for _IORC_.
	LIBNAME Function (p. 1070)	Assigns or clears a libref for a SAS library.
	LIBREF Function (p. 1074)	Verifies that a libref has been assigned.

Category	Language Elements	Description
	NOTE Function (p. 1151)	Returns an observation ID for the current observation of a SAS data set.
	OPEN Function (p. 1183)	Opens a SAS data set.
	PATHNAME Function (p. 1188)	Returns the physical name of an external file or a SAS library, or returns a blank.
	POINT Function (p. 1221)	Locates an observation that is identified by the NOTE function.
	RENAME Function (p. 1338)	Renames a member of a SAS library, an entry in a SAS catalog, an external file, or a directory.
	REWIND Function (p. 1344)	Positions the data set pointer at the beginning of a SAS data set.
	SYSEXIST Function (p. 1434)	Returns a value that indicates whether an operating-environment variable exists in your environment.
	SYSMSG Function (p. 1437)	Returns error or warning message text from processing the last data set or external file function.
	SYSRC Function (p. 1445)	Returns a system error number.
	VARFMT Function (p. 1495)	Returns the format that is assigned to a SAS data set variable.
	VARINFMT Function (p. 1497)	Returns the informat that is assigned to a SAS data set variable.
	VARLABEL Function (p. 1499)	Returns the label that is assigned to a SAS data set variable.
	VARLEN Function (p. 1501)	Returns the length of a SAS data set variable.
	VARNAME Function (p. 1503)	Returns the name of a SAS data set variable.
	VARNUM Function (p. 1504)	Returns the number of a variable's position in a SAS data set.
	VARTYPE Function (p. 1508)	Returns the data type of a SAS data set variable.
Search	WHICHC Function (p. 1570)	Searches for a character value that is equal to the first argument, and returns the index of the first matching value.
	WHICHN Function (p. 1571)	Searches for a numeric value that is equal to the first argument, and returns the index of the first matching value.
Sort	CALL SORTC Routine (p. 342)	Sorts the values of character arguments.
	SORT Function (p. 1397)	Sorts a list of variables.

Category	Language Elements	Description
Special	ADDRLONG Function (p. 140)	Returns the memory address of a variable on a 64-bit platform.
	CALL POKELONG Routine (p. 279)	Writes a value directly into memory on a 64-bit platform.
	CALL SLEEP Routine (p. 338)	For a specified period of time, suspends the execution of a program that invokes this CALL routine.
	CALL SYSTEM Routine (p. 369)	Submits an operating environment command for execution.
	DIF Function (p. 542)	Returns differences between an argument and its nth lag.
	FMTINFO Function (p. 714)	Retrieves information about a format or informat.
	GETOPTION Function (p. 761)	Returns the value of a SAS system or graphics option.
	INPUT Function (p. 956)	Returns the value that is produced when SAS converts an expression by using the specified informat.
	INPUTC Function (p. 959)	Enables you to specify a character informat at run time.
	INPUTN Function (p. 962)	Enables you to specify a numeric informat at run time.
	LAG Function (p. 1033)	Returns values from a queue.
	PEEKCLONG Function (p. 1214)	Stores the contents of a memory address in a character variable on a 64-bit platform.
	PEEKLONG Function (p. 1216)	Stores the contents of a memory address in a numeric variable on a 64-bit platform.
	PTRLONGADD Function (p. 1280)	Returns the pointer address as a character variable on a 64-bit platform.
	PUT Function (p. 1281)	Returns a value using a specified format.
	PUTC Function (p. 1284)	Enables you to specify a character format at run time.
	PUTN Function (p. 1288)	Enables you to specify a numeric format at run time.
	SLEEP Function (p. 1393)	Suspends the execution of a program that invokes this function for a period of time.
	SYSEXIST Function (p. 1434)	Returns a value that indicates whether an operating-environment variable exists in your environment.
	SYSGET Function (p. 1436)	Returns the value of the specified operating-environment variable.
	SYSPARM Function (p. 1439)	Returns the system parameter string.

Category	Language Elements	Description
	SYSPROCESSID Function (p. 1440)	Returns the process ID of the current process.
	SYSPROCESSNAME Function (p. 1441)	Returns the process name that is associated with a given process ID, or returns the name of the current process.
	SYSPROD Function (p. 1443)	Determines whether a product is licensed.
	SYSTEM Function (p. 1446)	Issues an operating environment command during a SAS session, and returns the system return code.
State and ZIP code	FIPNAME Function (p. 703)	Converts two-digit FIPS codes to uppercase state names.
	FIPNAMEL Function (p. 705)	Converts two-digit FIPS codes to mixed case state names.
	FIPSTATE Function (p. 707)	Converts two-digit FIPS codes to two-character state postal codes.
	STFIPS Function (p. 1410)	Converts state postal codes to FIPS state codes.
	STNAME Function (p. 1412)	Converts state postal codes to uppercase state names.
	STNAMEL Function (p. 1414)	Converts state postal codes to mixed case state names.
	ZIPCITY Function (p. 1580)	Returns a city name and the two-character postal code that corresponds to a ZIP code.
	ZIPCITYDISTANCE Function (p. 1583)	Returns the geodetic distance between two ZIP code locations.
	ZIPFIPS Function (p. 1584)	Converts ZIP codes to two-digit FIPS codes.
	ZIPNAME Function (p. 1586)	Converts ZIP codes to uppercase state names.
	ZIPNAMEL Function (p. 1588)	Converts ZIP codes to mixed-case state names.
	ZIPSTATE Function (p. 1590)	Converts ZIP codes to two-character state postal codes.
Trigonometric	ARCOS Function (p. 177)	Returns the arccosine.
	ARSIN Function (p. 179)	Returns the arcsine.
	ATAN Function (p. 182)	Returns the arc tangent.
	ATAN2 Function (p. 183)	Returns the arc tangent of the x and y coordinates of a right triangle, in radians.
	COS Function (p. 485)	Returns the cosine.
	COT Function (p. 487)	Returns the cotangent.
	CSC Function (p. 499)	Returns the cosecant.

Category	Language Elements	Description
	SEC Function (p. 1380)	Returns the secant.
	SIN Function (p. 1390)	Returns the sine.
	TAN Function (p. 1447)	Returns the tangent.
Variable Control	CALL LABEL Routine (p. 253)	Assigns a variable label to a specified character variable.
	CALL SET Routine (p. 336)	Links SAS data set variables to DATA step or macro variables that have the same name and data type.
	CALL VNAME Routine (p. 371)	Assigns a variable name as the value of a specified variable.
Variable Information	CALL VNEXT Routine (p. 373)	Returns the name, type, and length of a variable that is used in a DATA step.
	VARRAY Function (p. 1505)	Returns a value that indicates whether the specified name is an array.
	VARRAYX Function (p. 1507)	Returns a value that indicates whether the value of the specified argument is an array.
	VFORMAT Function (p. 1513)	Returns the format that is associated with the specified variable.
	VFORMATD Function (p. 1515)	Returns the decimal value of the format that is associated with the specified variable.
	VFORMATDX Function (p. 1517)	Returns the decimal value of the format that is associated with the value of the specified argument.
	VFORMATN Function (p. 1519)	Returns the format name that is associated with the specified variable.
	VFORMATNX Function (p. 1521)	Returns the format name that is associated with the value of the specified argument.
	VFORMATW Function (p. 1523)	Returns the format width that is associated with the specified variable.
	VFORMATWX Function (p. 1525)	Returns the format width that is associated with the value of the specified argument.
	VFORMATX Function (p. 1527)	Returns the format that is associated with the value of the specified argument.
	VINARRAY Function (p. 1529)	Returns a value that indicates whether the specified variable is a member of an array.
	VINARRAYX Function (p. 1530)	Returns a value that indicates whether the value of the specified argument is a member of an array.

Category	Language Elements	Description
	VINFORMAT Function (p. 1532)	Returns the informat that is associated with the specified variable.
	VINFORMATD Function (p. 1534)	Returns the decimal value of the informat that is associated with the specified variable.
	VINFORMATDX Function (p. 1536)	Returns the decimal value of the informat that is associated with the value of the specified variable.
	VINFORMATN Function (p. 1537)	Returns the informat name that is associated with the specified variable.
	VINFORMATNX Function (p. 1539)	Returns the informat name that is associated with the value of the specified argument.
	VINFORMATW Function (p. 1541)	Returns the informat width that is associated with the specified variable.
	VINFORMATWX Function (p. 1543)	Returns the informat width that is associated with the value of the specified argument.
	VINFORMATX Function (p. 1545)	Returns the informat that is associated with the value of the specified argument.
	VLABEL Function (p. 1547)	Returns the label that is associated with the specified variable.
	VLABELX Function (p. 1548)	Returns the label that is associated with the value of the specified argument.
	VLENGTH Function (p. 1550)	Returns the compile-time (allocated) size of the specified variable.
	VLENGTHX Function (p. 1552)	Returns the compile-time (allocated) size for the variable with a name that is the same as the value of the argument.
	VNAME Function (p. 1554)	Returns the name of the specified variable.
	VNAMEX Function (p. 1556)	Validates the value of the specified argument as a variable name.
	VTYPE Function (p. 1557)	Returns the type (character or numeric) of the specified variable.
	VTYPEX Function (p. 1559)	Returns the type (character or numeric) for the value of the specified argument.
	VVALUE Function (p. 1561)	Returns the formatted value that is associated with the variable that you specify.
	VVALUEX Function (p. 1563)	Returns the formatted value that is associated with the argument that you specify.

Category	Language Elements	Description
Web Tools	HTMLDECODE Function (p. 936)	Decodes a string that contains HTML numeric character references or HTML character entity references, and returns the decoded string.
	HTMLENCODE Function (p. 937)	Encodes characters using HTML character entity references, and returns the encoded string.
	URLDECODE Function (p. 1487)	Returns a string that was decoded using the URL escape syntax.
	URLENCODE Function (p. 1489)	Returns a string that was encoded using the URL escape syntax.

Dictionary

ABS Function

Returns the absolute value.

Category: Mathematical

Syntax

ABS(*argument*)

Required Argument

argument
specifies a constant, variable, or expression.

Details

The ABS function returns a nonnegative number that is equal in magnitude to the magnitude of the argument.

Example

```
data _null_;
   x=abs(2.4);
   y=abs(-3);
   put x= y=;
run;
```

The preceding statements produce these results:

```
x=2.4
y=3
```

ADDRLONG Function

Returns the memory address of a variable on a 64-bit platform.

Categories: Binary Results
Special
Client Only

Syntax

ADDRLONG(*variable*)

Required Argument

variable
specifies a variable.

Details

The return value is a character string that contains the binary representation of the address. To display this value, use the $HEX*w*. format to convert the binary value to its hexadecimal equivalent. If you store the result in a variable, that variable should be a character variable with a length of at least eight characters for portability. If you assign the result to a variable that does not have a specified length, that variable is assigned a length of 20 characters.

Example

This example returns the pointer address for the variable ITEM and formats the value.

```
data characterlist;
   item=6345;
   x=addrlong(item);
   put x $hex16.;
run;
```

SAS writes the following output to the log:

```
480063B020202020
```

AIRY Function

Returns the value of a differential equation.

Category: Mathematical

Syntax

AIRY(*x*)

Required Argument

x
specifies a numeric constant, variable, or expression.

Details

The AIRY function returns the value of a differential equation. (For more information, see the list of References on page 1609.) The AIRY function is the solution of this differential equation with two conditions:

$$w^{(2)} - xw = 0$$

Here is the first condition:

$$w(0) = \frac{1}{3^{\frac{2}{3}}\Gamma\left(\frac{2}{3}\right)}$$

Here is the second condition:

$$w'(0) = -\frac{1}{3^{\frac{1}{3}}\Gamma\left(\frac{1}{3}\right)}$$

Example

```
data _null_;
   x=airy(2.0);
   put x=;
   x=airy(-2.0);
   put x=;
run;
```

The preceding statements produce these results:

```
x=0.0349241304
x=0.2274074282
```

ALLCOMB Function

Generates all combinations of the values of *n* variables taken *k* at a time in a minimal change order.

Category: Combinatorial

Restriction: The ALLCOMB function cannot be executed when you use the %SYSFUNC macro.

Syntax

ALLCOMB(*count*, *k*, *variable-1, ..., variable-n*)

Required Arguments

count
specifies an integer variable that is assigned values from 1 to the number of combinations in a loop.

k
specifies an integer constant, variable, or expression between 1 and *n*, inclusive, that specifies the number of items in each combination.

variable
specifies either all numeric variables or all character variables that have the same length. The values of these variables are permuted.

Restriction: Specify no more than 33 items. If you need to find combinations of more than 33 items, use the CALL ALLCOMBI routine.

Requirement	Initialize these variables before executing the ALLCOMB function.
Tip	After executing ALLCOMB, the first *k* variables contain the values in one combination.

Details

Use the ALLCOMB function in a loop where the first argument to ALLCOMB accepts each integral value from 1 to the number of combinations, and where *k* is constant. The number of combinations can be computed by using the COMB function. On the first execution, the argument types and lengths are checked for consistency. On each subsequent execution, the values of two variables are interchanged.

For the ALLCOMB function, these actions occur:

- On the first execution, ALLCOMB returns 0.
- If the values of *variable-i* and *variable-j* were interchanged, where *i*<*j*, ALLCOMB returns *i*.
- If no values were interchanged because all combinations were already generated, ALLCOMB returns –1.

If you execute the ALLCOMB function with the first argument out of sequence, the results are not useful. In particular, if you initialize the variables and immediately execute the ALLCOMB function with a first argument of *j*, you do not get the *j*th combination (except when *j* is 1). To get the *j*th combination, you must execute ALLCOMB *j* times. The first argument takes values from 1 through *j* in that exact order.

Comparisons

SAS provides four functions or CALL routines for generating combinations:

- ALLCOMB generates all *possible* combinations of the *values*, *missing* or *nonmissing*, of n variables. The values can be any numeric or character value. Each combination is formed from the previous combination by removing one value and inserting another value.
- LEXCOMB generates all *distinct* combinations of the *nonmissing values* of several variables. The values can be any numeric or character value. The combinations are generated in lexicographic order.
- ALLCOMBI generates all combinations of the *indices* of N items, where *indices* are integers from 1 to N. Each combination is formed from the previous combination by removing one index and inserting another index.
- LEXCOMBI generates all combinations of the *indices* of N items, where *indices* are integers from 1 to N. The combinations are generated in lexicographic order.

ALLCOMBI is the fastest of these functions and CALL routines. LEXCOMB is the slowest.

Example

Here is an example of the ALLCOMB function.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   n=dim(x);
   k=3;
   ncomb=comb(n, k);
   do j=1 to ncomb+1;
      rc=allcomb(j, k, of x[*]);
      put j 5. +3 x1-x3 +3 rc=;
   end;
run;
```

The preceding statements produce these results:

```
    1   ant bee cat    rc=0
    2   ant bee ewe    rc=3
    3   ant bee dog    rc=3
    4   ant cat dog    rc=2
    5   ant cat ewe    rc=3
    6   ant dog ewe    rc=2
    7   bee dog ewe    rc=1
    8   bee dog cat    rc=3
    9   bee ewe cat    rc=2
   10   dog ewe cat    rc=1
   11   dog ewe cat    rc=-1
```

See Also

CALL Routines:

- “CALL ALLCOMB Routine” on page 215

ALLPERM Function

Generates all permutations of the values of several variables in a minimal change order.

Category: Combinatorial

Syntax

ALLPERM(*count*, *variable-1* <,*variable-2* ..., *variable-n*>)

Required Arguments

count
specifies a variable with an integer value that ranges from 1 to the number of permutations.

variable
specifies either all numeric variables or all character variables that have the same length. The values of these variables are permuted.

Restriction: Specify no more than 18 variables.

Requirement: Initialize these variables before you execute the ALLPERM function.

Details

The Basics

Use the ALLPERM function in a loop where the first argument to ALLPERM accepts each integral value from 1 to the number of permutations. On the first execution, the argument types and lengths are checked for consistency. On each subsequent execution, the values of two consecutive variables are interchanged.

Note: You can compute the number of permutations by using the PERM function. For more information, see the "PERM Function" on page 1217. For the ALLPERM function, the following values are returned:

- 0 if *count*=1
- J if the values of *variable-J* and *variable-k* are interchanged, where k=j+1
- –1 if *count*>N!

If you use the ALLPERM function and the first argument is out of sequence, the results are not useful. For example, if you initialize the variables and immediately execute the ALLPERM function with a first argument of *k*, your result is not the *k*th permutation (except when *k* is 1). To get the *k*th permutation, you must execute the ALLPERM function *k* times. The first argument takes values from 1 through *k* in that exact order.

ALLPERM always produces N! permutations even if some of the variables have equal values or missing values. If you want to generate only the distinct permutations when there are equal values, or if you want to omit missing values from the permutations, use the LEXPERM function instead.

Note: The ALLPERM function cannot be executed when you use the %SYSFUNC macro.

Comparisons

SAS provides three functions or CALL routines for generating all permutations:

- ALLPERM generates all possible permutations of the values, missing or nonmissing, of several variables. Each permutation is formed from the previous permutation by interchanging two consecutive values.
- LEXPERM generates all distinct permutations of the nonmissing values of several variables. The permutations are generated in lexicographic order.
- LEXPERK generates all distinct permutations of *k* of the nonmissing values of N variables. The permutations are generated in lexicographic order.

ALLPERM is the fastest of these functions and CALL routines. LEXPERK is the slowest.

Example

The following example generates permutations of given values by using the ALLPERM function.

```
data _null_;
   array x [4] $3 ('ant' 'bee' 'cat' 'dog');
   n=dim(x);
   nfact=fact(n);
   do i=1 to nfact+1;
      change=allperm(i, of x[*]);
      put i 5. +2 change +2 x[*];
   end;
run;
```

The preceding statements produce these results:

```
 1  0   ant bee cat dog
 2  3   ant bee dog cat
 3  2   ant dog bee cat
 4  1   dog ant bee cat
 5  3   dog ant cat bee
 6  1   ant dog cat bee
 7  2   ant cat dog bee
 8  3   ant cat bee dog
 9  1   cat ant bee dog
10  3   cat ant dog bee
11  2   cat dog ant bee
12  1   dog cat ant bee
13  3   dog cat bee ant
14  1   cat dog bee ant
15  2   cat bee dog ant
16  3   cat bee ant dog
17  1   bee cat ant dog
18  3   bee cat dog ant
19  2   bee dog cat ant
20  1   dog bee cat ant
21  3   dog bee ant cat
22  1   bee dog ant cat
23  2   bee ant dog cat
24  3   bee ant cat dog
25  -1   bee ant cat dog
```

See Also

Functions:

- “LEXPERM Function” on page 1065

CALL Routines:

- “CALL ALLPERM Routine” on page 222
- “CALL RANPERK Routine” on page 312
- “CALL RANPERM Routine” on page 314

ANYALNUM Function

Searches a character string for an alphanumeric character, and returns the first position at which the character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYALNUM(*string* <,*start*>)

Required Argument

string
specifies a character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 nonalphanumeric character.

Examples

Example 1: Scanning a String from Left to Right

This example uses the ANYALNUM function to search a string from left to right for alphanumeric characters.

```
data _null_;
   string='Next = Last + 1;';
   j=0;
   do until(j=0);
      j=anyalnum(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following results 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
That's all
```

Example 2: Scanning a String from Right to Left

This example uses the ANYALNUM function to search a string from right to left for alphanumeric characters.

```
data _null_;
   string='Next = Last + 1;';
   j=999999;
   do until(j=0);
      j=anyalnum(string, 1-j);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
```

```
    end;
run;
```

SAS writes the following results 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
That's all
```

See Also

Functions:

- "NOTALNUM Function" on page 1142

ANYALPHA Function

Searches a character string for an alphabetic character, and returns the first position at which the character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYALPHA(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 nonalphabetic character.

Examples

Example 1: Searching a String for Alphabetic Characters

The following example uses the ANYALPHA function to search a string from left to right for alphabetic characters.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anyalpha(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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
   That's all
```

Example 2: Identifying Control Characters by Using the ANYALPHA Function

You can execute the following program to show the control characters that are identified by the ANYALPHA function.

```
data test;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   anyalpha=anyalpha(byte);
   output;
 end;

 proc print data=test;
 run;
```

See Also

Functions:

- “NOTALPHA Function” on page 1144

ANYCNTRL Function

Searches a character string for a control character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYCNTRL(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 test;
do dec=0 to 255;
   drop byte;
   byte=byte(dec);
   hex=put(dec, hex2.);
   anycntrl=anycntrl(byte);
   if anycntrl then output;
 end;

 proc print data=test;
 run;
```

See Also

Functions:

- “NOTCNTRL Function” on page 1147

ANYDIGIT Function

Searches a character string for a digit, and returns the first position at which the digit is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYDIGIT(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The ANYDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system option.

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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 example uses the ANYDIGIT function to search for a character that is a digit.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anydigit(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=14 c=1
j=15 c=2
j=17 c=3
That's all
```

See Also

Functions:

- “NOTDIGIT Function” on page 1149

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 position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYFIRST(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The ANYFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system option.

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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 example 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_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anyfirst(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

The preceding statements produce these results:

```
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
That's all
```

See Also

Functions:

- “NOTFIRST Function” on page 1154

ANYGRAPH Function

Searches a character string for a graphical character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYGRAPH(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and LOCALE system options.

The ANYGRAPH function searches a string for the first occurrence of a graphical character. A graphical character is specified 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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

This example uses the ANYGRAPH function to search a string for graphical characters.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anygraph(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

The preceding statements produce these results:

```
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=;
That's all
```

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 test;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   anygraph=anygraph(byte);
   output;
```

```
end;

proc print data=test;
run;
```

See Also

Functions:

- "NOTGRAPH Function" on page 1156

ANYLOWER Function

Searches a character string for a lowercase letter, and returns the first position at which the letter is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYLOWER(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 example uses the ANYLOWER function to search a string for any character that is a lowercase letter.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anylower(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
```

```
   end;
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
That's all
```

See Also

Functions:

- "NOTLOWER Function" on page 1159

ANYNAME Function

Searches a character string for a character that is valid in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYNAME(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The ANYNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system option.

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 (for example, 0–9), 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 example uses the ANYNAME function to search a string for any character that is valid in a SAS variable name under VALIDVARNAME=V7.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anyname(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
```

```
            put +3 j= c=;
         end;
      end;
   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
That's all
```

See Also

Functions:

- “NOTNAME Function” on page 1161

ANYPRINT Function

Searches a character string for a printable character, and returns the first position at which that character is found.

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYPRINT(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 one of these ways:

- 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 these conditions 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 nonprintable character.

Examples

Example 1: Searching a String for a Printable Character

The following example uses the ANYPRINT function to search a string for printable characters.

```
data _null_;
```

```
   string='Next = _n_ + 12E3;';
   do j=1 to length(strip(string));
   c=substr(string,j,1);
   put +3 j= c=;
end;
   put +3 "That's all";
run;
```

The preceding statements produce these results:

```
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=;
That's all
```

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 test;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec,hex2.);
   anyprint=anyprint(byte);
   output;
 end;

 proc print data=test;
 run;
```

See Also

Functions:

- “NOTPRINT Function” on page 1163

ANYPUNCT Function

Searches a character string for a punctuation character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYPUNCT(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 or if it is missing, then

```
x=anypunct(x,.);
```

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 example uses the ANYPUNCT function to search a string for punctuation characters.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anypunct(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=6 c==
j=8 c=_
j=10 c=_
j=12 c=+
j=18 c=;
That's all
```

Example 2: Identifying Punctuation 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 test;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   anypunct=anypunct(byte);
   output;
 end;

 proc print data=test;
 run;
```

See Also

Functions:

- "NOTPUNCT Function" on page 1167

ANYSPACE Function

Searches a character string for a whitespace character (blank, horizontal or vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYSPACE(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 example uses the ANYSPACE function to search a string for a character that is a whitespace character.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anyspace(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=7 c=
j=11 c=
j=13 c=
That's all
```

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 test;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   anyspace=anyspace(byte);
   output;
 end;

 proc print data=test;
 run;
```

See Also

Functions:

- “NOTSPACE Function” on page 1169

ANYUPPER Function

Searches a character string for an uppercase letter, and returns the first position at which the letter is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYUPPER(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 example uses the ANYUPPER function to search a string for an uppercase letter.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anyupper(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=1 c=N
j=16 c=E
That's all
```

See Also

Functions:

- “NOTUPPER Function” on page 1172

ANYXDIGIT Function

Searches a character string for a hexadecimal character that represents a digit, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

ANYXDIGIT(*string* <,*start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

The ANYXDIGIT function does not depend on the TRANTAB, ENCODING, or LOCALE system option.

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, or if the second argument has a missing value, 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 one of these ways:

- 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 0 when one of these conditions 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 example uses the ANYXDIGIT function to search a string for a hexadecimal character that represents a digit.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=anyxdigit(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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
That's all
```

See Also

Functions:

- “NOTXDIGIT Function” on page 1174

ARCOS Function

Returns the arccosine.

Category: Trigonometric

Syntax

ARCOS(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range between −1 and 1

Details

The ARCOS function returns the arccosine (inverse cosine) of the argument. The value that is returned is specified in radians.

Example

```
data _null_;
   x=arcos(1);
   y=arcos(0);
   z=arcos(-0.5);
   put x= y= z=;
run;
```

The preceding statements produce these results:

```
x=0 y=1.5707963268 z=2.0943951024
```

ARCOSH Function

Returns the inverse hyperbolic cosine.

Category: Hyperbolic

Syntax

ARCOSH(*x*)

Required Argument

x

specifies a numeric constant, variable, or expression.

Range *x* >= 1

Details

The ARCOSH function computes the inverse hyperbolic cosine. The ARCOSH function is mathematically specified by the following equation, where *x* >= 1:

$$ARCOSH(x) = log\left(x + \sqrt{x^2 - 1}\right)$$

Example

The following example computes the inverse hyperbolic cosine.

```
data _null_;
   x=arcosh(5);
   x1=arcosh(13);
   put x=;
   put x1=;
run;
```

SAS writes the following output to the log:

```
x=2.2924316696
x1=3.2566139548
```

See Also

Functions:

- “TANH Function” on page 1448
- “ARSINH Function” on page 180
- “COSH Function” on page 486
- “SINH Function” on page 1391

■ “ARTANH Function” on page 181

ARSIN Function

Returns the arcsine.

Category: Trigonometric

Syntax

ARSIN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range between −1 and 1

Details

The ARSIN function returns the arcsine (inverse sine) of the argument. The value that is returned is specified in radians.

Example

```
data one;
   x=arsin(0);
   put x=;
   y=arsin(1);
   put y=;
   z=arsin(-0.5);
   put z=;
run;
```

The preceding statements produce these results:

```
x=0
y=1.5707963268
z=-0.523598776
```

ARSINH Function

Returns the inverse hyperbolic sine.

Category: Hyperbolic

Syntax

ARSINH(*x*)

Required Argument

x
specifies a numeric constant, variable, or expression.

Range $-\infty < x < \infty$

Details

The ARSINH function computes the inverse hyperbolic sine. The ARSINH function is mathematically specified 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.

Example

This example computes the inverse hyperbolic sine.

```
data _null_;
   x=arsinh(5);
   x1=arsinh(-5);
   put x=;
   put x1=;
run;
```

```
x=2.3124383413
x1=-2.312438341
```

See Also

Functions:

- “ARCOSH Function” on page 177
- “ARTANH Function” on page 181
- “COSH Function” on page 486
- “SINH Function” on page 1391
- “TANH Function” on page 1448

ARTANH Function

Returns the inverse hyperbolic tangent.

Category: Hyperbolic

Syntax

ARTANH(*x*)

Required Argument

x

specifies a numeric constant, variable, or expression.

Range $-1 < x < 1$

Details

The ARTANH function computes the inverse hyperbolic tangent. The ARTANH function is mathematically specified by the following equation, where –1 < x< 1:

$$ARTANH(x) = \frac{1}{2} log\left(\frac{1+x}{1-x}\right)$$

Example

The following example computes the inverse hyperbolic tangent.

```
data _null_;
   x=artanh(0.5);
   put x=;
```

```
run;
```

The preceding statements produce this result:

```
x=0.5493061443
```

See Also

Functions:

- “ARCOSH Function” on page 177
- “ARSINH Function” on page 180
- “COSH Function” on page 486
- “SINH Function” on page 1391
- “TANH Function” on page 1448

ATAN Function

Returns the arc tangent.

Category: Trigonometric

Syntax

ATAN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The ATAN function returns the 2-quadrant arc tangent (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 $\pi/2$. If the argument is missing, ATAN returns a missing value.

Comparisons

The ATAN function is similar to the ATAN2 function, except that ATAN2 calculates the arc tangent of the angle from the ratio of two arguments rather than from one argument.

Example

```
data one;
   x=atan(0);
   put x=;
run;
```

The preceding statements produce this result:

```
x=0
```

```
data one;
   x=atan(1);
   put x=;
run;
```

The preceding statements produce this result:

```
x=0.7853981634
```

```
data one;
   x=atan(-9.0);
   put x=;
run;
```

The preceding statements produce this result:

```
x=-1.460139106
```

See Also

Functions:

- “ATAN2 Function” on page 183

ATAN2 Function

Returns the arc tangent of the x and y coordinates of a right triangle, in radians.

Category: Trigonometric

Syntax

ATAN2(*expression-1*, *expression-2*)

Required 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.

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.

Details

The ATAN2 function returns the arc tangent (inverse tangent) of two numeric variables. The result of this function is similar to the result of calculating the arc tangent of *argument-1 / argument-2*, except that the signs of both arguments are used to determine the quadrant of the result. ATAN2 returns the result in radians, which is a value between $-\pi$ and π. If either of the arguments in ATAN2 is missing, ATAN2 returns a missing value.

Comparisons

The ATAN2 function is similar to the ATAN function, except that ATAN calculates the arc tangent of the angle from the value of one argument rather than from two arguments.

Example

```
data one;
   a=atan2(-1, 0.5);
   put a=;
   b=atan2(6, 8);
   put b=;
   c=atan2(5, -3);
   put c=;
run;
```

The preceding statements produce these results:

```
a=-1.107148718
b=0.6435011088
c=2.1112158271
```

See Also

Functions:

- “ATAN Function” on page 182

ATTRC Function

Returns the value of a character attribute for a SAS data set.

Category: SAS File I/O

Note: If the value of the attribute specified is invalid, a missing value is returned.

Syntax

ATTRC(*data-set-id*, CHARSET | COMPRESS | DATAREP | ENCODING | ENCRYPT | ENGINE | LABEL | LIB | MEM | MODE | MTYPE | SORTEDBY | SORTLVL | SORTSEQ | TYPE)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

CHARSET
returns a value for the character set of the computer that created the data set.

empty string	data set not sorted
ASCII	ASCII character set
EBCDIC	EBCDIC character set
ANSI	OS/2 ANSI standard ASCII character set
OEM	OS/2 OEM code format

COMPRESS
returns a value that specifies how a data set is compressed.

CHAR	specifies that the data set is compressed as character data.
BINARY	specifies that the data set is compressed as binary data.
NO	specifies that the data set is not compressed.

DATAREP
returns a value that indicates whether the data set is in a native format.

NATIVE indicates that the data set is in native format.

FOREIGN indicates that the data set is in a foreign format.

ENCODING
specifies the encoding of the data set.

```
%let dsn=libref.data_set_name;
%let dsid=%sysfunc(open(&dsn,i));
%put &dsn ENCODING is: %sysfunc(attrc(&dsid,encoding));
%let dsid=%sysfunc(close(&dsid));
```

ENCRYPT
returns 'YES' or 'NO' depending on whether the SAS data set is encrypted.

ENGINE
returns the name of the engine that is used to access the data set.

LABEL
returns the label assigned to the data set.

LIB
returns the libref of the SAS library in which the data set resides.

MEM
returns the SAS data set name.

MODE
returns the mode in which the SAS data set was opened, such as:

I INPUT mode allows random access if the engine supports it. Otherwise, it defaults to IN mode.

IN INPUT mode reads sequentially and allows revisiting observations.

IS INPUT mode reads sequentially but does not allow revisiting observations.

N NEW mode creates a new data set.

U UPDATE mode allows random access if the engine supports it. Otherwise, it defaults to UN mode.

UN UPDATE mode reads sequentially and allows revisiting observations.

US UPDATE mode reads sequentially but does not allow revisiting observations.

V UTILITY mode allows modification of variable attributes and indexes associated with the data set.

MTYPE
returns the SAS library member type.

SORTEDBY
returns an empty string if the data set is not sorted. Otherwise, it returns the names of the BY variables in the standard BY statement format.

SORTLVL
returns a value that indicates how a data set was sorted:

Empty string	Data set is not sorted.
WEAK	Sort order of the data set was established by the user (for example, through the SORTEDBY data set option). The system cannot validate its correctness, so the order of observations cannot be depended on.
STRONG	Sort order of the data set was established by the software (for example, through PROC SORT or the OUT= option in the CONTENTS procedure).

SORTSEQ
returns an empty string if the data set is sorted on the native computer or if the sort collating sequence is the default for the operating environment. Otherwise, it returns the name of the alternate collating sequence used to sort the file.

TYPE
returns the SAS data set type.

Examples

Example 1: Writing a Message about Input Sequential Mode to the SAS Log

This example generates a message if the SAS data set has not been opened in INPUT SEQUENTIAL mode. The message is written to the SAS log as follows:

```
%macro test;
%let dsid=%sysfunc(open(sashelp.class));
%let mode=%sysfunc(attrc(&dsid,MODE));
%let rc=%sysfunc(close(&dsid));
%put &mode;
%if &mode ne IS %then
%put Data set has not been opened in INPUT SEQUENTIAL mode.;
%mend;
%test
```

Example 2: Testing Whether a Data Set Has Been Sorted

This example tests whether a data set has been sorted and writes the result to the SAS log:

```
data _null_;
   dsid=open("sasdata.sortcars", "i");
   charset=attrc(dsid, "CHARSET");
   if charset = "" then
      put "Data set has not been sorted.";
   else put "Data set sorted with " charset
            "character set.";
   rc=close(dsid);
```

```
run;
```

See Also

Functions

- “ATTRN Function” on page 188
- “OPEN Function” on page 1183

ATTRN Function

Returns the value of a numeric attribute for a SAS data set.

Category: SAS File I/O

Note: If the value of the data set attribute specified is invalid, a missing value is returned.

Syntax

ATTRN(*data-set-id*, ALTERPW | ANOBS | ANY | ARAND | ARWU | AUDIT | AUDIT_DATA | AUDIT_BEFORE | AUDIT_ERROR | CRDTE | ICONST | INDEX | ISINDEX | ISSUBSET | LRECL | LRID | MAXGEN | MAXRC | MODTE | NDEL | NEXTGEN | NLOBS | NLOBSF | NOBS | NVARS | PW | RADIX | READPW | REUSE | TAPE | WHSTMT | WRITEPW)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

ALTERPW
specifies whether a password is required to alter the data set.

1 the data set is Alter protected.

0 the data set is not Alter protected.

ANOBS
specifies whether the engine knows the number of observations.

1 the engine knows the number of observations.

0 the engine does not know the number of observations.

ANY
specifies whether the data set has observations or variables.

-1 the data set has no observations or variables.

0 the data set has no observations.

1 the data set has observations and variables.

Alias VAROBS

ARAND
specifies whether the engine supports random access.

1 the engine supports random access.

0 the engine does not support random access.

Alias RANDOM

ARWU
specifies whether the engine can manipulate files.

1 the engine is not read-only. It can create or update SAS files.

0 the engine is read-only.

AUDIT
specifies whether logging to an audit file is enabled.

1 logging is enabled.

0 logging is suspended.

AUDIT_DATA
specifies whether after-update record images are stored.

1 after-update record images are stored.

0 after-update record images are not stored.

AUDIT_BEFORE
specifies whether before-update record images are stored.

1 before-update record images are stored.

0 before-update record images are not stored.

AUDIT_ERROR
specifies whether unsuccessful after-update record images are stored.

1 unsuccessful after-update record images are stored.

0 unsuccessful after-update record images are not stored.

CRDTE
specifies the date on which the data set was created. The value that is returned is the internal SAS datetime value for the creation date.

Tip Use the DATETIME. format to display this value.

ICONST
returns information about the existence of integrity constraints for a SAS data set.

0 no integrity constraints.

1 one or more general integrity constraints.

2 one or more referential integrity constraints.

3 both one or more general integrity constraints and one or more referential integrity constraints.

INDEX
specifies whether the data set supports indexing.

1 indexing is supported.

0 indexing is not supported.

ISINDEX
specifies whether the data set is indexed.

1 at least one index exists for the data set.

0 the data set is not indexed.

ISSUBSET
specifies whether the data set is a subset.

1 at least one WHERE clause is active.

0 no WHERE clause is active.

LRECL
specifies the logical record length.

LRID
specifies the length of the record ID.

MAXGEN
specifies the maximum number of generations.

MAXRC
specifies whether an application checks return codes.

1 an application checks return codes.

0 an application does not check return codes.

MODTE
specifies the last date and time that the data set was modified. The value returned is the internal SAS datetime value.

Tip Use the DATETIME. format to display this value.

NDEL
specifies the number of observations in the data set that are marked for deletion.

NEXTGEN
specifies the next generation number to generate.

NLOBS
specifies the number of logical observations (the observations that are not marked for deletion). An active WHERE clause does not affect this number.

−1 the number of observations is not available.

NLOBSF
specifies the number of logical observations (the observations that are not marked for deletion) by forcing each observation to be read and by taking the FIRSTOBS system option, the OBS system option, and the WHERE clauses into account.

Tip Passing NLOBSF to ATTRN requires the engine to read every observation from the data set that matches the WHERE clause. Based on the file type and file size, reading these observations can be a time-consuming process.

NOBS
specifies the number of physical observations (including the observations that are marked for deletion). An active WHERE clause does not affect this number.

−1 the number of observations is not available.

NVARS
specifies the number of variables in the data set.

PW
specifies whether a password is required to access the data set.

1 the data set is protected.

0 the data set is not protected.

RADIX
specifies whether access by observation number (radix addressability) is allowed.

1 access by observation number is allowed.

0 access by observation number is not allowed.

Note: A data set that is accessed by a tape engine is index addressable although it cannot be accessed by an observation number.

READPW
specifies whether a password is required to read the data set.

1 the data set is Read protected.

0 the data set is not Read protected.

REUSE
specifies whether new observations can be written to free space in compressed SAS data sets.

1 free space can be reused.

0 free space cannot be reused.

TAPE
specifies the status of the data set tape.

1 the data set is a sequential file.

0 the data set is not a sequential file.

WHSTMT
specifies the active WHERE clauses.

0 no WHERE clause is active.

1 a permanent WHERE clause is active.

2 a temporary WHERE clause is active.

3 both permanent and temporary WHERE clauses are active.

WRITEPW
specifies whether a password is required to write to the data set.

1 the data set is Write protected.

0 the data set is not Write protected.

Examples

Example 1: Checking for an Active WHERE Clause

This example checks whether a WHERE clause is currently active for a data set.

```
%macro test;
%let dsid=%sysfunc(open(sashelp.class));
%let iswhere=%sysfunc(attrn(&dsid,whstmt));
%let rc=%sysfunc(close(&dsid));
%put &iswhere;
%if &iswhere %then
%put A WHERE clause is currently active.;
%mend;
%test
```

Example 2: Checking for an Indexed Data Set

This example checks whether a data set is indexed.

```
data _null_;
   dsid=open("mydata");
   isindex=attrn(dsid, "isindex");
   if isindex then put "data set is indexed";
   else put "data set is not indexed";
run;
```

Example 3: Checking a Data Set for Password Protection

This example checks whether a data set is protected with a password.

```
data _null_;
   dsid=open("mydata");
   pw=attrn(dsid, "pw");
   if pw then put "data set is protected";
run;
```

See Also

Functions:

- “ATTRC Function” on page 185
- “OPEN Function” on page 1183

BAND Function

Returns the bitwise logical AND of 32-bit integer values.

Category: Bitwise Logical Operations

Syntax

BAND(*argument-1*, *argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

Details

A bitwise AND is a binary operation that takes two equal-length binary representations and performs the logical AND operation on each pair of the corresponding bits.

If either argument contains a missing value, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, each argument is internally converted to a

32-bit integer. BAND computes the bitwise logical AND of these two integers and converts the result back to a returned-double-floating-point value.

Example

```
data one;
   x = 12;
   y = 10;
   z = BAND(x,y);
   put x=binary4.;
   put y=binary4.;
   put z=binary4. Z=;

run;
```

The preceding statements produce this result:

```
x=1100
y=1010
z=1000 z=8
```

BETA Function

Returns the value of the beta function.

Category: Mathematical

Syntax

BETA(*a*, *b*)

Required Arguments

a
is the first shape parameter, where *a*>0.

b
is the second shape parameter, where *b*>0.

Details

The BETA function is mathematically given by this equation:

$$\beta(a,b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx$$

In this equation, $a>0$, $b>0$. It should be noted that $\Gamma(.)$ is the GAMMA function:

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

If the expression cannot be computed, BETA returns a missing value.

Example

```
data _null_;
   x=beta(5,3);
   y=logbeta(5,3);
   put x= y=;
run;
```

The preceding statements produce these results:

```
x=0.0095238095 y=-4.65396035
```

See Also

Functions:

- "LOGBETA Function" on page 1080

BETAINV Function

Returns a quantile from the beta distribution.

Category: Quantile

Syntax

BETAINV(*p*, *a*, *b*)

Required Arguments

p

is a numeric probability.

Range $0 \le p \le 1$

a
is a numeric shape parameter.

Range $a > 0$

b
is a numeric shape parameter.

Range $b > 0$

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

```
data one;
   x=betainv(0.001, 2, 4);
   put x=;
run;
```

The preceding statements produce this result:

```
x=0.0101017879
```

See Also

Functions:

- "QUANTILE Function" on page 1293

BHAMMING_32 Function

Returns the bitwise Hamming distance between two 32-bit integer values.

Category: Bitwise Logical Operations

Syntax

BHAMMING_32(*argument-1*, *argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

Details

The bitwise Hamming distance between two bit strings is the number of positions in the strings where a bit in one string differs from the corresponding bit in the other string.

If either argument contains a missing value, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, each argument is internally converted to a 32-bit integer. BHAMMING_32 computes the bitwise Hamming distance between these two integers and returns the result as a numeric integer value.

Example

This example shows the functionality of BHAMMING_32.

```
data _null_;
   x = 15;
   y = 124;
   z = BHAMMING_32(x,y);
   put x=binary8.;
   put y=binary8.;
   put z=;
run;
```

These statements produce this result.

```
x=00001111
y=01111100
z=5
```

BHAMMING_HEX Function

Returns the bitwise Hamming distance between two hexadecimal bit strings.

Category: Bitwise Operations

Syntax

BHAMMING_HEX(*hex-string-1*, *hex-string-2*);

Required Arguments

hex-string-1
specifies a character string containing hexadecimal characters. Leading and trailing blanks are ignored.

Range The maximum length depends on the calling environment.

Note *hex-string-1* and *hex-string-2* must have the same length.

hex-string-2
specifies a character string containing hexadecimal characters. Leading and trailing blanks are ignored.

Range The maximum length depends on the calling environment.

Note *hex-string-1* and *hex-string-2* must have the same length.

Details

The bitwise Hamming distance between two bit strings is the number of positions in the strings where a bit in one string differs from the corresponding bit in the other string.

If either argument is blank, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, the hexadecimal strings are internally converted to bit strings. BHAMMING_HEX computes the bitwise Hamming distance between these two bit strings and returns the result as a numeric integer value.

Example

This example shows the functionality of BHAMMING_HEX.

```
%macro hex2bits(x);          1
```

```
   &x._bits=repeat(&x, 4);    2
   &x._bits=' ';
   length hex_char $1;
   drop j k blob hex_char;
   k=1;
   do j = 1 to length(&x);
      hex_char = char(&x,j);
      blob = input( hex_char, hex1.);
      if n(blob) then do;
         substr(&x._bits,k,4)=put(blob, binary4.);
         k + 5;
      end;
   end;
%mend  hex2bits;
data _null_;
   x = '10F';              3
   y = '17C';              4
   z = BHAMMING_HEX(x,y);  5
   %hex2bits(x);           6
   %hex2bits(y);           7
   put x_bits=;            8
   put y_bits=;            9
   put z=;                 10
run;
```

1 This macro converts a character string, specified by &x, that contains hexadecimal digits to a character string, specified by &x._bits, that contains binary digits.

2 This code establishes the output length.

3 The hex value, 10F, corresponds to the binary value 0001 0000 1111 .

4 The hex value 17C corresponds to the binary value 0001 0111 1100 .

5 The BHAMMING_HEX function returns the bitwise Hamming distance between the two hexadecimal bit strings specified by x and y.

6 Convert the value of x to a binary value specfied by x_bits.

7 Convert the value of y to a binary value specified by y_bits.

8 Display the binary value of x.

9 Display the binary value of y.

10 Display the result from the BHAMMING_HEX processing.

These statements produce this result.

```
x_bits=0001 0000 1111
y_bits=0001 0111 1100
z=5
```

BLACKCLPRC Function

Calculates call prices for European options on futures, based on the Black model.

Category: Financial

Syntax

BLACKCLPRC(*E*, *t*, *F*, *r*, *sigma*)

Required Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *F* in the same units.

t
is a nonmissing value that specifies time to maturity, in years.

F
is a nonmissing, positive value that specifies the future price.

Requirement Specify *F* and *E* in the same units.

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

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 the future price.

N
specifies the cumulative normal density function.

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate, which 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 on page 11.

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. Both of these functions return a scalar value.

Example

```
data one;
   a=blackclprc(50, .25, 48, .05, .25);
   b=blackclprc(9, 1/12, 10, .05, .2);
   put a=;
   put b=;
run;
```

The preceding statements produce these results:

```
a=1.5513014272
b=1.0031514194
```

See Also

Functions:

- "BLACKPTPRC Function" on page 202

BLACKPTPRC Function

Calculates put prices for European options on futures, based on the Black model.

Category: Financial

Syntax

BLACKPTPRC(*E*, *t*, *F*, *r*, *sigma*)

Required Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *F* in the same units.

t
is a nonmissing value that specifies time to maturity, in years.

F
is a nonmissing, positive value that specifies the future price.

Requirement Specify *F* and *E* in the same units.

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

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, which is an annual rate that is expressed in terms of continuous compounding.

t
specifies the time to expiration, in years.

F
specifies the 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 on page 11.

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. Both of these functions return a scalar value.

Example

```
data one;
   a=blackptprc(298, .25, 350, .06, .25);
   b=blackptprc(145, .5, 170, .05, .2);
   put a=;
   put b=;
run;
```

The preceding statements produce these results:

```
a=1.8598056393
b=1.4123497991
```

See Also

Functions:

- "BLACKCLPRC Function" on page 200

BLKSHCLPRC Function

Calculates call prices for European options on stocks, based on the Black-Scholes model.

Category: Financial

Syntax

BLKSHCLPRC(*E*, *t*, *S*, *r*, *sigma*)

Required Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t
is a nonmissing value that specifies the time to maturity, in years.

S
is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

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, which 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 on page 11 .

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. Both of these functions return a scalar value.

Example

```
data one;
   a=blkshclprc(50, .25, 48, .05, .25);
   b=blkshclprc(9, 1/12, 10, .05, .2);
   put a=;
   put b=;
run;
```

The preceding statements produce these results:

```
a=1.7989420195
b=1.0435083341
```

See Also

Functions:

- "BLKSHPTPRC Function" on page 206

BLKSHPTPRC Function

Calculates put prices for European options on stocks, based on the Black-Scholes model.

Category: Financial

Syntax

BLKSHPTPRC(*E*, *t*, *S*, *r*, *sigma*)

Required Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t
is a nonmissing value that specifies the time to maturity, in years.

S
is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

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, which 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 on page 11.

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. Both of these functions return a scalar value.

Example

```
data one;
   a=blkshptprc(230, .5, 290, .04, .25);
   b=blkshptprc(350, .3, 400, .05, .2);
   put a=;
   put b=;
run;
```

The preceding statements produce these results:

```
a=1.5659744295
b=1.6409194307
```

See Also

Functions:

- "BLKSHCLPRC Function" on page 204

BLSHIFT Function

Returns the bitwise logical left shift of two 32-bit integer values.

Category: Bitwise Logical Operations

Syntax

BLSHIFT(*argument-1*, *argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range An integer value between 0 to 31, inclusive

Details

A logical shift is a bitwise operation that shifts all the bits of its operand. The two base variants are the logical left shift and the logical right shift.

If either argument contains a missing value, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, each argument is internally converted to a 32-bit integer. BLSHIFT computes the bitwise logical left shift of these two integers and converts the result back to a returned-double-floating-point value.

Example

```
data one;
   x = 12;
   y = 10;
   z = BLSHIFT(x,y);
   put x=binary4.;
   put y=binary4.;
   put z=binary4. Z=;
run;
```

The preceding statements produce this result:

```
x=1100
y=1010
z=0000 z=12288
```

BNOT Function

Returns the bitwise logical NOT of 32-bit integer values.

Category: Bitwise Logical Operations

Syntax

BNOT(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

Details

The bitwise logical NOT, or complement, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value. Bits that are 0 become 1, and those that are 1 become 0.

If the argument contains a missing value, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, the argument is internally converted to a 32-bit integer. BNOT computes the bitwise logical NOT of the integer and converts the result back to a returned-double-floating-point value.

Example

```
data one;
   x=bnot(0F000000Fx);
   put x=hex.;
run;
```

The preceding statements produce this result:

```
x=0FFFFFF0
```

BOR Function

Returns the bitwise logical OR of 32-bit integer values.

Category: Bitwise Logical Operations

Syntax

BOR(*argument-1, argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and $(2^{32})-1$ inclusive

Details

A bitwise logical OR is a binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits. The result in each position is 0 if both bits are 0, otherwise the result is 1.

If either argument contains a missing value, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, each argument is internally converted to a 32-bit integer. BOR computes the bitwise logical OR of these two integers and converts the result back to a returned-double-floating-point value.

Example

```
data one;
   x = 12;
   y = 10;
   z = bor(x,y);
   put x=binary4.;
   put y=binary4.;
   put z=binary4. Z=;
run;
```

The preceding statements produce this result:

```
x=1100
y=1010
z=1110 z=14
```

BRSHIFT Function

Returns the bitwise logical right shift of 32-bit integer values.

Category: Bitwise Logical Operations

Syntax

BRSHIFT(*argument-1*, *argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and (2^{32})–1 inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range An integer between 0 to 31, inclusive

Details

A logical shift is a bitwise operation that shifts all the bits of its operand. The two base variants are the logical left shift and the logical right shift.

If either argument contains a missing value, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, each argument is internally converted to a 32-bit integer. BRSHIFT computes the bitwise logical right shift of these two integers and converts the result back to a returned-double-floating-point value.

Example

```
data one;
   x = 12;
   y = 10;
   z = brshift(x,y);
   put x=binary4.;
   put y=binary4.;
   put z=binary4. Z=;
run;
```

The preceding statements produce this result:

```
x=1100
y=1010
z=0000 z=0
```

BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two 32-bit integer values.

Category: Bitwise Logical Operations

Syntax

BXOR(*argument-1, argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and (2^{32})–1 inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range An integer value between 0 and (2^{32})–1 inclusive

Details

A bitwise XOR is a binary operation that takes two bit patterns of equal length and performs the logical exclusive OR operation on each pair of corresponding bits. The result in each position is 1 if only one of the bits is 1, but will be 0 if both are 0 or both are 1.

If either argument contains a missing value, the function returns a missing value and sets _ERROR_ equal to 1. Otherwise, each argument is internally converted to a 32-bit integer. BXOR computes the bitwise logical exclusive OR of these two integers and converts the result back to a returned-double-floating-point value.

Example

```
data one;
   x = 12;
   y = 10;
   z = BXOR(x,y);
   put x=binary4.;
   put y=binary4.;
   put z=binary4. Z=;
run;
```

The preceding statements produce this result:

```
x=1100
y=1010
z=0110 z=6
```

BYTE Function

Returns one character in the ASCII collating sequence.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Operating environment: The Linux OS uses the ASCII code sequence.

Syntax

BYTE (*n*)

Required Argument

n

specifies an integer that represents a specific ASCII character.

Range 0–255

Details

Length of Returned Variable

In a DATA step, if the BYTE function returns a value to a variable that has not previously been assigned a length, that variable is assigned a length of 1.

ASCII Collating Sequence

For ASCII 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 in certain ASCII operating environments, but the information those characters represent varies with the operating environment.

Example

```
data
one;


x=byte(80);

   put
x=;

run;
```

The preceding statements produce this ASCII result:

```
x=P
```

See Also

Functions:

- “COLLATE Function” on page 442
- “RANK Function” on page 1330

CALL ALLCOMB Routine

Generates all combinations of the values of *n* variables taken *k* at a time in a minimal change order.

Category: Combinatorial

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL ALLCOMB(*count*, *k*, *variable-1*, ..., *variable-n*);

Required Arguments

count
specifies an integer variable that is assigned from 1 to the number of combinations in a loop.

k
specifies an integer constant, variable, or expression between 1 and *n*, inclusive, that specifies the number of items in each combination.

variable
specifies either all numeric variables or all character variables that have the same length. The values of these variables are permuted.

Restriction: Specify no more than 33 items. If you need to find combinations of more than 33 items, use the CALL ALLCOMBI routine.

Requirement: Initialize these variables before calling the ALLCOMB routine.

Tip: After calling the ALLCOMB routine, the first *k* variables contain the values in one combination.

Details

CALL ALLCOMB Processing

Use the CALL ALLCOMB routine in a loop where the first argument to CALL ALLCOMB accepts each integral value from 1 to the number of combinations, and where *k* is constant. The number of combinations can be computed by using the COMB function. On the first call, the argument types and lengths are checked for consistency. On each subsequent call, the values of two variables are interchanged.

If you call the ALLCOMB routine with the first argument out of sequence, the results are not useful. In particular, if you initialize the variables and then immediately call ALLCOMB with a first argument of *j*, then you do not get the *j*th combination (except when *j* is 1). To get the *j*th combination, you must call ALLCOMB *j* times. The first argument takes values from 1 through *j* in that exact order.

Using the CALL ALLCOMB Routine with Macros

You can call the ALLCOMB routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL ALLCOMB routine, both of the following values are specified:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR is set to 0, and &SYSINFO is set to one of the following values:

- 0 if *count*=1
- *j* if the values of *variable-j* and *variable-k* were interchanged, where *j*<*k*

- –1 if no values were interchanged because all distinct combinations were already generated

Comparisons

SAS provides four functions or CALL routines for generating combinations:

- ALLCOMB generates all *possible* combinations of the *values*, *missing* or *nonmissing*, of *n* variables. The values can be any numeric or character value. Each combination is formed from the previous combination by removing one value and inserting another value.
- LEXCOMB generates all *distinct* combinations of the *nonmissing values* of several variables. The values can be any numeric or character value. The combinations are generated in lexicographic order.
- ALLCOMBI generates all combinations of the *indices* of *n* items, where *indices* are integers from 1 to *n*. Each combination is formed from the previous combination by removing one index and inserting another index.
- LEXCOMBI generates all combinations of the *indices* of *n* items, where *indices* are integers from 1 to *n*. The combinations are generated in lexicographic order.

ALLCOMBI is the fastest of these functions and CALL routines. LEXCOMB is the slowest.

Examples

Example 1: Using CALL ALLCOMB in a DATA Step

The following example shows the CALL ALLCOMB routine that is used with the DATA step.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   n=dim(x);
   k=3;
   ncomb=comb(n, k);
   do j=1 to ncomb+1;
      call allcomb(j, k, of x[*]);
      put j 5. +3 x1-x3;
   end;
run;
```

SAS writes the following results to the log:

```
 1   ant bee cat
 2   ant bee ewe
 3   ant bee dog
 4   ant cat dog
 5   ant cat ewe
 6   ant dog ewe
 7   bee dog ewe
 8   bee dog cat
 9   bee ewe cat
10   dog ewe cat
11   dog ewe cat
```

Example 2: Using CALL ALLCOMB with Macros and Displaying the Return Code

Here is an example of the CALL ALLCOMB routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=2;
   %let ncomb=%sysfunc(comb(5, &k));
   %do j=1 %to &ncomb+1;
      %syscall allcomb(j, k, x1, x2, x3, x4, x5);
      %let jfmt=%qsysfunc(putn(&j, 5.));
      %let pad=%qsysfunc(repeat(%str( ),30-%length(&x1 &x2)));
      %put &jfmt:  &x1 &x2 &pad sysinfo=&sysinfo;
   %end;
%mend;

%test
```

SAS writes the following results to the log:

```
 1:  ant -0.1234                                sysinfo=0
 2:  ant zebra                                    sysinfo=2
 3:  ant hippopotamus                         sysinfo=2
 4:  ant 10000000000                         sysinfo=2
 5:  -0.1234 10000000000                   sysinfo=1
 6:  -0.1234 zebra                                 sysinfo=2
 7:  -0.1234 hippopotamus                  sysinfo=2
 8:  10000000000 hippopotamus        sysinfo=1
 9:  10000000000 zebra                        sysinfo=2
10:  hippopotamus zebra                    sysinfo=1
11:  hippopotamus zebra                    sysinfo=-1
```

See Also

Functions:

- “ALLCOMB Function” on page 142

CALL ALLCOMBI Routine

Generates all combinations of the indices of *n* objects taken *k* at a time in a minimal change order.

Category: Combinatorial

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL ALLCOMBI(*n*, *k*, *index-1*, ..., *index-k*, <, *index-added*, *index-removed*>);

Required Arguments

n
is a numeric constant, variable, or expression that specifies the total number of objects.

k
is a numeric constant, variable, or expression that specifies the number of objects in each combination.

index
is a numeric variable that contains indices of the objects in the returned combination. Indices are integers between 1 and n, inclusive.

Tip If *index-1* is missing or 0, ALLCOMBI initializes the indices to `index-1=1` through `index-k=k`. Otherwise, ALLCOMBI creates a new combination by removing one index from the combination and adding another index.

Optional Arguments

index-added
is a numeric variable in which ALLCOMBI returns the value of the index that was added.

index-removed
is a numeric variable in which ALLCOMBI returns the value of the index that was removed.

Details

CALL ALLCOMBI Processing

Before you make the first call to ALLCOMBI, complete one of these tasks:

- Set *index-1* equal to 0 or to a missing value.
- Initialize *index-1* through *index-k* to distinct integers between 1 and n, inclusive.

The number of combinations of n objects taken k at a time can be computed as COMB(*n*, *k*). To generate all combinations of n objects taken k at a time, call ALLCOMBI in a loop that executes COMB(*n*, *k*) times.

Using the CALL ALLCOMBI Routine with Macros

If you call ALLCOMBI from the macro processor with %SYSCALL, you must initialize all arguments to numeric values. &SYSCALL reformats the values that are returned.

If an error occurs during the execution of the CALL ALLCOMBI routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR and &SYSINFO are set to 0.

Comparisons

The CALL ALLCOMBI routine generates all combinations of the indices of n objects taken k at a time in a minimal change order. The CALL ALLCOMB routine generates all combinations of the values of n variables taken k at a time in a minimal change order.

Examples

Example 1: Using CALL ALLCOMBI in a DATA Step

The following example shows the CALL ALLCOMBI routine that is used in a DATA step.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   array c[3] $3;
   array i[3];
   n=dim(x);
   k=dim(i);
   i[1]=0;
   ncomb=comb(n, k);   /* The one extra call goes back */
   do j=1 to ncomb+1;  /* to the first combination. */
```

```
      call allcombi(n, k, of i[*], add, remove);
      do h=1 to k;
         c[h]=x[i[h]];
      end;
      put @4 j= @10 'i= ' i[*] +3 'c= ' c[*] +3 add= remove=;
   end;
run;
```

SAS writes the following results to the log:

```
j=1   i= 1 2 3    c= ant bee cat    add=0 remove=0
j=2   i= 1 3 4    c= ant cat dog    add=4 remove=2
j=3   i= 2 3 4    c= bee cat dog    add=2 remove=1
j=4   i= 1 2 4    c= ant bee dog    add=1 remove=3
j=5   i= 1 4 5    c= ant dog ewe    add=5 remove=2
j=6   i= 2 4 5    c= bee dog ewe    add=2 remove=1
j=7   i= 3 4 5    c= cat dog ewe    add=3 remove=2
j=8   i= 1 3 5    c= ant cat ewe    add=1 remove=4
j=9   i= 2 3 5    c= bee cat ewe    add=2 remove=1
j=10  i= 1 2 5    c= ant bee ewe    add=1 remove=3
j=11  i= 1 2 3    c= ant bee cat    add=3 remove=5
```

Example 2: Using CALL ALLCOMBI with Macros

Here is an example of the CALL ALLCOMBI routine that is used with macros.

```
%macro test;
   %let x1=0;
   %let x2=0;
   %let x3=0;
   %let add=0;
   %let remove=0;
   %let n=5;
   %let k=3;
   %let ncomb=%sysfunc(comb(&n, &k));
   %do j=1 %to &ncomb;
      %syscall allcombi(n, k, x1, x2, x3, add, remove);
      %let jfmt=%qsysfunc(putn(&j, 5.));
      %put &jfmt: &x1 &x2 &x3 add=&add remove=&remove;
   %end;
%mend;
%test
```

SAS writes the following results to the log:

```
 1: 1 2 3 add=0 remove=0
 2: 1 3 4 add=4 remove=2
 3: 2 3 4 add=2 remove=1
 4: 1 2 4 add=1 remove=3
 5: 1 4 5 add=5 remove=2
 6: 2 4 5 add=2 remove=1
 7: 3 4 5 add=3 remove=2
 8: 1 3 5 add=1 remove=4
 9: 2 3 5 add=2 remove=1
10: 1 2 5 add=1 remove=3
```

See Also

CALL Routines:

- "CALL ALLCOMB Routine" on page 215

CALL ALLPERM Routine

Generates all permutations of the values of several variables in a minimal change order.

Category: Combinatorial

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL ALLPERM(*count*, *variable-1* <, *variable-2 ... variable-n*>);

Required Arguments

count
specifies an integer variable that ranges from 1 to the number of permutations.

variable
specifies either all numeric variables or all character variables that have the same length. The values of these variables are permuted.

Restriction: Specify no more than 18 variables.

Requirement: Initialize these variables before you call the ALLPERM routine.

Details

CALL ALLPERM Processing

Use the CALL ALLPERM routine in a loop where the first argument to CALL ALLPERM takes each integral value from 1 to the number of permutations. On the first call, the argument types and lengths are checked for consistency. On each subsequent call, the values of two consecutive variables are interchanged.

Note: You can compute the number of permutations by using the PERM function. For more information, see PERM Function on page 1217.

If you call the ALLPERM routine and the first argument is out of sequence, the results are not useful. In particular, if you initialize the variables and then immediately call the ALLPERM routine with a first argument of K, your result is not the Kth permutation (except when K is 1). To get the Kth permutation, you must call the ALLPERM routine K times. The first argument takes values from 1 through K in that exact order.

ALLPERM always produces N! permutations even if some of the variables have equal values or missing values. If you want to generate only the distinct permutations when there are equal values, or if you want to omit missing values from the permutations, use the LEXPERM function instead.

Using the CALL ALLPERM Routine with Macros

You can call the ALLPERM routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL ALLPERM routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR is set to 0, and &SYSINFO is set to one of the following values:

- 0 if *count*=1
- J if 1<*count*<=N! and the values of *variable-J* and *variable-K* were interchanged, where J+1=K
- –1 if *count*>N!

Comparisons

SAS provides three functions or CALL routines for generating all permutations:

- ALLPERM generates all possible permutations of the values, missing or nonmissing, of several variables. Each permutation is formed from the previous permutation by interchanging two consecutive values.
- LEXPERM generates all distinct permutations of the nonmissing values of several variables. The permutations are generated in lexicographic order.
- LEXPERK generates all distinct permutations of K of the nonmissing values of N variables. The permutations are generated in lexicographic order.

ALLPERM is the fastest of these functions and CALL routines. LEXPERK is the slowest.

Examples

Example 1: Using CALL ALLPERM in a DATA Step

This example generates permutations of given values by using the CALL ALLPERM routine.

```
data _null_;
   array x [4] $3 ('ant' 'bee' 'cat' 'dog');
   n=dim(x);
   nfact=fact(n);
   do i=1 to nfact;
      call allperm(i, of x[*]);
      put i 5. +2 x[*];
   end;
run;
```

SAS writes the following results to the log:

```
 1  ant bee cat dog
 2  ant bee dog cat
 3  ant dog bee cat
 4  dog ant bee cat
 5  dog ant cat bee
 6  ant dog cat bee
 7  ant cat dog bee
 8  ant cat bee dog
 9  cat ant bee dog
10  cat ant dog bee
11  cat dog ant bee
12  dog cat ant bee
13  dog cat bee ant
14  cat dog bee ant
15  cat bee dog ant
16  cat bee ant dog
17  bee cat ant dog
18  bee cat dog ant
19  bee dog cat ant
20  dog bee cat ant
21  dog bee ant cat
22  bee dog ant cat
23  bee ant dog cat
24  bee ant cat dog
```

Example 2: Using CALL ALLPERM with Macros

Here is an example of the CALL ALLPERM routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let nperm=%sysfunc(perm(4));
   %do j=1 %to &nperm+1;
      %syscall allperm(j, x1, x2, x3, x4);
      %let jfmt=%qsysfunc(putn(&j, 5.));
      %put &jfmt:   &x1 &x2 &x3 &x4 sysinfo=&sysinfo;
```

```
   %end;
%mend;

%test;
```

SAS writes the following results to the log:

```
 1:   ant -0.1234 10000000000 hippopotamus sysinfo=0
 2:   ant -0.1234 hippopotamus 10000000000 sysinfo=3
 3:   ant hippopotamus -0.1234 10000000000 sysinfo=2
 4:   hippopotamus ant -0.1234 10000000000 sysinfo=1
 5:   hippopotamus ant 10000000000 -0.1234 sysinfo=3
 6:   ant hippopotamus 10000000000 -0.1234 sysinfo=1
 7:   ant 10000000000 hippopotamus -0.1234 sysinfo=2
 8:   ant 10000000000 -0.1234 hippopotamus sysinfo=3
 9:   10000000000 ant -0.1234 hippopotamus sysinfo=1
10:   10000000000 ant hippopotamus -0.1234 sysinfo=3
11:   10000000000 hippopotamus ant -0.1234 sysinfo=2
12:   hippopotamus 10000000000 ant -0.1234 sysinfo=1
13:   hippopotamus 10000000000 -0.1234 ant sysinfo=3
14:   10000000000 hippopotamus -0.1234 ant sysinfo=1
15:   10000000000 -0.1234 hippopotamus ant sysinfo=2
16:   10000000000 -0.1234 ant hippopotamus sysinfo=3
17:   -0.1234 10000000000 ant hippopotamus sysinfo=1
18:   -0.1234 10000000000 hippopotamus ant sysinfo=3
19:   -0.1234 hippopotamus 10000000000 ant sysinfo=2
20:   hippopotamus -0.1234 10000000000 ant sysinfo=1
21:   hippopotamus -0.1234 ant 10000000000 sysinfo=3
22:   -0.1234 hippopotamus ant 10000000000 sysinfo=1
23:   -0.1234 ant hippopotamus 10000000000 sysinfo=2
24:   -0.1234 ant 10000000000 hippopotamus sysinfo=3
25:   -0.1234 ant 10000000000 hippopotamus sysinfo=-1
```

See Also

Functions:

- “ALLPERM Function” on page 144
- “LEXPERM Function” on page 1065

CALL Routines:

- “CALL RANPERK Routine” on page 312
- “CALL RANPERM Routine” on page 314

CALL CATS Routine

Removes leading and trailing blanks, and returns a concatenated character string.

Category: Character

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL CATS(*result* <, *item-1*, ..., *item-n*>);

Required Argument

result
specifies a character variable.

Restriction The CALL CATS routine accepts only a character variable as a valid argument for *result*. Do not use a constant or a SAS expression because CALL CATS is unable to update these arguments.

Optional Argument

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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.

Details

The CALL CATS routine returns the result in the first argument, *result*. The routine appends the values of the arguments that follow to *result*. If the length of *result* is not large enough to contain the entire result, SAS performs these actions:

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function CALL and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause

The CALL CATS routine removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a separator such as a blank or comma.

CALL Routine	Equivalent Statement
`CALL CATS(OF X1-X4);`	`X1=TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))||TRIM(LEFT(X4));`
`CALL CATT(OF X1-X4);`	`X1=TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4);`
`CALL CATX(SP, OF X1-X4); *`	`X1=TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP||TRIM(LEFT(X3))||SP||TRIM(LEFT(X4));`

Note: If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding separator. For example, `CALL CATX("+","X"," ", "Z"," ");` produces `X+Z`.

Example

This example shows how the CALL CATS routine concatenates strings.

```
data _null_;
   length answer $ 37;
   x='Athens is t  ';
   y=' he Olym   ';
   z='   pics site for 2004. ';
   call cats(answer, x, y, z);
   put answer;
run;
```

SAS writes the following result to the log:

```
Athens is the Olympics site for 2004.
```

See Also

Functions:

- “CAT Function” on page 375
- “CATQ Function” on page 378
- “CATS Function” on page 383
- “CATT Function” on page 385
- “CATX Function” on page 388

CALL Routines:

- "CALL CATT Routine" on page 228
- "CALL CATX Routine" on page 230

CALL CATT Routine

Removes trailing blanks and returns a concatenated character string.

Category: Character

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL CATT(*result* <, *item-1*, ..., *item-n*>);

Required Argument

result
specifies a character variable.

Restriction The CALL CATT routine accepts only a character variable as a valid argument for *result*. Do not use a constant or a SAS expression because CALL CATT is unable to update these arguments.

Optional Argument

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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

The CALL CATT routine returns the result in the first argument, *result*. The routine appends the values of the arguments that follow to *result*. If the length of *result* is not large enough to contain the entire result, SAS performs these actions:

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function CALL and lists the argument that caused the truncation, except in SQL or in a WHERE clause

- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause

The CALL CATT routine removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a separator such as a blank or comma.

CALL Routine	Equivalent Statement
`CALL CATS(OF X1-X4);`	`X1=TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4));`
`CALL CATT(OF X1-X4);`	`X1=TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4);`
`CALL CATX(SP, OF X1-X4); *`	`X1=TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4));`

Note: If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding separator. For example, `CALL CATX("+","X"," ", "Z"," ");` produces `X+Z`.

Example

This example shows how the CALL CATT routine concatenates strings.

```
data _null_;
   length answer $ 37;
   x=Tokyo is t  ';
   y='he Olym   ';
   z='pics site for 2020. ';
   call catt(answer, x, y, z);
   put answer;
run;
```

The preceding statements produce this result:

```
Tokyo is the Olympics site for 2020.
```

See Also

Functions:

- “CAT Function” on page 375
- “CATQ Function” on page 378
- “CATS Function” on page 383
- “CATT Function” on page 385
- “CATX Function” on page 388

CALL Routines:

- “CALL CATS Routine” on page 225
- “CALL CATX Routine” on page 230

CALL CATX Routine

Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.

Category: Character

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL CATX(*delimiter*, *result* <, *item-1*, ..., *item-n*>);

Required Arguments

delimiter
specifies a character string that is used as a delimiter between concatenated strings.

result
specifies a character variable.

Restriction The CALL CATX routine accepts only a character variable as a valid argument for *result*. Do not use a constant or a SAS expression because CALL CATX is unable to update these arguments.

Optional Argument

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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.

Details

The CALL CATX routine returns the result in the second argument, *result*. The routine appends the values of the arguments that follow to *result*. If the length of *result* is not large enough to contain the entire result, SAS performs these actions:

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function CALL and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause

The CALL CATX routine removes leading and trailing blanks from numeric arguments after formatting the numeric value with the BEST*w*. format.

Comparisons

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a delimiter such as a blank or comma.

CALL Routine	Equivalent Statement
`CALL CATS(OF X1-X4);`	`X1=TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4));`
`CALL CATT(OF X1-X4);`	`X1=TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4);`
`CALL CATX(SP, OF X1-X4); *`	`X1=TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4));`

Note: If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding delimiter. For example, `CALL CATX("+",newvar,"X"," ", "Z"," ");` produces `X+Z`.

Example

This example shows how the CALL CATX routine concatenates strings.

```
data _null_;
   length answer $ 50;
   separator='%%$%%';
   x='Athens is t  ';
   y='he Olym     ';
   z='  pics site for 2004.  ';
   call catx(separator, answer, x, y, z);
   put answer;
run;
```

SAS writes the following result to the log:

```
Athens is t%%$%%he Olym%%$%%pics site for 2004.
```

See Also

Functions:

- “CAT Function” on page 375
- “CATQ Function” on page 378
- “CATS Function” on page 383
- “CATT Function” on page 385
- “CATX Function” on page 388

CALL Routines:

- “CALL CATS Routine” on page 225
- “CALL CATT Routine” on page 228

CALL COMPCOST Routine

Sets the costs of operations for later use by the COMPGED function

Category:	Character
Restriction:	Use with the COMPGED function
Interaction:	When invoked by the %SYSCALL macro statement, CALL COMPCOST removes quotation marks from its arguments. For more information, see “Invoking CALL Routines and the %SYSCALL Macro Statement” on page 12.
Note:	Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL COMPCOST(*operation-1*, *value-1* <, *operation-2*, *value-2* ... *operation-n*, *value-n*>);

Required 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 458 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
BLANK=	very large
DELETE=	100
DOUBLE=	very large
FDELETE=	equal to DELETE
FINSERT=	equal to INSERT
FREPLACE=	equal to REPLACE

Operation	Default Cost
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 call to the COMPCOST routine, 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 calls the COMPCOST routine, the costs that are specified remain in effect until your program calls the COMPCOST routine again, or until the step that contains the call to COMPCOST terminates.

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.

```
options pageno=1 nodate linesize=80 pagesize=60;
data test;
   length String $8 Operation $40;
   if _n_ = 1 then call compcost('insert=', 10, 'DEL=', 11, 'r=', 12);
   input String Operation;
   GED=compged(string, 'baboon');
   datalines;
baboon  match
xbaboon insert
babon   delete
baXoon  replace
```

```
;
proc print data=test label;
   label GED='Generalized Edit Distance';
   var String Operation GED;
run;
```

The following output shows the results.

Output 3.1 *Generalized Edit Distance Based on Operation*

The SAS System

Obs	String	Operation	Generalized Edit Distance
1	baboon	match	0
2	xbaboon	insert	10
3	babon	delete	11
4	baXoon	replace	12

See Also

Functions:

- “COMPARE Function” on page 447
- “COMPGED Function” on page 456
- “COMPLEV Function” on page 463

CALL EXECUTE Routine

Resolves the argument, and issues the resolved value for execution at the next step boundary.

Category: Macro

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

System option SOURCE controls whether the code that is passed to the CALL EXECUTE routine is written to the SAS log. When SOURCE is in effect, the code is written to the log. Otherwise, the code is suppressed. For batch jobs, the default typically is SOURCE. Use the OPTIONS procedure to see the current setting and an OPTIONS statement to change it:

```
proc options option=source; run; /* Print current setting */
options nosource;                /* Do not write source to the SAS log */
```

See: See "CALL EXECUTE Routine" in *SAS Macro Language Reference for SAS Viya Workbench*

Syntax

CALL EXECUTE(*argument*);

Required Argument

argument
specifies a character expression or a constant that yields a macro invocation or a SAS statement. *Argument* can be:

- a character string, enclosed in quotation marks.
- the name of a DATA step character variable. Do not enclose the name of the DATA step variable in quotation marks.
- a character expression that the DATA step resolves to a macro text expression or a SAS statement.

Details

If *argument* resolves to a macro invocation, the macro executes immediately and DATA step execution pauses while the macro executes. If *argument* resolves to a SAS statement or if execution of the macro generates SAS statements, the statement(s) execute after the end of the DATA step that contains the CALL EXECUTE routine. CALL EXECUTE is fully documented in *SAS Macro Language: Reference*.

CALL GRAYCODE Routine

Generates all subsets of *n* items in a minimal change order.

Category: Combinatorial

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL GRAYCODE(*k*, *numeric-variable-1*, ..., *numeric-variable-n*);

CALL GRAYCODE(*k*, *character-variable* <, *n* <, *in-out*> >);

Required Arguments

k

specifies a numeric variable. Initialize *k* to either of these values before executing the CALL GRAYCODE routine:

- a negative number, which causes CALL GRAYCODE to initialize the subset to be empty
- the number of items in the initial set indicated by *numeric-variable-1* through *numeric-variable-n* or *character-variable*, which must be an integer value between 0 and N, inclusive

The value of *k* is updated when CALL GRAYCODE is executed. The value that is returned is the number of items in the subset.

numeric-variable

specifies numeric variables that have values of 0 or 1 that are updated when CALL GRAYCODE is executed. A value of 1 for *numeric-variable-j* indicates that the *j*th item is in the subset. A value of 0 for *numeric-variable-j* indicates that the *j*th item is not in the subset.

If you assign a negative value to *k* before you execute CALL GRAYCODE, you do not need to initialize *numeric-variable-1* through *numeric-variable-n* before executing CALL GRAYCODE unless you want to suppress the note about uninitialized variables.

If you assign a value between 0 and *n* inclusive to *k* before you execute CALL GRAYCODE, you must initialize *numeric-variable-1* through *numeric-variable-n* to *k* values of 1 and *n-k* values of 0.

character-variable

specifies a character variable that has a length of at least *n* characters. The first *n* characters indicate which items are in the subset. By default, an "I" in the *j*th position indicates that the *j*th item is in the subset, and an "O" in the *j*th position indicates that the *j*th item is out of the subset. You can change the two characters by specifying the *in-out* argument.

If you assign a negative value to *k* before you execute CALL GRAYCODE, you do not need to initialize *character-variable* before executing CALL GRAYCODE unless you want to suppress the note about an uninitialized variable.

If you assign a value between 0 and *n* inclusive to *k* before you execute CALL GRAYCODE, you must initialize *character-variable* to *k* characters that indicate an item is in the subset, and *k-k* characters that indicate an item is out of the subset.

Optional Arguments

n

specifies a numeric constant, variable, or expression. By default, *n* is the length of *character-variable*.

in-out

specifies a character constant, variable, or expression. The default value is "IO." The first character is used to indicate that an item is in the subset. The second character is used to indicate that an item is out of the subset.

Details

Using CALL GRAYCODE in a DATA Step

When you execute the CALL GRAYCODE routine with a negative value of *k*, the subset is initialized to be empty.

When you execute the CALL GRAYCODE routine with an integer value of *k* between 0 and *n* inclusive, one item is either added to the subset or removed from the subset, and the value of *k* is updated to equal the number of items in the subset.

To generate all subsets of *n* items, you can initialize *k* to a negative value and execute CALL GRAYCODE in a loop that iterates 2**n* times. If you want to start with a nonempty subset, initialize *k* to be the number of items in the subset, initialize the other arguments to specify the desired initial subset, and execute CALL GRAYCODE in a loop that iterates 2**n*-1 times. The sequence of subsets that are generated by CALL GRAYCODE is cyclical, so you can begin with any subset that you want.

Using the CALL GRAYCODE Routine with Macros

You can call the GRAYCODE routine when you use the %SYSCALL macro. Differences exist when you use CALL GRAYCODE in a DATA step and when you use the routine with macros. This list describes using CALL GRAYCODE with macros:

- All arguments must be initialized to non-blank values.
- If you use the *character-variable* argument, it must be initialized to a non-blank, nonnumeric character string that contains at least *n* characters.
- If you use the *in-out* argument, it must be initialized to a string that contains two characters that are not blanks, digits, decimal points, or plus and minus signs.

If %SYSCALL identifies an argument as being the wrong type, or if %SYSCALL is unable to identify the type of argument, &SYSERR and &SYSINFO are *not* set.

Otherwise, if an error occurs during the execution of the CALL GRAYCODE routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR is set to 0o, and &SYSINFO is set to one of these values:

- 0 if the value of *k* on input is negative
- the index of the item that was added or removed from the subset if the value of *k* on input is a valid nonnegative integer.

Examples

Example 1: Using a Character Variable and Positive Initial k with CALL GRAYCODE

This example uses the CALL GRAYCODE routine to generate subsets in a minimal change order.

```
data _null_;
   x='++++';
   n=length(x);
   k=countc(x, '+');
   put '    1' +3 k= +2 x=;
   nsubs=2**n;
   do i=2 to nsubs;
      call graycode(k, x, n, '+-');
      put i 5. +3 k= +2 x=;
   end;
run;
```

SAS writes the following results to the log:

```
 1   k=4   x=++++
 2   k=3   x=-+++
 3   k=2   x=-+-+
 4   k=3   x=++-+
 5   k=2   x=+--+
 6   k=1   x=---+
 7   k=0   x=----
 8   k=1   x=+---
 9   k=2   x=++--
10   k=1   x=-+--
11   k=2   x=-++-
12   k=3   x=+++-
13   k=2   x=+-+-
14   k=1   x=--+-
15   k=2   x=--++
16   k=3   x=+-++
```

Example 2: Using %SYSCALL with Numeric Variables and Negative k

This example uses the %SYSCALL macro with numeric variables to generate subsets in a minimal change order.

```
%macro test;
   %let n=3;
   %let x1=.;
   %let x2=.;
   %let x3=.;
   %let k=-1;
   %let nsubs=%eval(2**&n + 1);
   %put nsubs=&nsubs k=&k x: &x1 &x2 &x3;
   %do j=1 %to &nsubs;
      %syscall graycode(k, x1, x2, x3);
      %put &j: k=&k x: &x1 &x2 &x3 sysinfo=&sysinfo;
   %end;
```

```
%mend;
%test;
```

SAS writes the following results to the log:

```
nsubs=9 k=-1 x: . . .
1: k=0 x: 0 0 0 sysinfo=0
2: k=1 x: 1 0 0 sysinfo=1
3: k=2 x: 1 1 0 sysinfo=2
4: k=1 x: 0 1 0 sysinfo=1
5: k=2 x: 0 1 1 sysinfo=3
6: k=3 x: 1 1 1 sysinfo=1
7: k=2 x: 1 0 1 sysinfo=2
8: k=1 x: 0 0 1 sysinfo=1
9: k=0 x: 0 0 0 sysinfo=3
```

Example 3: Using %SYSCALL with a Character Variable and Negative k

This example uses the %SYSCALL macro with a character variable to generate subsets in a minimal change order.

```
%macro test(n);
   %*** Initialize the character variable to a
        sufficiently long nonblank, nonnumeric value. ;
   %let x=%sysfunc(repeat(_, &n-1));
   %let k=-1;
   %let nsubs=%eval(2**&n + 1);
   %put nsubs=&nsubs k=&k x="&x";
   %do j=1 %to &nsubs;
      %syscall graycode(k, x, n);
      %put &j: k=&k x="&x" sysinfo=&sysinfo;
   %end;
%mend;
%test(3);
```

SAS writes the following results to the log:

```
nsubs=9 k=-1 x="___"
1: k=0 x="OOO" sysinfo=0
2: k=1 x="IOO" sysinfo=1
3: k=2 x="IIO" sysinfo=2
4: k=1 x="OIO" sysinfo=1
5: k=2 x="OII" sysinfo=3
6: k=3 x="III" sysinfo=1
7: k=2 x="IOI" sysinfo=2
8: k=1 x="OOI" sysinfo=1
9: k=0 x="OOO" sysinfo=3
```

See Also

Functions:

- “GRAYCODE Function” on page 886

CALL IS8601_CONVERT Routine

Converts an ISO 8601 interval to datetime and duration values, or converts datetime and duration values to an ISO 8601 interval.

Category: Date and Time

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL IS8601_CONVERT(*convert-from*, *convert-to*, <*from-variables*>, <*to-variables*>,
<*date-time-replacements*>);

Required Arguments

convert-from
specifies a keyword in single quotation marks that indicates the source for the conversion, such as a date, a datetime and duration, a duration, or an interval value. *Convert-from* can have one of these values:

'd*n*'
specifies a date value, where *n* is a value from 1 to 6. The default value is 1. *N* is the number of components in the *from-variables* or *to-variables* arguments. These components are valid:

- year
- month
- day
- hour
- minute
- second

'dt*n*'
specifies a datetime value, where *n* is a value from 1 to 6. The default value is 1. *N* is the number of components in the *from-variables* or *to-variables* arguments. These components are valid:

- year
- month
- day
- hour

- minute
- second

'dt/dt'
specifies that the source value for the conversion is a datetime/datetime value.

'dt/du'
specifies that the source value for the conversion is a datetime/duration value.

'du*n*'
specifies that the source value for the conversion is a duration value, where *n* is a value from 1 to 6. The default value is 1. *N* is the number of components in the *from-variables* or *to-variables* arguments. These components are valid:

- year
- month
- day
- hour
- minute
- second

'du/dt'
specifies that the source value for the conversion is a duration/datetime value.

'intvl'
specifies that the source value for the conversion is an interval value.

convert-to
specifies a keyword in single quotation marks that indicates the results of the conversion. *Convert-to* can have one of these values:

'd*n*'
specifies a date value, where *n* is a value from 1 to 6. The default value is 1. *N* is the number of components in the *from-variables* or *to-variables* arguments. These components are valid:

- year
- month
- day
- hour
- minute
- second

'dt*n*'
specifies a datetime value, where *n* is a value from 1 to 6. The default value is 1. *N* is the number of components in the *from-variables* or *to-variables* arguments. These components are valid:

- year

- month
- day
- hour
- minute
- second

'dt/dt'
specifies to create a datetime/datetime interval.

'dt/du'
specifies to create a datetime/duration interval.

'du*n*'
specifies to create a duration, where *n* is a value from 1 to 6. The default value is 1. *N* is the number of components in the *from-variables* or *to-variables* arguments. These components are valid:

- year
- month
- day
- hour
- minute
- second

'du/dt'
specifies to create a duration/datetime interval.

'end'
specifies to create a value that is the ending datetime or duration of an interval value.

'intvl'
specifies to create an interval value.

'start'
specifies to create a value that is the beginning datetime or duration of an interval value.

Optional Arguments

from-variable
specifies one or two variables that contain the source value. Specify one variable for an interval value, and specify two variables, one each for datetime and duration values. The datetime and duration values are interval components where the first value is the beginning value of the interval and the second value is the ending value of the interval.

Requirements An integer variable must be at least a 16-byte character variable whose value is read by the $N8601B*w.d* informat or the $N8601E*w.d* informat. The integer variable can also be an integer

value that is returned by invoking the CALL IS8601_CONVERT routine.

A datetime value must be a SAS datetime value or an 8-byte character value that is read by the $N8601B*w.d* informat or the $N8601E*w.d* informat, or by invoking the CALL IS8601_CONVERT routine.

A duration value must be a numeric value that represents the number of seconds in the duration, or an 8-byte character value whose value is read by the $N8601B*w.d* informat or the $N8601E*w.d* informat, or by invoking the CALL IS8601_CONVERT routine.

to-variable

specifies one or two variables that contain converted values. Specify one variable for an interval value, and specify two variables, one each, for datetime and duration values.

Requirement An interval variable must be at least a 16-byte character variable.

Tip The datetime and duration variables can be numeric or character. To avoid losing precision of a numeric value, the length of a numeric variable must be at least eight characters. Datetime and duration character variables must be at least 16 bytes; they are padded with blank characters for values that are less than the length of the variable.

date-time-replacements

specifies date or time component values to use when a month, day, or time component is omitted from an interval, datetime, or duration value. The *date-time-replacements* argument is specified as a series of numbers that are separated by commas and represent components in this order: year, month, day, hour, minute, and second. Components of *date-time-replacements* can be omitted only in the reverse order: second, minute, hour, day, month, and year. If no substitute values are specified, the conversion is performed using default values.

Default The following default values for date and time components are omitted:

1	month
1	day
0	hour
0	minute
0	second

Requirement A year component must be part of the datetime or duration value and therefore is not valid in *date-time-replacements*. A comma is required as a placeholder for the year in *date-time-replacements*. For example, in the replacement value string, `,9,4,,2,'`, the first comma is a placeholder for a year value.

Details

The Basics

ISO 8601 representations of date, time, and datetime values within the ISO 8601 standards consist of basic and extended notations. A value is basic when delimiters that separate the various components within a value are omitted. A value is extended when delimiters are used to separate those components. A set of formats and informats is used with ISO 8601 date, time, and datetime values. For more information, see "Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats for SAS Viya Workbench* and "Reading Dates and Times by Using the ISO 8601 Basic and Extended Notations" in *SAS Formats and Informats for SAS Viya Workbench*.

After your values are stored as SAS variables, you can calculate intervals, durations, and datetime values with the CALL IS8601_CONVERT routine. This routine enables you to convert an ISO 8601 interval to datetime and duration values, or to convert datetime and duration values to an ISO 8601 interval. The CALL IS8601_CONVERT routine accepts missing values in the d*n*, dt*n*, and du*n* arguments. You can also use the CALL IS8601_CONVERT routine to perform calculations with dates and times.

You can use the year, month, day, hour, minute, and second components in any order.

When only a duration is passed to the CALL routine, the following rules apply in determining the duration values. The total days in a month vary: January has 31, Feburary has 28.... Also leap-year-day totals must be considered.

Start day day in the first date

Endday

How Arguments in the CALL IS8601_CONVERT Routine Are Used

The first argument to the CALL IS8601_CONVERT routine, *convert-from*, can be one or two values. The number of values is based on how many variables you provide to the routine for an expected result. For example, datetime and duration values can be specified with an expected output of an interval. In this example, the supplied first argument would be `'dt/du'`. If two datetime values are supplied with an expected duration as output, the first argument would be `'dt/dt'`.

The second argument to the CALL IS8601_CONVERT routine, *convert-to*, can also be one or two values. The values depend on the type of result that you expect SAS to compute using the input that you supply in the first argument.

Note: The minimum length for durations is 16, and the minimum length for intervals is 32.

Calculating The Duration Values

These rules apply to determining duration values. The months in a year have different totals: January has 31 days, February has 28 days and so forth. Also leap years must be calculated in determining duration values.

Examples

Example 1: Using the CALL IS8601_CONVERT Routine

The following DATA step uses the CALL IS8601_CONVERT routine to perform these tasks:

- create an interval by using datetime and duration values
- create datetime and duration values from an interval that was created using the CALL IS8601_CONVERT routine
- create an interval from datetime and duration values by using replacement values for omitted date and time components in the datetime value

For easier reading, numeric variables end with an N and character variables end with a C.

```
data _null_;
      /* Declare variable length and type. Character datetime and
duration   */
      /* values must be at least 16 characters. To avoid losing
precision,   */
      /* the numeric datetime value has a length of
8.                        */
   length dtN duN 8 dtC duC $16 intervalC $32;
      /* Assign a numeric datetime value and a character duration
value.     */
   dtN='15Sep2011:09:00:00'dt;
   duC=input('P2y3m4dT5h6m7s', $n8601b.);
   put dtN=;
   put duC=;
      /* Create an interval from a datetime and duration value and
format the */
      /* interval using the ISO 8601 extended notation for character
values.  */
   call is8601_convert('dt/du', 'intvl', dtN, duC, intervalC);
   put '**  Character interval created from datetime and duration
values **';
   put intervalC $n8601e.;
   put ' ';
      /* Create numeric datetime and duration values from an interval
and     */
      /* format the values using the ISO 8601 extended notation for
numeric   */
      /*
values.                                                              */
   call is8601_convert('intvl', 'dt/du', intervalC, dtN, duN);
```

```
   put '**  Character datetime and duration created from an interval
**/';
   put dtN=;
   put duN=;
   put ' ';
      /* Assign a new datetime value with omitted
components.                  */
   dtC=input('2012---15T10:-:-', $n8601b.);
   put '** This datetime is a character value. **';
   put dtC $n8601h.;
   put ' ';
      /* Create an interval by reading a datetime value that has
omitted date  */
      /* and time components.  Use replacement values for the month,
minutes,  */
      /* and
seconds.                                                          */
   call is8601_convert('du/dt', 'intvl', duC, dtC, intervalC, , 7, , ,
35, 45);
   put '**  Interval created using a datetime with omitted
values,      **';
   put '**  inserting replacement values for month (7), minute
(35)     **';
   put '**  seconds
(45).                                              **';
   put intervalC $n8601e.;
   put ' ';
run;
```

SAS writes the following results to the log:

```
dtN=1631696400
duC=0002304050607FFC
**  Character interval created from datetime and duration values **/
2011-09-15T09:00:00.000/P2Y3M4DT5H6M7S

**  Character datetime and duration created from an interval  **/
dtN=1631696400
duN=71211967

** This datetime is a character value. **
2012---15T10:-:-

**  Interval created using a datetime with omitted values,      **
**  inserting replacement values for month (7), minute (35)     **
**  seconds (45).                                               **
P2Y3M4DT5H6M7S/2012-07-15T10:35:45
```

Example 2: Passing a Duration to the CALL Routine

When only a duration is passed to the CALL routine, 30 days is used to calculate one month and 365 days is used to calculate one year. In this example, the first two parameters are durations.

```
data b;
input start $;
   CALL IS8601_CONVERT('DU', 'DU', start, endDUR);
   number_of_days=enddur/86400;
  datalines;
```

```
P2M3D
P1Y1M1D
P1M4D
;
run;
proc print;
run;
```

The SAS System

Obs	start	endDUR	number_of_days
1	P2M3D	5443200	63
2	P1Y1M1D	34214400	396
3	P1M4D	2937600	34

Example 3: Finding an Interval Value

This example writes duration and datetime values to the log.

```
data _null_;
   length mynew $32;
   x='P8w';
   y='11feb2012:12:35:22'dt;
   call is8601_convert('du/dt', 'intvl', x, y, mynew);
   put mynew=$n8601e.;
run;
```

SAS writes the following results to the log:

```
mynew=P8W/2012-02-11T12:35:22.000
```

Example 4: Finding the Start Date of an Interval

This example returns the start date of an interval.

```
data temp;
   length dt $32;
   x='P6w';
   y='11feb2012:11:13:22'dt;
   call is8601_convert('du/dt', 'start', x, y, dt);
   put dt=$n8601e.;
run;
```

SAS writes the following results to the log:

```
dt=2011-12-31T00:00:00.000
```

In this example, P6w specifies a six-week duration, and *y* is the date from which *start* is calculated. *start* is the beginning duration of an interval. This datetime value is six weeks before *y*.

Example 5: Finding an Interval Using Duration and Datetime Values

This example uses duration and datetime values to find an interval.

```
data _null_;
   infile datalines;
   input mo d yr hour min sec;
   length mynew $32;
   x='P8w';
   call is8601_convert('du/dt6', 'intvl', x, mo, d, yr, hour, min,
sec, mynew);
   put mynew=$n8601e.;
   datalines;
02 22 2011 10 30 45
12 13 2010 12 35 25
03 26 2010 10 10 10
;
run;
```

SAS writes the following results to the log:

```
mynew=P8W/0002-14-91T10:30:45.000
mynew=P8W/0012-13-90T12:35:25.000
mynew=P8W/0003-14-90T10:10:10.000
```

Example 6: Finding the Start Date for Multiple Intervals

This example uses data lines as input to find the start date for an interval.

```
data temp;
   input y mo d h min s;
   length mynew $16;
      /* This value is the duration. */
   x='P2w';
      /* This CALL routine uses the date and time values that are */
      /* listed in the data lines to create a DATETIME value.     */
   call is8601_convert('dt6', 'dt', y, mo, d, h, min, s, dt);
   put dt=datetime.;
      /* This CALL routine uses the DATETIME value that was previously
*/
      /* created.
*/
   call is8601_convert('du/dt', 'start', x, dt, mynew);
   put mynew=$n8601e.;
datalines;
2011 6 7 10 15 20
2011 12 5 10 25 45
2011 6 30 10 32 20
;
```

SAS writes the following results to the log:

```
dt=07JUN11:10:15:20
mynew=2011-05-24T00:00:00.000
dt=05DEC11:10:25:45
mynew=2011-11-21T00:00:00.000
dt=30JUN11:10:32:20
mynew=2011-06-16T00:00:00.000
```

Example 7: Converting a Duration to a SAS Time Value

In this example, you supply the duration value P8w, which specifies eight weeks. This type of value is often coupled with a datetime value to produce a duration, but the value can be used alone and converted to a SAS time value. There is no SAS informat to convert the P8w value to a SAS time value, but you can use the CALL IS8601_CONVERT routine to convert the value.

```
data a;
   x='P8w';
   call is8601_convert('du', 'du', x, mynew);
   put mynew=time8.;
run;
```

The value of Mynew is 1344:00, which indicates 1344 hours.

In this example, the following statements apply:

- The *X* variable is a duration. Therefore, 'du' is the first argument.
- A time value is expected as output, but there is no *convert-to* value for time. As a result, the value for 'du' is used for duration. Both arguments require single quotation marks.
- *X* is the variable name that is supplied, and Mynew is the variable that is being created.
- Because the result is a SAS time value, you can use a SAS time format (TIME8.).

Example 8: Converting a Duration Value and a Datetime Value to an Interval

This example takes the previous example one step further by using the same duration value with a datetime value to create an interval as output. The following DATA step is based on an event that lasts for eight weeks, ending on February 11, 2012, at 12:22 p.m.

```
data a;
   length mynew $32;
   x='P8w';
   y='11feb2012:12:22'dt;
   call is8601_convert('du/dt', 'intvl', x, y, mynew);
   put mynew=$n8601e.;
run;
```

The value of Mynew is P8w/2012-02-11T12:22:00.000.

In this example, the following statements apply:

- The first argument, 'du/dt', to the CALL IS8601_CONVERT routine indicates the types of variables that are being passed in for conversion. The value 'du/dt' specifies that two variables are being passed: one is a duration value, *X*, and the

other is a datetime value, *Y*. Because the duration value comes before the datetime value, the datetime value is assumed to be the end of the interval.

- The result that you want from this DATA step is an interval. Therefore, the second argument is 'intvl'.
- The remaining arguments name the incoming variables, *X* and *Y*, and the new variable, Mynew, that is being created. The order of these variables must match the types of variables that are specified in the first argument. For example, if *X* and *Y* are reversed, the following note appears in the log:

```
NOTE: Invalid argument to function IS8601_CONVERT('du/dt','intvl',1644582120,
      'P8w',''[12 of 32 characters shown]) at line 770 column 9.
mynew=************************************************
mynew=  x=P8w y=1644582120 _ERROR_=1 _N_=1
```

Example 9: Computing the Start Datetime of an Interval When You Have a Duration and a Datetime

"Example 8: Converting a Duration Value and a Datetime Value to an Interval" on page 250 computes an interval when datetime and duration values are supplied. This example computes the starting datetime for an interval when duration and datetime values are supplied.

```
data a;
   length mynew $16;
      /* If you omit the LENGTH statement, replace the PUT statement */
      /* with 'put mynew datetime22.;'.                              */
   x='P8w';
   y='11feb2012:12:22'dt;
   call is8601_convert('du/dt', 'start', x, y, mynew);
   put mynew=$n8601e.;
run;
```

The value of Mynew is 2011-12-17T00:00:00.000.

In this example, the following statements apply:

- The CALL IS8601_CONVERT routine uses the argument `start` as the second argument in order to compute the starting datetime value for the interval.
- Because the SAS datetime value is computed, you can remove the LENGTH statement so that Mynew is created as a numeric variable. If you remove the LENGTH statement, add a PUT statement that specifies the DATETIME22. format. It is also valid to leave the LENGTH statement and create variable output by using the $N8601E*w*. format, as shown in the second PUT statement.

Example 10: Converting a Duration of One Type to a Duration of a Different Type

If you receive data that is represented in hours, you can perform a conversion similar to Example 9 to obtain output as a duration.

```
data _null_;
   x='1271:59:00';
   time=input(x, time10.);
   length dur $16;
```

```
   call is8601_convert('du', 'du', time, dur);
   put dur=$n8601e.;
run;
```

The value of dur is P0Y1M22DT23H59M0.0S.

In this example, the following statements apply:

- The TIME*w.d* format reads the time value into SAS.
- The CALL IS8601_CONVERT routine converts the time value to a duration value.

Example 11: Converting Two Datetime Values to a Duration

This example determines the amount of time between two datetime values and creates a duration as output.

```
data a;
   length dur $16;
   start='02apr2012:12:30:22'dt;
   end='08apr2012:14:32:22'dt;
   call is8601_convert('dt/dt', 'du', start, end, dur);
   put dur=$n8601e.;
run;
```

The value of dur is P6DT2H2M.

In this example, the following statements apply:

- The CALL IS8601_CONVERT routine uses these arguments:
  - 'dt/dt' indicates that two datetime values are being passed into the function.
  - 'du' indicates that a duration value is the expected outcome.
  - 'start' is the name of the first datetime variable.
  - 'end' is the name of the second datetime variable.
  - 'dur' is the name of the desired output variable.
- A LENGTH statement is required in order to specify that Dur is a character variable and that the length should be `16`. If you create an interval, a length of 32 is required.
- The routine creates an indecipherable hexadecimal value. Therefore, it is necessary to format the value with one of the $N8601 formats. In this case, use the $N8601E*w.d* format.

Example 12: Converting an Interval to a Datetime Value and a Duration

You can specify an interval value as two datetime values that are separated by a forward slash (/). The slash separates the beginning and ending values for an event, as shown in this example.

```
data _null_;
   length Final2 $16;
   int=input('2012-03-15T14:32:00/2012-03-29T09:45:00', $n8601e40.);
   call is8601_convert('intvl', 'dt/du', int, Final1, Final2);
```

```
   put Final1= / Final2=$n8601e.;
run;
```

The resulting values for Final1 and Final2 are listed here:

- Final1=1647441120
- Final2=P13DT19H13M

In this example, the following statements apply:

- The datetime values are specified inside single quotation marks so that the INPUT function and the $N8601E*w*. informat can convert the values into an interval value in the variable *Int*.
- The CALL IS8601_CONVERT routine converts that interval value to two variables: Final1 is a datetime variable, and Final2 is a duration variable.
- Because a SAS datetime variable (Final1) is a numeric variable that can be stored in 8 bytes, a length value is not required for this variable in the LENGTH statement. However, a duration requires a length of 16. Therefore, a length is specified for Final2 in the LENGTH statement.
- Because Final2 is a duration, the $N8601E*w*. format is used to write its value to the log so that the output is understandable.

If the first datetime value in the example was missing the month value (2012- - -15T14:32:00), you could supply that value by specifying the value in the last arguments of the CALL routine, as shown here:

```
call is8601_convert('intvl', 'dt/du', int, Final1, Final2, , 2);
```

The month is specified here as 2, and the two consecutive commas preceding that value indicate that a year value was not supplied. Therefore, a datetime and a duration are computed based on the date 2012–02–15T14:32:00 and the other date that is specified in the code. In this case, the output from the PUT statement for each variable is as follows:

- Final1=1644935520
- Final2=P1M13DT19H13M

CALL LABEL Routine

Assigns a variable label to a specified character variable.

Category: Variable Control

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL LABEL(*variable-1*, *variable-2*);

Required Arguments

variable-1
specifies any SAS variable. If *variable-1* does not have a label, the variable name is assigned as the value of *variable-2*.

variable-2
specifies any SAS character variable. Variable labels can be up to 256 characters long. Therefore, the length of *variable-2* should be at least 256 characters to avoid truncating variable labels.

Note To conserve space, you should set the length of *variable-2* to the length of the label for *variable-1*, if it is known.

Details

The CALL LABEL routine assigns the label of the *variable-1* variable to the character variable *variable-2*.

Example

This example uses the CALL LABEL routine with array references to assign the labels of all variables in the data set Old as values of the variable LAB in data set New.

```
data new;
   set old;

      /* all character variables in old */
   array abc{*} _character_;
      /* all numeric variables in old */
   array def{*} _numeric_;
      /* lab is not in either array */
   length lab $256;
   do i=1 to dim(abc);
         /* get label of character variable */
      call label(abc{i},lab);
         /* write label to an observation */
      output;
   end;
   do j=1 to dim(def);
         /* get label of numeric variable */
      call label(def{j},lab);
         /* write label to an observation */
      output;
```

```
    end;
    stop;
    keep lab;
run;
```

See Also

Functions:

- "VLABEL Function" on page 1547

CALL LEXCOMB Routine

Generates all distinct combinations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

Interaction: When invoked by the %SYSCALL macro statement, CALL LEXCOMB removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 12.

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL LEXCOMB(*count*, *k*, *variable-1*, ..., *variable-n*);

Required Arguments

count
specifies an integer value that is assigned values from 1 to the number of combinations in a loop.

k
specifies an integer constant, variable, or expression between 1 and *n*, inclusive, that specifies the number of items in each combination.

variable
specifies either all numeric variables or all character variables that have the same length. The values of these variables are permuted.

Requirement Initialize these variables before you call the LEXCOMB routine.

Tip After calling LEXCOMB, the first *k* variables contain the values in one combination.

Details

The Basics

Use the CALL LEXCOMB routine in a loop where the first argument to CALL LEXCOMB takes each integral value from 1 to the number of distinct combinations of the nonmissing values of the variables. In each call to LEXCOMB within this loop, *k* should have the same value.

Number of Combinations

When all of the variables have nonmissing, unequal values, the number of combinations is COMB(*n*,*k*). If the number of variables that have missing values is *m* and all the nonmissing values are unequal, LEXCOMB produces COMB(*n*-*m*,*k*) combinations because the missing values are omitted from the combinations.

When some of the variables have equal values, the exact number of combinations is difficult to compute. If you cannot compute the exact number of combinations, use the LEXCOMB function instead of the CALL LEXCOMB routine.

CALL LEXCOMB Processing

On the first call to the LEXCOMB routine, the following actions occur:

- The argument types and lengths are checked for consistency.
- The *m* missing values are assigned to the last *m* arguments.
- The *n*-*m* nonmissing values are assigned in ascending order to the first *n*-*m* arguments following *count*.

On subsequent calls, up to and including the last combination, the next distinct combination of the nonmissing values is generated in lexicographic order.

If you call the LEXCOMB routine with the first argument out of sequence, the results are not useful. In particular, if you initialize the variables and then immediately call the LEXCOMB routine with a first argument of *j*, you will not get the *j*th combination (except when *j* is 1). To get the *j*th combination, you must call the LEXCOMB routine *j* times, and the first argument takes values from 1 through *j* in that exact order.

Using the CALL LEXCOMB Routine with Macros

You can call the LEXCOMB routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same length, but they are required to be the same type. If %SYSCALL identifies an argument as numeric, %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL LEXCOMB routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR is set to 0 and &SYSINFO is set to one of the following values:

- 1 if *count*=1 and at least one variable has a nonmissing value
- 1 if the value of *variable-1* changed
- *j* if *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j*=*i*+1
- –1 if all distinct combinations have already been generated

Comparisons

The CALL LEXCOMB routine generates all distinct combinations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order. The CALL ALLCOMB routine generates all combinations of the values of *n* variables taken *k* at a time in a minimal change order.

Examples

Example 1: Using CALL LEXCOMB in a DATA Step

This example calls the LEXCOMB routine to generate distinct combinations in lexicographic order.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   n=dim(x);
   k=3;
   ncomb=comb(n, k);
   do j=1 to ncomb;
      call lexcomb(j, k, of x[*]);
      put j 5. +3 x1-x3;
   end;
run;
```

SAS writes the following results to the log:

```
 1   ant bee cat
 2   ant bee dog
 3   ant bee ewe
 4   ant cat dog
 5   ant cat ewe
 6   ant dog ewe
 7   bee cat dog
 8   bee cat ewe
 9   bee dog ewe
10   cat dog ewe
```

Example 2: Using CALL LEXCOMB with Macros

Here is an example of the CALL LEXCOMB routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=baboon;
   %let x3=baboon;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=2;
   %let ncomb=%sysfunc(comb(5, &k));
      %do j=1 %to &ncomb;
         %syscall lexcomb(j, k, x1, x2, x3, x4, x5);
         %let jfmt=%qsysfunc(putn(&j, 5. ));
         %let pad=%qsysfunc(repeat(%str( ), 20-%length(&x1 &x2)));
         %put &jfmt: &x1 &x2 &pad sysinfo=&sysinfo;
         %if &sysinfo < 0 %then %let j=%eval(&ncomb+1);
      %end;
%mend;
%test
```

SAS writes the following results to the log:

```
1: ant baboon             sysinfo=1
2: ant hippopotamus       sysinfo=2
3: ant zebra              sysinfo=2
4: baboon baboon          sysinfo=1
5: baboon hippopotamus    sysinfo=2
6: baboon zebra           sysinfo=2
7: hippopotamus zebra     sysinfo=1
8: hippopotamus zebra     sysinfo=-1
```

See Also

Functions:

- “LEXCOMB Function” on page 1057

CALL Routines:

- “CALL ALLCOMB Routine” on page 215

CALL LEXCOMBI Routine

Generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order.

Syntax

CALL LEXCOMBI(*n*, *k*, *index-1*, ..., *index-k*);

Required Arguments

n
is a numeric constant, variable, or expression that specifies the total number of objects.

k
is a numeric constant, variable, or expression that specifies the number of objects in each combination.

index
is a numeric variable that contains indices of the objects in the combination that is returned. Indices are integers between 1 and *n*, inclusive.

Tip If *index-1* is missing or 0, the CALL LEXCOMBI routine initializes the indices to `index-1=1` through `index-k=k`. Otherwise, CALL LEXCOMBI creates a new combination by removing one index from the combination and adding another index.

Details

CALL LEXCOMBI Processing

Before the first call to the LEXCOMBI routine, complete one of the following tasks:

- Set *index-1* equal to 0 or to a missing value.
- Initialize *index-1* through *index-k* to distinct integers between 1 and *n*, inclusive.

The number of combinations of *n* objects taken *k* at a time can be computed as COMB(*n*,*k*). To generate all combinations of *n* objects taken *k* at a time, call LEXCOMBI in a loop that executes COMB(*n*,*k*) times.

Using the CALL LEXCOMBI Routine with Macros

If you call the LEXCOMBI routine from the macro processor with %SYSCALL, you must initialize all arguments to numeric values. %SYSCALL reformats the values that are returned.

If an error occurs during the execution of the CALL LEXCOMBI routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR is set to 0 and &SYSINFO is set to one of the following values:

- 1 if the value of *variable-1* changed
- *j* if *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j*=*i*+1
- –1 if all distinct combinations have already been generated

Comparisons

The CALL LEXCOMBI routine generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order. The CALL ALLCOMBI routine generates all combinations of the indices of *n* objects taken *k* at a time in a minimal change order.

Examples

Example 1: Using the CALL LEXCOMBI Routine with the DATA Step

The following example uses the CALL LEXCOMBI routine to generate combinations of indices in lexicographic order.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   array c[3] $3;
   array i[3];
   n=dim(x);
   k=dim(i);
   i[1]=0;
   ncomb=comb(n, k);
   do j=1 to ncomb;
      call lexcombi(n, k, of i[*]);
      do h=1 to k;
         c[h]=x[i[h]];
      end;
      put @4 j= @10 'i= ' i[*] +3 'c= ' c[*];
   end;
run;
```

SAS writes the following results to the log:

```
j=1   i= 1 2 3    c= ant bee cat
j=2   i= 1 2 4    c= ant bee dog
j=3   i= 1 2 5    c= ant bee ewe
j=4   i= 1 3 4    c= ant cat dog
j=5   i= 1 3 5    c= ant cat ewe
j=6   i= 1 4 5    c= ant dog ewe
j=7   i= 2 3 4    c= bee cat dog
j=8   i= 2 3 5    c= bee cat ewe
j=9   i= 2 4 5    c= bee dog ewe
j=10  i= 3 4 5    c= cat dog ewe
```

Example 2: Using the CALL LEXCOMBI Routine with Macros and Displaying the Return Code

Here is an example of the CALL LEXCOMBI routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=0;
   %let x2=0;
   %let x3=0;
   %let n=5;
   %let k=3;
   %let ncomb=%sysfunc(comb(&n, &k));
   %do j=1 %to &ncomb+1;
      %syscall lexcombi(n, k, x1, x2, x3);
      %let jfmt=%qsysfunc(putn(&j, 5.));
      %let pad=%qsysfunc(repeat(%str(), 6-%length(&x1 &x2 &x3)));
      %put &jfmt: &x1 &x2 &x3 &pad sysinfo=&sysinfo;
   %end;
%mend;
%test
```

SAS writes the following results to the log:

```
 1: 1 2 3  sysinfo=1
 2: 1 2 4  sysinfo=3
 3: 1 2 5  sysinfo=3
 4: 1 3 4  sysinfo=2
 5: 1 3 5  sysinfo=3
 6: 1 4 5  sysinfo=2
 7: 2 3 4  sysinfo=1
 8: 2 3 5  sysinfo=3
 9: 2 4 5  sysinfo=2
10: 3 4 5  sysinfo=1
11: 3 4 5  sysinfo=-1
```

See Also

CALL Routines:

- “CALL LEXCOMB Routine” on page 255
- “CALL ALLCOMBI Routine” on page 219

CALL LEXPERK Routine

Generates all distinct permutations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

Interaction: When invoked by the %SYSCALL macro statement, CALL LEXPERK removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 12.

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL LEXPERK(*count*, *k*, *variable-1*, ..., *variable-n*);

Required Arguments

count
specifies an integer variable that is assigned a value from 1 to the number of permutations in a loop.

k
specifies an integer constant, variable, or expression between 1 and *n*, inclusive, that specifies the number of items in each permutation.

variable
specifies either all numeric variables or all character variables that have the same length. The values of these variables are permuted.

Requirement: Initialize these variables before you call the LEXPERK routine.

Tip: After calling LEXPERK, the first *k* variables contain the values in one permutation.

Details

The Basics

Use the CALL LEXPERK routine in a loop where the first argument to CALL LEXPERK accepts each integral value from 1 to the number of distinct permutations of *k* nonmissing values of the variables. In each call to LEXPERK within this loop, *k* should have the same value.

Number of Permutations

When all of the variables have nonmissing, unequal values, the number of permutations is PERM(,*k*). If the number of variables that have missing values is *m*, and all the nonmissing values are unequal, CALL LEXPERK produces PERM(*n*-*m*,*k*) permutations because the missing values are omitted from the permutations. When some of the variables have equal values, the exact number of permutations is difficult to compute. If you cannot compute the exact number of permutations, use the LEXPERK function instead of the CALL LEXPERK routine.

CALL LEXPERK Processing

On the first call to the LEXPERK routine, the following actions occur:

- The argument types and lengths are checked for consistency.
- The *m* missing values are assigned to the last *m* arguments.
- The *n*-*m* nonmissing values are assigned in ascending order to the first *n*-*m* arguments following *count*.

On subsequent calls, up to and including the last permutation, the next distinct permutation of *k* nonmissing values is generated in lexicographic order.

If you call the LEXPERK routine with the first argument out of sequence, the results are not useful. In particular, if you initialize the variables and then immediately call the LEXPERK routine with a first argument of *j*, you do not get the *j*th permutation (except when *j* is 1). To get the *j*th permutation, you must call LEXPERK *j* times. The first argument takes values from 1 through *j* in that exact order.

Using the CALL LEXPERK Routine with Macros

You can call the LEXPERK routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same length, but they are required to be the same type. If %SYSCALL identifies an argument as numeric, %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL LEXPERK routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR is set to 0 and &SYSINFO is set to one of the following values:

- 1 if *count*=1 and at least one variable has a nonmissing value
- 1 if *count*>1 and the value of *variable-1* changed
- *j* if *count*>1 and *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j*=*i*+1
- –1 if all distinct permutations were already generated

Comparisons

The CALL LEXPERK routine generates all distinct permutations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order. The CALL ALLPERM routine generates all permutations of the values of several variables in a minimal change order.

Examples

Example 1: Using CALL LEXPERK in a DATA Step

Here is an example of the CALL LEXPERK routine.

```
data _null_;
   array x[5] $3 ('V' 'W' 'X' 'Y' 'Z');
   n=dim(x);
   k=3;
   nperm=perm(n, k);
   do j=1 to nperm;
      call lexperk(j, k, of x[*]);
      put j 5. +3 x1-x3;
   end;
run;
```

SAS writes the following results to the log:

```
 1   V W X
 2   V W Y
 3   V W Z
 4   V X W
 5   V X Y
 6   V X Z
 7   V Y W
 8   V Y X
 9   V Y Z
10   V Z W
11   V Z X
12   V Z Y
13   W V X
14   W V Y
15   W V Z
16   W X V
17   W X Y
18   W X Z
19   W Y V
20   W Y X
21   W Y Z
22   W Z V
23   W Z X
24   W Z Y
25   X V W
26   X V Y
27   X V Z
28   X W V
29   X W Y
30   X W Z
31   X Y V
32   X Y W
33   X Y Z
34   X Z V
35   X Z W
36   X Z Y
37   Y V W
38   Y V X
39   Y V Z
40   Y W V
41   Y W X
42   Y W Z
43   Y X V
44   Y X W
45   Y X Z
46   Y Z V
47   Y Z W
48   Y Z X
49   Z V W
50   Z V X
51   Z V Y
52   Z W V
53   Z W X
54   Z W Y
55   Z X V
56   Z X W
57   Z X Y
58   Z Y V
59   Z Y W
60   Z Y X
```

Example 2: Using CALL LEXPERK with Macros

Here is an example of the CALL LEXPERK routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=baboon;
   %let x3=baboon;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=2;
   %let nperk=%sysfunc(perm(5, &k));
   %do j=1 %to &nperk;
      %syscall lexperk(j, k, x1, x2, x3, x4, x5);
      %let jfmt=%qsysfunc(putn(&j, 5.));
      %let pad=%qsysfunc(repeat(%str(), 20-%length(&x1 &x2)));
      %put &jfmt: &x1 &x2 &pad sysinfo=&sysinfo;
      %if &sysinfo<0 %then %let j=%eval(&nperk+1);
   %end;
%mend;
%test
```

SAS writes the following results to the log:

```
 1: ant baboon  sysinfo=1
 2: ant hippopotamus  sysinfo=2
 3: ant zebra  sysinfo=2
 4: baboon ant  sysinfo=1
 5: baboon baboon  sysinfo=2
 6: baboon hippopotamus  sysinfo=2
 7: baboon zebra  sysinfo=2
 8: hippopotamus ant  sysinfo=1
 9: hippopotamus baboon  sysinfo=2
10: hippopotamus zebra  sysinfo=2
11: zebra ant  sysinfo=1
12: zebra baboon  sysinfo=2
13: zebra hippopotamus  sysinfo=2
14: zebra hippopotamus  sysinfo=-1
```

See Also

Functions:

- “LEXPERM Function” on page 1065

CALL Routines:

- “CALL ALLPERM Routine” on page 222
- “CALL RANPERK Routine” on page 312
- “CALL RANPERM Routine” on page 314

CALL LEXPERM Routine

Generates all distinct permutations of the nonmissing values of several variables in lexicographic order.

Category: Combinatorial

Interaction: When invoked by the %SYSCALL macro statement, CALL LEXPERM removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 12.

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL LEXPERM(*count*, *variable-1* <, ..., *variable-N*>);

Required Arguments

count
specifies a numeric variable that has an integer value that ranges from 1 to the number of permutations.

variable
specifies either all numeric variables or all character variables that have the same length. The values of these variables are permuted by LEXPERM.

Requirement Initialize these variables before you call the LEXPERM routine.

Details

Determine the Number of Distinct Permutations

These variables are defined for use in the equation that follows:

N
specifies the number of variables that are being permuted (that is, the number of arguments minus one).

M
specifies the number of missing values among the variables that are being permuted.

d
specifies the number of distinct nonmissing values among the arguments.

N_i
for i=1 through i=d, N_i specifies the number of instances of the i*th* distinct value.

The number of distinct permutations of nonmissing values of the arguments is expressed as follows:

$$P = \frac{(N_1 + N_2 + ... + N_d)!}{N_1!N_2!...N_d!} < = N!$$

CALL LEXPERM Processing

Use the CALL LEXPERM routine in a loop where the argument *count* accepts each integral value from 1 to P. You do not need to compute P provided you exit the loop when CALL LEXPERM returns a value that is less than 0.

For 1=*count*<P, the following actions occur:

- The argument types and lengths are checked for consistency.
- The M missing values are assigned to the last M arguments.
- The N-M nonmissing values are assigned in ascending order to the first N-M arguments following *count*.
- CALL LEXPERM returns 1.

For 1<*count*<=P, the following actions occur:

- The next distinct permutation of the nonmissing values is generated in lexicographic order.
- If *variable-1* through *variable-I* did not change, but *variable-J* did change, where J=I+1, then CALL LEXPERM returns J.

For *count*>P, CALL LEXPERM returns –1.

If the CALL LEXPERM routine is executed with the first argument out of sequence, the results might not be useful. In particular, if you initialize the variables and then immediately execute CALL LEXPERM with a first argument of K, you do not get the K*th* permutation (except when K is 1). To get the K*th* permutation, you must execute CALL LEXPERM K times. The first argument accepts values from 1 through K in that exact order.

Using the CALL LEXPERM Routine with Macros

You can call the LEXPERM routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same length, but they must be the same type. If %SYSCALL identifies an argument as numeric, %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL LEXPERM routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR is set to 0 and &SYSINFO is set to one of the following values:

- 1 if 1=*count*<P
- 1 if 1<*count*<=P and the value of *variable-1* changed
- J if 1<*count*<=P and *variable-1* through *variable-I* did not change, but *variable-J* did change, where J=I+1
- –1 if *count*>P

Comparisons

SAS provides three functions or CALL routines for generating all permutations:

- ALLPERM generates all *possible* permutations of the values, *missing* or *nonmissing*, of several variables. Each permutation is formed from the previous permutation by interchanging two consecutive values.
- LEXPERM generates all *distinct* permutations of the *nonmissing* values of several variables. The permutations are generated in lexicographic order.
- LEXPERK generates all *distinct* permutations of K of the *nonmissing* values of N variables. The permutations are generated in lexicographic order.

ALLPERM is the fastest of these functions and CALL routines. LEXPERK is the slowest.

Examples

Example 1: Using CALL LEXPERM in a DATA Step

The following example uses the DATA step to generate all distinct permutations of the nonmissing values of several variables in lexicographic order.

```
data _null_;
   array x[4] $3 ('ant' 'bee' 'cat' 'dog');
   n=dim(x);
   nfact=fact(n);
   do i=1 to nfact;
      call lexperm(i, of x[*]);
      put i 5. +2 x[*];
   end;
run;
```

SAS writes the following results to the log:

```
 1  ant bee cat dog
 2  ant bee dog cat
 3  ant cat bee dog
 4  ant cat dog bee
 5  ant dog bee cat
 6  ant dog cat bee
 7  bee ant cat dog
 8  bee ant dog cat
 9  bee cat ant dog
10  bee cat dog ant
11  bee dog ant cat
12  bee dog cat ant
13  cat ant bee dog
14  cat ant dog bee
15  cat bee ant dog
16  cat bee dog ant
17  cat dog ant bee
18  cat dog bee ant
19  dog ant bee cat
20  dog ant cat bee
21  dog bee ant cat
22  dog bee cat ant
23  dog cat ant bee
24  dog cat bee ant
```

Example 2: Using CALL LEXPERM with Macros

Here is an example of the CALL LEXPERM routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=baboon;
   %let x3=baboon;
   %let x4=hippopotamus;
   %let n=4;
   %let nperm=%sysfunc(perm(4));
   %do j=1 %to &nperm;
      %syscall lexperm(j, x1, x2, x3, x4);
      %let jfmt=%qsysfunc(putn(&j, 5.));
      %put &jfmt: &x1 &x2 &x3 &x4 sysinfo=&sysinfo;
      %if &sysinfo<0 %then %let j=%eval(&nperm+1);
   %end;
%mend;

%test;
```

SAS writes the following results to the log:

```
 1: ant baboon baboon hippopotamus sysinfo=1
 2: ant baboon hippopotamus baboon sysinfo=3
 3: ant hippopotamus baboon baboon sysinfo=2
 4: baboon ant baboon hippopotamus sysinfo=1
 5: baboon ant hippopotamus baboon sysinfo=3
 6: baboon baboon ant hippopotamus sysinfo=2
 7: baboon baboon hippopotamus ant sysinfo=3
 8: baboon hippopotamus ant baboon sysinfo=2
 9: baboon hippopotamus baboon ant sysinfo=3
10: hippopotamus ant baboon baboon sysinfo=1
11: hippopotamus baboon ant baboon sysinfo=2
12: hippopotamus baboon baboon ant sysinfo=3
13: hippopotamus baboon baboon ant sysinfo=-1
```

See Also

Functions:

- “LEXPERK Function” on page 1063
- “LEXPERM Function” on page 1065

CALL Routines:

- “CALL ALLPERM Routine” on page 222
- “CALL RANPERK Routine” on page 312
- “CALL RANPERM Routine” on page 314

CALL LOGISTIC Routine

Applies the logistic function to each argument.

Category: Mathematical

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL LOGISTIC(*argument*<, *argument–1*, ... *argument-n*>);

Required Argument

argument
is a numeric variable.

Restriction The CALL LOGISTIC routine accepts variables as the only valid arguments. Do not use a constant or a SAS expression because the CALL routine is unable to update these arguments.

Details

The CALL LOGISTIC routine replaces each argument with the logistic value of that argument. For example, an arbitrary argument x_j is replaced by the following equation:

$$\frac{e^{x}j}{1+e^{x}j}$$

If any argument contains a missing value, CALL LOGISTIC returns missing values for all the arguments.

Example

```
data one;
   x=0.5;
   y=-0.5;
   call logistic(x, y);
   put x= y=;
run;
```

The preceding statements produce these results:

```
x=0.6224593312 y=0.3775406688
```

CALL MISSING Routine

Assigns missing values to the specified character or numeric variables.

Categories: Character
Missing Values

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL MISSING(*variable-name-1* <, *variable-name-2*, ... *variable-name-n*>);

Required Argument

variable-name
specifies the name of SAS character or numeric variables.

Details

The CALL MISSING routine assigns an ordinary numeric missing value (.) to each numeric variable in the argument list.

The CALL MISSING routine assigns a character missing value (a blank) to each character variable in the argument list. If the current length of the character variable equals the maximum length, the current length is not changed. Otherwise, the current length is set to 1.

You can mix character and numeric variables in the argument list.

Comparisons

The MISSING function checks whether the argument has a missing value but does not change the value of the argument.

Example

```
data one;
   prod='shoes';
   invty=7498;
   sales=23759;
   call missing(sales);
   put prod= invty= sales=;
run;
```

The preceding statements produce this result:

```
prod=shoes invty=7498 sales=.
```

```
data one;
   prod='shoes';
   invty=7498;
   sales=23759;
   call missing(prod, invty);
   put prod= invty= sales=;
run;
```

The preceding statements produce this result:

```
prod= invty=. sales=23759
```

```
data one;
   prod='shoes';
   invty=7498;
   sales=23759;
   call missing(of _all_);
   put prod= invty= sales=;
run;
```

The preceding statements produce this result:

```
prod= invty=. sales=.
```

See Also

Functions:

- "MISSING Function" on page 1108

CALL MODULE Routine

Calls an external routine without any return code.

Category: External Routines

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

CAUTION: **Starting in 2025.04, the CALL MODULE call routine is removed from SAS as a security precaution.** The call routine is no longer supported and should not be used. This call routine should be removed from DATA step programs to prevent an error.

Syntax

CALL MODULE(<*control*>, *module*, *argument-1*, *argument-2* ..., *argument-n*);

number=**MODULEN**(<*control*>, *module*, *argument-1*, *argument-2* ..., *argument-n*);

character=**MODULEC**(<*control*>, *module*, *argument-1* ..., *argument-2*, *argument-n*);

CALL MODULEI(<*control*>, *module*, *argument-1*, *argument-2* ..., *argument-n*);

number=**MODULEIN**(<*control*>, *module*, *argument-1*, *argument-2* ..., *argument-n*);

character=**MODULEIC**(<*control*>, *module*, *argument-1*, *argument-2* ..., *argument-n*);

Required Arguments

module

specifies the name of the external module to use. The *module* can be specified as a shared library with the routine name or ordinal value, separated by a comma. You do not need to specify the shared library name if you specified the MODULE attribute for the routine in the SASCBTBL attribute table, as long as the routine name is unique (that is, no other routine has the same name in the attribute file).

The module must reside in a shared library, and it must be able to be called externally. Although the shared library name is not case sensitive, the routine name is based on the restraints of the routine's implementation language, so the routine name is case sensitive.

If the shared library supports ordinal-value naming, you can provide the shared library name followed by a decimal number, such as 'XYZ,30'.

You can specify *module* as a SAS character expression instead of as a constant. Most often, it is passed as a constant.

argument-1, argument-2, ..., argument-n

specifies the arguments to pass to the requested routine. Use the proper attributes for the arguments (that is, numeric arguments for numeric attributes and character arguments for character attributes).

CAUTION

Be sure to use the correct arguments and attributes. If you use incorrect arguments or attributes for a shared library function, you can cause SAS to crash or you might see unexpected results.

Optional Argument

control

is an optional control string whose first character must be an asterisk (*), followed by any combination of the following characters:

I prints the hexadecimal representations of all arguments to the MODULE function and to the requested shared library routine before and after the shared library routine is called. You can use this option to help diagnose problems that are caused by incorrect arguments or attribute tables. If you specify the I option, the E option is implied.

E prints detailed error messages. Without the E option (or the I option, which supersedes it), the only error message that the MODULE function generates is "Invalid argument to function," which is usually not enough information to determine the cause of the error.

S*x* uses *x* as a separator character to separate field definitions. You can then specify *x* in the argument list as its own character argument to serve as a delimiter for a list of arguments that you want to group together as a single structure. Use this option only if you do not supply

an entry in the SASCBTBL attribute table. If you do supply an entry for this module in the SASCBTBL attribute table, you should use the FDSTART option in the ARG statement in the table to separate structure definitions.

H provides brief help information about the syntax of the MODULE routines, the attribute file format, and the suggested SAS formats and informats.

For example, the control string '*IS/' specifies that parameter lists be printed and that the string '/' is to be treated as a separator character in the argument list.

Details

Using IML Functions with the CALL MODULE Routine

The following functions permit vector and matrix arguments. You can use them only within the IML procedure:

- CALL MODULEI
- MODULEIN
- MODULEIC

For more information, see the *SAS/IML Studio: User's Guide*.

Working with Attribute Tables

The CALL MODULE routine executes a routine that resides in an external library. The routine uses the required *module-name* argument to identify the name of the library, as well as optional arguments.

CALL MODULE builds a parameter list by using the information in the arguments and a routine description and argument attribute table that you define in a separate file. The attribute table is a sequential text file that contains descriptions of the routines that you can invoke with the CALL MODULE routine. The purpose of the table is to define how CALL MODULE interprets its supplied arguments when it builds a parameter list to pass to the external routine. The attribute table contains a description for each external routine that you intend to call, and descriptions of each argument associated with that routine.

Before you invoke CALL MODULE, you must define the fileref of SASCBTBL to point to the external file that contains the attribute table. You can name the file whatever you want when you create it. In this way, you use SAS variables and formats as arguments to CALL MODULE and ensure that these arguments are properly converted before being passed to the external routine. If you do not define this fileref, CALL MODULE calls the requested routine without altering the arguments.

CAUTION

Using the CALL MODULE routine without a defined attribute table can cause SAS to fail or force you to reset your computer. You must use an attribute table for all external functions that you want to invoke.

Using Special Formats with the CALL MODULE Routine

Numeric Alignment

By default, the CALL MODULE routine forces numeric alignment for fields that are defined with the IB*w*., PIB*w*., and RB*w*. formats within the SASCBTBL table. Therefore, the fields are padded to provide an appropriate 4-byte or 8-byte boundary. Most current compilers expect this alignment.

New formats were created specifically for the CALL MODULE routine to accommodate routines that were built with a compiler that does not expect alignment. These new formats are IBUNALN*w*., PIBUNALN*w*., and RBUNALN*w*. Use these formats instead of the IB*w*., PIB*w*., and RB*w*. formats.

The IBUNALN*w*., PIBUNALN*w*. and RBUNALN*w*. Formats

The IBUNALN*w*., PIBUNALN*w*., and RBUNALN*w*. formats are used only with the CALL MODULE routine. They were created because the default behavior within an FDSTART structure forces alignment when you use the IB*w*. and PIB*w*. formats. The IBUNALN*w*., PIBUNALN*w*., and RBUNALN*w*. formats do not align data.

This example shows a SASCBTBL definition, which uses the IB*w*. format with the FDSTART structure:

```
filename sascbtbl temp;
data _null_;
   file sascbtbl;
   input;
   put _infile_;
   datalines4;
routine xyz minarg=2 maxarg=2;
arg 1 char format=$3. fdstart;
arg 2 num format=ib4.;
;;;;
```

This example shows a DATA step that uses the CALL MODULE routine based on the preceding SASCBTBL definition:

```
data _null_;
   x='abc';
   y=1;
   call module('xyz', x, y);
run;
```

When XYZ is called, it receives a single argument, which is the address of an aggregate structure. The aggregate structure contains the three-bytes `'abc'` followed by an alignment byte, which is then followed by the value 1 from the IB4. format ('00000001'x in a big endian environment and '01000000'x in a little endian environment). If the XYZ routine is not expecting alignment, it misreads the numeric value.

This example shows an SASCBTBL definition, which uses the IBUNALN*w*. format with the FDSTART structure:

```
filename sascbtbl temp;
data _null_;
   file sascbtbl;
   input;
   put _infile_;
   datalines4;
routine xyz minarg=2 maxarg=2;
arg 1 char format=$3. fdstart;
arg 2 num format=ibunaln4.;
;;;;
```

XYZ is given the address of an aggregate structure that contains `'abc'` followed immediately by the value of 1, with no intervening alignment bytes.

Alignment occurs when numeric formats IB*w*., PIB*w*., or RB*w*. are used. If the width is 2, the alignment occurs on an even byte boundary. If the width is 4, the alignment occurs on a 4-byte boundary. If the width is 8, the alignment occurs on an 8-byte boundary.

Comparisons

- The MODULEN and MODULEC functions return a number and a character, respectively. The CALL MODULE routine does not return a value.
- The CALL MODULEI routine and the MODULEIC and MODULEIN functions allow vector and matrix arguments. Their return values are scalar. You can invoke CALL MODULEI, MODULEIC, and MODULEIN routines only from the IML procedure.

Example: Using the CALL MODULE Routine

This example calls the XYZ routine. Use the following attribute table:

```
routine xyz minarg=2 maxarg=2;
arg 1 input num byvalue format=ib4.;
arg 2 output char format=$char10.;
```

Here is the sample SAS code that calls the XYZ function:

```
data _null_;
   call module('xyz', 1, x);
run;
```

See Also

Functions:

- "MODULEC Function" on page 1116
- "MODULEN Function" on page 1117

CALL POKELONG Routine

Writes a value directly into memory on a 64-bit platform.

Category: Special

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL POKELONG(*source*, *pointer* <, *length*> <, *floating-point*>);

Required Arguments

source
specifies a character constant, variable, or expression that contains a value to write into memory.

pointer
specifies a character string that contains the virtual address of the data that the CALL POKELONG routine alters.

Optional Arguments

length
specifies a numeric SAS expression that contains the number of bytes to write from the *source* to the address that is indicated by the *pointer*. If you omit *length*, the CALL POKELONG routine copies the entire value of *source* to the specified memory location.

floating-point
specifies that the value of *source* is stored as a floating-point number. The value of *floating-point* can be any number.

Tip If you do not use the *floating-point* argument, then *source* is stored as an integer value.

Details

The CALL POKELONG routine is intended only for experienced programmers in specific cases.

CAUTION

Typing directly into memory can cause devastating problems. If you use this routine, use extreme caution in your programming and in your typing. This routine bypasses the normal safeguards that prevent you from destroying a vital element in your SAS session or in another piece of software that is active at the time.

If you do not have access to the memory location that you specify, the CALL POKELONG routine returns an `Invalid argument` error.

If you use the fourth argument, a floating-point number is assumed to be the value that is stored. If you do not use the fourth argument, an integer value is assumed to be stored.

CALL PRXCHANGE Routine

Performs a pattern-matching replacement.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Interaction: When invoked by the %SYSCALL macro statement, CALL PRXCHANGE removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 12.

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

CALL PRXCHANGE(*regular-expression-id*, *times*, *old-string* <, *new-string* <, *result-length*
<, *truncation-value* <, *number-of-changes*> > > >);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned from the PRXPARSE function.

times
is a numeric constant, variable, or expression that specifies the number of times to find a match and replace a matching pattern.

Tip If the value of *times* is –1, all matching patterns are replaced.

old-string
specifies a character expression that you want to find, and then replace with the results of *new-string*.

Tip All changes are made to *old-string* if you do not use the *new-string* argument.

Optional Arguments

new-string
specifies a character variable in which to place the results of the change to *old-string*.

Tip If you use the *new-string* argument in the call to the PRXCHANGE routine, *old-string* is not modified.

result-length
is a numeric variable with a return value that is the number of characters that are copied into the result.

Tip Trailing blanks in the value of *old-string* are not copied to *new-string*, and are therefore not included as part of the length in *result-length*.

truncation-value
is a numeric variable with a returned value that is either 0 or 1, depending on the result of the change operation:

0 if the entire replacement result is not longer than the length of *new-string*.

1 if the entire replacement result is longer than the length of *new-string*.

number-of-changes
is a numeric variable with a returned value that is the total number of replacements that were made. If the result is truncated when it is placed into *new-string*, the value of *number-of-changes* is not guaranteed.

Details

The CALL PRXCHANGE routine matches and replaces a pattern. If the value of *times* is –1, the replacement is performed as many times as possible.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 53.

Comparisons

The CALL PRXCHANGE routine is similar to the PRXCHANGE function. The difference is that the CALL routine returns the value of the pattern-matching replacement as one of its parameters instead of as a return argument.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. For a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Example

The following example replaces all occurrences of cat, rat, or bat with the value TREE.

```
data _null_;
      /* Use a pattern to replace all occurrences of cat,      */
      /* rat, or bat with the value TREE.                      */
   length text $ 46;
   RegularExpressionId = prxparse('s/[crb]at/tree/');
   text = 'The woods have a bat, cat, bat, and a rat!';
      /* Use CALL PRXCHANGE to perform the search and replace. */
      /* Because the argument times has a value of -1, the     */
      /* replacement is performed as many times as possible.   */
   call prxchange(RegularExpressionId, -1, text);
   put text;
run;
```

SAS writes the following results to the log:

```
The woods have a tree, tree, tree, and a tree!
```

See Also

Functions:

- "PRXCHANGE Function" on page 1261
- "PRXPAREN Function" on page 1272
- "PRXPARSE Function" on page 1274
- "PRXMATCH Function" on page 1267
- "PRXPOSN Function" on page 1276

CALL Routines:

- "CALL PRXDEBUG Routine" on page 283
- "CALL PRXFREE Routine" on page 285
- "CALL PRXNEXT Routine" on page 287
- "CALL PRXPOSN Routine" on page 289
- "CALL PRXSUBSTR Routine" on page 292

CALL PRXDEBUG Routine

Enables Perl regular expressions in a DATA step to send debugging output to the SAS log.

Category: Character String Matching

Restriction: Use with the CALL PRXCHANGE, CALL PRXFREE, CALL PRXNEXT, CALL PRXPOSN, CALL PRXSUBSTR, PRXPARSE, PRXPAREN, and PRXMATCH functions and CALL routines.

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

CALL PRXDEBUG(*on-off*);

Required Argument

on-off
specifies a numeric constant, variable, or expression. If the value of *on-off* is positive and nonzero, debugging is turned on. If the value of *on-off* is zero, debugging is turned off.

Details

The CALL PRXDEBUG routine provides information about how a Perl regular expression is compiled and about which steps are taken when a pattern is matched to a character value.

You can turn debugging on and off multiple times in your program if you want to see debugging output for particular Perl regular expression function calls.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 53.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. For a list and short description of these

functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Example

This example produces debugging output.

```
data _null_;
      /* Turn the debugging option on. */
   call prxdebug(1);
   putlog 'PRXPARSE: ';
   re = prxparse('/[bc]d(ef*g)+h[ij]k$/');
   putlog 'PRXMATCH: ';
   pos = prxmatch(re, 'abcdefg_gh_');
      /* Turn the debugging option off. */
   call prxdebug(0);
run;
```

SAS writes the following results to the log:

```
PRXPARSE:
Compiling REx '[bc]d(ef*g)+h[ij]k$'  1
Final program:
   1: ANYOFM[bc] (3)                          2
   3: EXACT <d> (5)
   5: CURLYX<1:1>[0]{1,INFTY} (21)

9:   OPEN1 (11)
  11:     EXACT <e> (13)
  13:     STAR (16)
  14:       EXACT <f> (0)
  16:     EXACT <g> (18)
  18:   CLOSE1
(20)
  20: WHILEM[1/1] (0)
  21: NOTHING (22)
  22: EXACT <h> (24)
  24: ANYOFR[ij] (26)
  26: EXACT <k> (28)
  28: SEOL (29)
  29:
END (0)
anchored utf8 "de" at 1..1 floating utf8 "gh" at 3..9223372036854775807 (checking
floating) stclass
ANYOFM[bc] minlen 7                          3

PRXMATCH:
Matching REx "[bc]d(ef*g)+h[ij]k$" against "abcdefg_gh_"
UTF-8 pattern and string...
Intuit: trying to determine minimum start position...
  doing 'check' fbm scan, [3..9] gave -1
  Did not find floating substr "gh"...
Match rejected by optimizer
```

The following items correspond to the lines that are numbered in the preceding SAS log.

1 This line shows the precompiled form of the Perl regular expression.

2 This line begins a list of program nodes in compiled form for regular expressions.

3 These two lines provide optimizer information. In the example, the optimizer found that the match should contain the substring `de` at offset 1 and the substring `gh` at an offset between 3 and infinity. To rule out a pattern match quickly, Perl checks substring `gh` before it checks substring `de`.

The optimizer might use the information that the match begins at the *first* ID (line 2), with a character class (line 5), and cannot be shorter than seven characters (line 6).

See Also

Functions:

- "PRXCHANGE Function" on page 1261
- "PRXPAREN Function" on page 1272
- "PRXMATCH Function" on page 1267
- "PRXPARSE Function" on page 1274
- "PRXPOSN Function" on page 1276

CALL Routines:

- "CALL PRXCHANGE Routine" on page 280
- "CALL PRXFREE Routine" on page 285
- "CALL PRXNEXT Routine" on page 287
- "CALL PRXPOSN Routine" on page 289
- "CALL PRXSUBSTR Routine" on page 292

CALL PRXFREE Routine

Frees memory that was allocated for a Perl regular expression.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

CALL PRXFREE(*regular-expression-id*);

Required Argument

regular-expression-id
specifies a numeric variable with a value that is the identification number that is returned by the PRXPARSE function. *regular-expression-id* is set to missing if the call to the PRXFREE routine occurs without error.

Details

The CALL PRXFREE routine frees unneeded resources that were allocated for a Perl regular expression.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 53.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

See Also

Functions:

- "PRXCHANGE Function" on page 1261
- "PRXPAREN Function" on page 1272
- "PRXPARSE Function" on page 1274
- "PRXPOSN Function" on page 1276

CALL Routines:

- "CALL PRXCHANGE Routine" on page 280
- "CALL PRXDEBUG Routine" on page 283
- "CALL PRXNEXT Routine" on page 287
- "CALL PRXPOSN Routine" on page 289
- "CALL PRXSUBSTR Routine" on page 292

CALL PRXNEXT Routine

Returns the position and length of a substring that matches a pattern and iterates over multiple matches within one string.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Interaction: When invoked by the %SYSCALL macro statement, CALL PRXNEXT removes the quotation marks from arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 12.

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

CALL PRXNEXT(*regular-expression-id*, *start*, *stop*, *source*, *position*, *length*);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is the identification number that is returned by the PRXPARSE function.

start
is a numeric variable that specifies the position at which to start the pattern matching in *source*. If the match is successful, CALL PRXNEXT returns a value of *position* + MAX(1, *length*). If the match is not successful, the value of *start* is not changed.

stop
is a numeric constant, variable, or expression that specifies the last character to use in *source*. If *stop* is –1, the last character is the last non-blank character in *source*.

source
specifies a character constant, variable, or expression that you want to search.

position
is a numeric variable with a returned value that is the position in *source* at which the pattern begins. If no match is found, CALL PRXNEXT returns 0.

length
is a numeric variable with a returned value that is the length of the string that is matched by the pattern. If no match is found, CALL PRXNEXT returns 0.

Details

The CALL PRXNEXT routine searches the variable *source* with a pattern. The routine returns the position and length of a pattern match that is located between the *start* and *stop* positions in *source*. Because the value of the *start* parameter is updated to be the position of the next character that follows a match, CALL PRXNEXT enables you to search a string for a pattern multiple times in succession.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 53.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. For a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Example

This example finds all instances of cat, rat, or bat in a text string.

```
data _null_;
   ExpressionID = prxparse('/[crb]at/');
   text = 'The woods have a bat, cat, and a rat!';
   start = 1;
   stop = length(text);
      /* Use PRXNEXT to find the first instance of the pattern, */
      /* then use DO WHILE to find all further instances.       */
      /* PRXNEXT changes the start parameter so that searching  */
      /* begins again after the last match.                     */
   call prxnext(ExpressionID, start, stop, text, position, length);
      do while (position > 0);
         found = substr(text, position, length);
         put found= position= length=;
         call prxnext(ExpressionID, start, stop, text, position,
length);
      end;
run;
```

SAS writes the following results to the log:

```
found=bat position=18 length=3
   found=cat position=23 length=3
   found=rat position=34 length=3
```

See Also

Functions:

- “PRXCHANGE Function” on page 1261
- “PRXMATCH Function” on page 1267
- “PRXPAREN Function” on page 1272
- “PRXPARSE Function” on page 1274
- “PRXPOSN Function” on page 1276

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXPOSN Routine” on page 289
- “CALL PRXSUBSTR Routine” on page 292

CALL PRXPOSN Routine

Returns the start position and length for a capture buffer.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

CALL PRXPOSN(*regular-expression-id*, *capture-buffer*, *start* <, *length*>);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned by the PRXPARSE function.

capture-buffer
is a numeric constant, variable, or expression with a value that identifies the capture buffer from which to retrieve the start position and length.

- If the value of *capture-buffer* is 0, CALL PRXPOSN returns the start position and length of the entire match.
- If the value of *capture-buffer* is between 1 and the number of open parentheses, CALL PRXPOSN returns the start position and length for that capture buffer.
- If the value of *capture-buffer* is greater than the number of open parentheses, CALL PRXPOSN returns missing values for the start position and length.

start
is a numeric variable with a returned value that is the position at which the capture buffer is found.

- If the value of *capture-buffer* is not found, CALL PRXPOSN returns 0 for the start position.
- If the value of *capture-buffer* is greater than the number of open parentheses in the pattern, CALL PRXPOSN returns a missing value for the start position.

Optional Argument

length
is a numeric variable with a returned value that is the pattern length of the previous pattern match.

- If the pattern match is not found, CALL PRXPOSN returns 0 for the length.
- If the value of *capture-buffer* is greater than the number of open parentheses in the pattern, CALL PRXPOSN returns a missing value for the length.

Details

The CALL PRXPOSN routine uses the results of PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT to return a capture buffer. A match must be found by one of these functions for the CALL PRXPOSN routine to return meaningful information.

A capture buffer is part of a match, enclosed in parentheses, that is specified in a regular expression. CALL PRXPOSN does not return the text for the capture buffer directly. It requires a call to the SUBSTR function to return the text.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 53.

Comparisons

The CALL PRXPOSN routine is similar to the PRXPOSN function, except that CALL PRXPOSN returns the position and length of the capture buffer rather than the capture buffer itself.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. For a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Examples

Example 1: Finding Submatches within a Match

The following example searches a regular expression and calls the PRXPOSN routine to find the position and length of three submatches.

```
data _null_;
   patternID = prxparse('/(\d\d):(\d\d)(am|pm)/');
   text = 'The time is 09:56am.';
   if prxmatch(patternID, text) then do;
      call prxposn(patternID, 1, position, length);
      hour = substr(text, position, length);
      call prxposn(patternID, 2, position, length);
      minute = substr(text, position, length);
      call prxposn(patternID, 3, position, length);
      ampm = substr(text, position, length);
      put hour= minute= ampm=;
      put text=;
   end;
run;
```

SAS writes the following results to the log:

```
hour=09 minute=56 ampm=am
text=The time is 09:56am.
```

Example 2: Parsing Time Data

This example parses time data and writes the results to the SAS log.

```
data _null_;
   if _N_ = 1 then
   do;
      retain patternID;
      pattern = "/(\d+):(\d\d)(?:\.(\d+))?/";
      patternID = prxparse(pattern);
   end;

   array match[3] $ 8;
   input minsec $80.;
   position = prxmatch(patternID, minsec);
```

```
      if position ^= 0 then
      do;
         do i = 1 to prxparen(patternID);
            call prxposn(patternID, i, start, length);
            if start ^= 0 then
               match[i] = substr(minsec, start, length);
         end;
         put match[1] "minutes, " match[2] "seconds" @;
         if ^missing(match[3]) then
            put ", " match[3] "milliseconds";
      end;
      datalines;
   14:56.456
   45:32
   ;
```

SAS writes the following results to the log:

```
14 minutes, 56 seconds, 456 milliseconds
   45 minutes, 32 seconds
```

See Also

Functions:

- “PRXCHANGE Function” on page 1261
- “PRXMATCH Function” on page 1267
- “PRXPAREN Function” on page 1272
- “PRXPARSE Function” on page 1274
- “PRXPOSN Function” on page 1276

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXSUBSTR Routine” on page 292

CALL PRXSUBSTR Routine

Returns the position and length of a substring that matches a pattern.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Interaction: When invoked by the %SYSCALL macro statement, CALL PRXSUBSTR removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 12.

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

CALL PRXSUBSTR(*regular-expression-id*, *source*, *position* <, *length*>);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is an identification number that is returned by the PRXPARSE function.

source
specifies a character constant, variable, or expression that you want to search.

position
is a numeric variable with a returned value that is the position in *source* where the pattern begins. If no match is found, CALL PRXSUBSTR returns 0.

Optional Argument

length
is a numeric variable with a returned value that is the length of the substring that is matched by the pattern. If no match is found, CALL PRXSUBSTR returns 0.

Details

The CALL PRXSUBSTR routine searches the variable *source* with the pattern from PRXPARSE, returns the position of the start of the string, and if specified, returns the length of the string that is matched. By default, when a pattern matches more than one character that begins at a specific position, CALL PRXSUBSTR selects the longest match.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 53.

Comparisons

CALL PRXSUBSTR performs the same matching as PRXMATCH, but CALL PRXSUBSTR also enables you to use the *length* argument to receive more information about the match.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. For a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Examples

Example 1: Finding the Position and Length of a Substring

This example searches a string for a substring and returns its position and length in the string.

```
data _null_;
      /* Use PRXPARSE to compile the Perl regular expression. */
   patternID = prxparse('/world/');
      /* Use PRXSUBSTR to find the position and length of the string.
*/
   call prxsubstr(patternID, 'Hello world!', position, length);
   put position= length=;
run;
```

SAS writes the following results to the log:

```
position=7 length=5
```

Example 2: Finding a Match in a Substring

This example searches for addresses that contain avenue, drive, or road and extracts the text that was found.

```
data _null_;
   if _N_ = 1 then
   do;
      retain ExpressionID;
         /* The i option specifies a case insensitive search. */
      pattern = "/ave|avenue|dr|drive|rd|road/i";
      ExpressionID = prxparse(pattern);
   end;
   input street $80.;
   call prxsubstr(ExpressionID, street, position, length);
   if position ^= 0 then
   do;
      match = substr(street, position, length);
      put match:$QUOTE. "found in " street:$QUOTE.;
   end;
   datalines;
153 First Street
```

```
6789 64th Ave
4 Moritz Road
7493 Wilkes Place
;
run;
```

SAS writes the following results to the log:

```
"Ave" found in "6789 64th Ave"
   "Road" found in "4 Moritz Road"
```

See Also

Functions:

- “PRXCHANGE Function” on page 1261
- “PRXMATCH Function” on page 1267
- “PRXPAREN Function” on page 1272
- “PRXPARSE Function” on page 1274
- “PRXPOSN Function” on page 1276

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXPOSN Routine” on page 289

CALL RANBIN Routine

Returns a random variate from a binomial distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANBIN(*seed*, *n*, *p*, *x*);

Required Arguments

seed
is the seed value. A new value for *seed* is returned each time CALL RANBIN is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See Seed Values on page 20 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 21 for more information about seed values

n
is an integer number of independent Bernoulli trials.

Range $n > 0$

p
is a numeric probability of a success parameter.

Range $0<p<1$

x
is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANBIN is executed.

Details

The CALL RANBIN routine updates *seed* and returns a variate *x* that is generated from a binomial distribution with mean *np* and variance $np(1-p)$. If $n \leq 50$, $np \leq 5$, or $n(1-p) \leq 5$, SAS uses an inverse transform method applied to a RANUNI uniform variate. If $n>50$, $np>5$, and $n(1-p)>5$, SAS uses the normal approximation to the binomial distribution. In that case, the Box-Muller transformation of RANUNI uniform variates is used.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

Comparisons

The CALL RANBIN routine gives greater control of the seed and random number streams than does the RANBIN function.

Example

This example uses the CALL RANBIN routine.

```
data u1 (keep=x);
   seed=104;
   do i=1 to 5;
      call ranbin(seed, 2000, 0.2 ,x);
      output;
   end;
   call symputx('seed', seed);
run;

data u2 (keep=x);
   seed=&seed;
   do i=1 to 5;
      call ranbin(seed, 2000, 0.2 ,x);
      output;
   end;
run;

data all;
   set u1 u2;
   z=ranbin(104, 2000, 0.2);
run;

proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.2 *Output from the CALL RANBIN Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	423	423
2	418	418
3	403	403
4	394	394
5	429	429
6	369	369
7	413	413
8	417	417
9	400	400
10	383	383

See Also

Functions:

- “RANBIN Function” on page 1301
- “RAND Function” on page 1304

CALL RANCAU Routine

Returns a random variate from a Cauchy distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANCAU(*seed*, *x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANCAU is executed.

Range *seed* < $2^{31} - 1$

Note If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See For more information about seed values, Seed Values on page 20 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 21

x

is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANCAU is executed.

Details

The CALL RANCAU routine updates *seed* and returns a variate *x* that is generated from a Cauchy distribution that has a location parameter of 0 and a scale parameter of 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

An acceptance-rejection procedure applied to RANUNI uniform variates is used. If u and v are independent uniform (−1/2, 1/2) variables and $u^2+v^2 \leq 1/4$, then u/v is a Cauchy variate.

For more information about random number functions and CALL routines, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20.

Comparisons

The CALL RANCAU routine gives greater control of the seed and random number streams than does the RANCAU function.

Example

This example uses the CALL RANCAU routine.

```
data case;
   retain Seed_1 Seed_2 Seed_3 45;
   do i=1 to 10;
      call rancau(Seed_1, X1);
      call rancau(Seed_2, X2);
      X3=rancau(Seed_3);
      if i=5 then
          do;
             Seed_2=18;
             Seed_3=18;
          end;
      output;
   end;
run;

proc print;
   id i;
   var Seed_1-Seed_3 X1-X3;
run;
```

Output 3.3 *Output from the CALL RANCAU Routine*

The SAS System

i	Seed_1	Seed_2	Seed_3	X1	X2	X3
1	1404437564	1404437564	45	-1.14736	-1.14736	-1.14736
2	1326029789	1326029789	45	-0.23735	-0.23735	-0.23735
3	1988843719	1988843719	45	-0.15474	-0.15474	-0.15474
4	1233028129	1233028129	45	4.97935	4.97935	4.97935
5	50049159	18	18	0.20402	0.20402	0.20402
6	802575599	991271755	18	3.43645	4.44427	3.43645
7	1233458739	1437043694	18	6.32808	-1.79200	6.32808
8	52428589	959908645	18	0.18815	-1.67610	0.18815
9	1216356463	1225034217	18	0.80689	3.88391	0.80689
10	1711885541	425626811	18	0.92971	-1.31309	0.92971

Here is another example of the CALL RANCAU routine.

```
data u1(keep=x);
   seed=104;
   do i=1 to 5;
      call rancau(seed, X);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed=&seed;
   do i=1 to 5;
      call rancau(seed, X);
      output;
   end;
run;

data all;
   set u1 u2;
   z=rancau(104);
run;

proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.4 *Output from the CALL RANCAU Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	-0.6780	-0.6780
2	0.1712	0.1712
3	1.1372	1.1372
4	0.1478	0.1478
5	16.6536	16.6536
6	0.0747	0.0747
7	-0.5872	-0.5872
8	1.4713	1.4713
9	0.1792	0.1792
10	-0.0473	-0.0473

See Also

Functions:

- "RANCAU Function" on page 1302
- "RAND Function" on page 1304

CALL RANCOMB Routine

Permutes the values of the arguments and returns a random combination of *k* out of *n* values.

Category: Combinatorial

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANCOMB(*seed*, *k*, *variable-1* <, *variable-2, ..., variable-n*>);

Required Arguments

seed
is a numeric variable that contains the random number seed. For more information about seeds, see "Concepts" on page 20.

k
is the number of values that you want to have in the random combination.

variable
specifies all numeric variables or all character variables that have the same length. *K* values of these variables are randomly permuted.

Details

The Basics

If there are *n* variables, CALL RANCOMB permutes the values of the variables in such a way that the first *k* values are sorted in ascending order and form a random combination of *k* out of the *n* values. That is, all $n!/(k!(n-k)!)$ combinations of *k* out of the *n* values are equally likely to be returned as the first *k* values.

If an error occurs during the execution of the CALL RANCOMB routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR and &SYSINFO are set to 0.

Using CALL RANCOMB with Macros

You can call the CALL RANCOMB routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. However, if the first *k* values that are returned include both character and numeric values, those values are not sorted. If %SYSCALL identifies an argument as numeric, %SYSCALL reformats the returned value.

Examples

Example 1: Using CALL RANCOMB in a DATA Step

This example shows how to generate random combinations of given values by using the CALL RANCOMB routine in a DATA step.

```
data _null_;
   array x x1-x5 (1 2 3 4 5);
   seed=1234567890123;
   do n=1 to 10;
      call rancomb(seed, 3, of x1-x5);
      put seed= @20 ' x= ' x1-x3;
```

```
   end;
run;
```

SAS writes the following results to the log:

```
seed=1332351321     x= 2 4 5
seed=829042065      x= 1 3 4
seed=767738639      x= 2 3 5
seed=1280236105     x= 2 4 5
seed=670350431      x= 1 2 5
seed=1956939964     x= 2 3 4
seed=353939815      x= 1 3 4
seed=1996660805     x= 1 2 5
seed=1835940555     x= 2 4 5
seed=910897519      x= 2 3 4
```

Example 2: Using CALL RANCOMB with a Macro

Here is an example of the CALL RANCOMB routine that is used with macros.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=3;
   %let seed=12345;
   %do j=1 %to 10;
      %syscall rancomb(seed, k, x1, x2, x3, x4, x5);
      %put j=&j    &x1 &x2 &x3;
   %end;
   %mend;

%test;
```

SAS writes the following results to the log:

```
j=1   -0.1234 hippopotamus zebra
j=2   hippopotamus -0.1234 10000000000
j=3   hippopotamus ant zebra
j=4   -0.1234 zebra ant
j=5   -0.1234 ant hippopotamus
j=6   10000000000 hippopotamus ant
j=7   10000000000 hippopotamus ant
j=8   ant 10000000000 -0.1234
j=9   zebra -0.1234 10000000000
j=10   zebra hippopotamus 10000000000
```

See Also

Functions:

- “RAND Function” on page 1304

CALL Routines:

- "CALL ALLPERM Routine" on page 222
- "CALL RANPERK Routine" on page 312
- "CALL RANPERM Routine" on page 314

CALL RANEXP Routine

Returns a random variate from an exponential distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANEXP(*seed*, *x*);

Required Arguments

seed
is the seed value. A new value for *seed* is returned each time CALL RANEXP is executed.

Range $seed < 2^{31} - 1$

Note If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See Seed Values on page 20 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 21 for more information about seed values

x
is a numeric variable. A new value for the random variate *x* is returned each time CALL RANEXP is executed.

Details

The CALL RANEXP routine updates *seed* and returns a variate *x* that is generated from an exponential distribution that has a parameter of 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

The CALL RANEXP routine uses an inverse transform method applied to a RANUNI uniform variate.

For information about random number functions and CALL routines, see “Using Random-Number Functions and CALL Routines in the DATA Step” on page 20 .

Comparisons

The CALL RANEXP routine gives greater control of the seed and random number streams than does the RANEXP function.

Example

This example uses the CALL RANEXP routine.

```
data u1(keep=x);
   seed=104;
   do i=1 to 5;
      call ranexp(seed, x);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed=&seed;
   do i=1 to 5;
      call ranexp(seed, x);
      output;
   end;
run;

data all;
   set u1 u2;
   z=ranexp(104);
run;

proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.5 *Output from the CALL RANEXP Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	1.44347	1.44347
2	0.11740	0.11740
3	0.54175	0.54175
4	0.02280	0.02280
5	0.16645	0.16645

See Also

Functions:

- "RAND Function" on page 1304
- "RANEXP Function" on page 1325

CALL RANGAM Routine

Returns a random variate from a gamma distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANGAM(*seed*, *a*, *x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANGAM is executed.

Range $seed < 2^{31} - 1$

Note If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See Seed Values on page 20 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 21

a

is a numeric shape parameter.

Range $a > 0$

x

is a numeric variable. A new value for the random variate *x* is returned each time CALL RANGAM is executed.

Details

The CALL RANGAM routine updates *seed* and returns a variate *x* that is generated from a gamma distribution with parameter *a*.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

For $a>1$, an acceptance-rejection method by Cheng (1977) is used . For $a \leq 1$, an acceptance-rejection method by Fishman (1978) is used. For more information, see "References" on page 1609 .

For information about random number functions and CALL routines, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20.

Comparisons

The CALL RANGAM routine gives greater control of the seed and random number streams than does the RANGAM function.

Examples

Example 1

This example uses the CALL RANGAM routine.

```
data u1(keep=x);
   seed=104;
   do i=1 to 5;
      call rangam(seed, 1, x);
      output;
   end;
```

```
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed=&seed
   do i=1 to 5;
      call rangam(seed, 1, x);
      output;
   end;
run;

data all;
   set u1 u2;
   z=rangam(104, 1);
run;

proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.6 *Output from the CALL RANGAM Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	1.44347	1.44347
2	0.11740	0.11740
3	0.54175	0.54175
4	0.02280	0.02280
5	0.16645	0.16645

Example 2

Here is another example that uses the CALL RANGAM routine.

```
data case;
   retain Seed_1 Seed_2 Seed_3 45;
   a=2;
   do i=1 to 10;
      call rangam(Seed_1, a, X1);
      call rangam(Seed_2, a, X2);
      X3=rangam(Seed_3, a);
      if i=5 then
         do;
            Seed_2=18;
            Seed_3=18;
         end;
      output;
```

```
   end;
run;

proc print;
   id i;
   var Seed_1-Seed_3 X1-X3;
run;
```

Output 3.7 *Output from the CALL RANGAM Routine*

The SAS System

i	Seed_1	Seed_2	Seed_3	X1	X2	X3
1	1404437564	1404437564	45	1.30569	1.30569	1.30569
2	1326029789	1326029789	45	1.87514	1.87514	1.87514
3	1988843719	1988843719	45	1.71597	1.71597	1.71597
4	50049159	50049159	45	1.59304	1.59304	1.59304
5	802575599	18	18	0.43342	0.43342	0.43342
6	100573943	991271755	18	1.11812	1.32646	1.11812
7	1986749826	1437043694	18	0.68415	0.88806	0.68415
8	52428589	959908645	18	1.62296	2.46091	1.62296
9	1216356463	1225034217	18	2.26455	4.06596	2.26455
10	805366679	425626811	18	2.16723	6.94703	2.16723

Changing Seed_2 for the CALL RANGAM statement when I=5 forces the stream of the variates for X2 to deviate from the stream of the variates for X1. However, changing Seed_3 on the RANGAM function has no effect.

See Also

Functions:

- “RAND Function” on page 1304
- “RANGAM Function” on page 1327

CALL RANNOR Routine

Returns a random variate from a normal distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANNOR(*seed*, *x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANNOR is executed.

Range *seed* < $2^{31} - 1$

Note If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See Seed Values on page 20 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 21

x

is a numeric variable. A new value for the random variate *x* is returned each time CALL RANNOR is executed.

Details

The CALL RANNOR routine updates *seed* and returns a variate *x* that is generated from a normal distribution, with mean 0 and variance 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

The CALL RANNOR routine uses the Box-Muller transformation of RANUNI uniform variates.

For information about random number functions and CALL routines, see “Using Random-Number Functions and CALL Routines in the DATA Step” on page 20 .

Comparisons

The CALL RANNOR routine gives greater control of the seed and random number streams than does the RANNOR function.

Example

This example uses the CALL RANNOR routine.

```
data u1(keep=x);
   seed=104;
   do i=1 to 5;
      call rannor(seed, X);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed=&seed;
   do i=1 to 5;
      call rannor(seed, X);
      output;
   end;
run;

data all;
   set u1 u2;
   z=rannor(104);
run;

proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.8 *Output from the CALL RANNOR Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	1.30390	1.30390
2	1.03049	1.03049
3	0.19491	0.19491
4	-0.34987	-0.34987
5	1.64273	1.64273
6	-1.75842	-1.75842
7	0.75080	0.75080
8	0.94375	0.94375
9	0.02436	0.02436
10	-0.97256	-0.97256

See Also

Functions:

- “RAND Function” on page 1304
- “RANNOR Function” on page 1331

CALL RANPERK Routine

Permutes the values of the arguments and returns a random permutation of *k* out of *n* values.

Category: Combinatorial

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANPERK(*seed*, *k*, *variable-1* <, *variable-2, ..., variable-n*>);

Required Arguments

seed
is a numeric variable that contains the random number seed. For more information about seeds, see “Concepts” on page 20.

k
is the number of values that you want to have in the random permutation.

variable
specifies all numeric variables or all character variables that have the same length. *K* values of these variables are randomly permuted.

Details

Using CALL RANPERK with Macros

You can call the RANPERK routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL RANPERK routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR and &SYSINFO are set to 0.

Examples

Example 1: Using CALL RANPERK in a DATA Step

The following example shows how to generate random permutations of given values by using the CALL RANPERK routine.

```
data _null_;
   array x x1-x5 (1 2 3 4 5);
   seed=1234567890123;
   do n=1 to 10;
      call ranperk(seed, 3, of x1-x5);
      put seed=@20 ' x= ' x1-x3;
   end;
run;
```

SAS writes the following results to the log:

```
seed=1332351321     x= 5 4 2
seed=829042065      x= 4 1 3
seed=767738639      x= 5 1 2
seed=1280236105     x= 3 2 5
seed=670350431      x= 4 3 5
seed=1956939964     x= 3 1 2
seed=353939815      x= 4 2 1
seed=1996660805     x= 3 4 5
seed=1835940555     x= 5 1 4
seed=910897519      x= 5 1 2
```

Example 2: Using CALL RANPERK with a Macro

Here is an example of the CALL RANPERK routine that is used with macros.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=3;
   %let seed=12345;
   %do j=1 %to 10;
      %syscall ranperk(seed, k, x1, x2, x3, x4, x5);
      %put j=&j   &x1 &x2 &x3;
   %end;
   %mend;

%test;
```

SAS writes the following results to the log:

```
j=1   -0.1234 hippopotamus zebra
j=2   hippopotamus -0.1234 10000000000
j=3   hippopotamus ant zebra
j=4   -0.1234 zebra ant
j=5   -0.1234 ant hippopotamus
j=6   10000000000 hippopotamus ant
j=7   10000000000 hippopotamus ant
j=8   ant 10000000000 -0.1234
j=9   zebra -0.1234 10000000000
j=10   zebra hippopotamus 10000000000
```

See Also

Functions:

- “RAND Function” on page 1304

Call Routines:

- “CALL ALLPERM Routine” on page 222
- “CALL RANPERM Routine” on page 314

CALL RANPERM Routine

Randomly permutes the values of the arguments.

Category: Combinatorial

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANPERM(*seed*, *variable-1* <, *variable-2, ..., variable-n*>);

Required Arguments

seed
is a numeric variable that contains the random number seed. For more information about seeds, see “Concepts” on page 20.

variable
specifies all numeric variables or all character variables that have the same length. The values of these variables are randomly permuted.

Details

Using CALL RANPERM with Macros

You can call the RANPERM routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL RANPERM routine, both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, &SYSERR and &SYSINFO are set to 0.

Examples

Example 1: Using CALL RANPERM in a DATA Step

This example generates random permutations of given values by using the CALL RANPERM routine.

```
data _null_;
         array x x1-x4 (1 2 3 4);
         seed=1234567890123;
         do n=1 to 10;
            call ranperm(seed, of x1-x4);
         put seed=@20 ' x= ' x1-x4;
         end;
run;
```

SAS writes the following results to the log:

```
seed=1332351321     x= 1 3 2 4
seed=829042065      x= 3 4 2 1
seed=767738639      x= 4 2 3 1
seed=1280236105     x= 1 2 4 3
seed=670350431      x= 2 1 4 3
seed=1956939964     x= 2 4 3 1
seed=353939815      x= 4 1 2 3
seed=1996660805     x= 4 3 1 2
seed=1835940555     x= 4 3 2 1
seed=910897519      x= 3 2 1 4
```

Example 2: Using CALL RANPERM with a Macro

Here is an example of the CALL RANPERM routine that is used with the %SYSCALL macro.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
```

```
      %let x3=1e10;
      %let x4=hippopotamus;
      %let x5=zebra;
      %let seed=12345;
      %do j=1 %to 10;
         %syscall ranperm(seed, x1, x2, x3, x4, x5);
         %put j=&j    &x1 &x2 &x3;
      %end;
      %mend;

   %test;
```

SAS writes the following results to the log:

```
j=1   zebra ant hippopotamus
j=2   10000000000 ant -0.1234
j=3   -0.1234 10000000000 ant
j=4   hippopotamus ant zebra
j=5   -0.1234 zebra 10000000000
j=6   -0.1234 hippopotamus ant
j=7   zebra ant -0.1234
j=8   -0.1234 hippopotamus ant
j=9   ant -0.1234 hippopotamus
j=10   -0.1234 zebra 10000000000
```

See Also

Functions:

- “RAND Function” on page 1304

CALL Routines:

- “CALL ALLPERM Routine” on page 222
- “CALL RANPERK Routine” on page 312

CALL RANPOI Routine

Returns a random variate from a Poisson distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANPOI(*seed*, *m*, *x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANPOI is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See "Concepts" on page 20 and "Pseudorandom Numbers" on page 21

m

is a numeric mean parameter.

Range $m \geq 0$

x

is a numeric variable. A new value for the random variate *x* is returned each time CALL RANPOI is executed.

Details

The CALL RANPOI routine updates *seed* and returns a variate *x* that is generated from a Poisson distribution, with mean *m*.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

For $m < 85$, an inverse transform method applied to a RANUNI uniform variate is used (Fishman, 1976; see "References" on page 1609.) For $m \geq 85$, the normal approximation of a Poisson random variable is used. To expedite execution, internal variables are calculated only on initial calls (that is, with each new *m*).

For information about random number functions and CALL routines, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20 .

Comparisons

The CALL RANPOI routine gives greater control of the seed and random number streams than does the RANPOI function.

Example

This example uses the CALL RANPOI routine.

```
data u1(keep=x);
   seed=104;
```

```
   do i=1 to 5;
      call ranpoi(seed, 2000, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed=&seed;
   do i=1 to 5;
      call ranpoi(seed, 2000, x);
      output;
   end;
run;
data all;
   set u1 u2;
   z=ranpoi(104, 2000);
run;
proc print label;
   label x='Separate Streams' z = 'Single Stream';
run;
```

Output 3.9 *Output from the CALL RANPOI Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	2058	2058
2	2046	2046
3	2009	2009
4	1984	1984
5	2073	2073
6	1921	1921
7	2034	2034
8	2042	2042
9	2001	2001
10	1957	1957

See Also

Functions:

- “RAND Function” on page 1304
- “RANPOI Function” on page 1332

CALL RANTBL Routine

Returns a random variate from a tabled probability distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANTBL(*seed*, p_1, ...p_i, ...p_n, *x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANTBL is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See "Concepts" on page 20 and "Pseudorandom Numbers" on page 21

p_i

is a numeric SAS value.

Range $0 \leq p_i \leq 1$ for $0 < i \leq n$

x

is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANTBL is executed.

Details

The CALL RANTBL routine updates *seed* and returns a variate *x* generated from the probability mass function defined by p_1 through p_n.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

An inverse transform method applied to a RANUNI uniform variate is used. The CALL RANTBL routine returns this data:

$$1 \ with\ probability\ p_1$$
$$2 \ with\ probability\ p_2$$
$$.$$
$$.$$
$$.$$
$$n \ with\ probability\ p_n$$
$$n+1 \ with\ probability\ 1 - \sum_{i=1}^{n} p_i \ if \sum_{i=1}^{n} p_i \le 1$$

If, for some index $j<n$,

$$\sum_{i=1}^{j} p_i \ge 1$$

RANTBL returns only the indices 1 through *j*, with the probability of occurrence of the index *j* equal to

$$1 - \sum_{i=1}^{j-1} p_{\mathrm{i}}$$

For information about random number functions and CALL routines, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20.

Comparisons

The CALL RANTBL routine gives greater control of the seed and random number streams than does the RANTBL function.

Example

This example uses the CALL RANTBL routine.

```
data u1(keep=x);
   seed=104;
   do i=1 to 5;
      call rantbl(seed, .02, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed=&seed;
   do i=1 to 5;
      call rantbl(seed, .02, x);
      output;
   end;
run;
data all;
   set u1 u2;
```

```
   z=rantbl(104, .02);
run;
proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.10 *Output from the CALL RANTBL Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	2	2
2	2	2
3	2	2
4	2	2
5	2	2
6	2	2
7	2	2
8	2	2
9	2	2
10	2	2

See Also

Functions:

- “RAND Function” on page 1304
- “RANTBL Function” on page 1334

CALL RANTRI Routine

Returns a random variate from a triangular distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANTRI(*seed*, *h*, *x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANTRI is executed.

Range $seed < 2^{31} - 1$

Note If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See "Concepts" on page 20 and "Pseudorandom Numbers" on page 21

h

is a numeric SAS value.

Range $0<h<1$

x

is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANTRI is executed.

Details

The CALL RANTRI routine updates *seed* and returns a variate *x* generated from a triangular distribution on the interval (0,1) with parameter *h*, which is the modal value of the distribution.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

The CALL RANTRI routine uses an inverse transform method applied to a RANUNI uniform variate.

For information about random number functions and CALL routines, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20.

Comparisons

The CALL RANTRI routine gives greater control of the seed and random number streams than does the RANTRI function.

Example

This example uses the CALL RANTRI routine.

```
data u1(keep=x);
   seed=104;
   do i=1 to 5;
      call rantri(seed, .5, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed=&seed;
   do i=1 to 5;
      call rantri(seed, .5, x);
      output;
   end;
run;
data all;
   set u1 u2;
   z=rantri(104, .5);
run;
proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.11 *Output from the CALL RANTRI Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	0.34359	0.34359
2	0.76466	0.76466
3	0.54269	0.54269
4	0.89384	0.89384
5	0.72311	0.72311
6	0.68763	0.68763
7	0.48468	0.48468
8	0.38467	0.38467
9	0.29881	0.29881
10	0.80369	0.80369

See Also

Functions:

- "RAND Function" on page 1304
- "RANTRI Function" on page 1335

CALL RANUNI Routine

Returns a random variate from a uniform distribution.

Category: Random Number

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL RANUNI(*seed*, *x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANUNI is executed.

Range *seed* < $2^{31} - 1$

Tip If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See "Concepts" on page 20 and "Pseudorandom Numbers" on page 21

x

is a numeric variable. A new value for the random variate *x* is returned each time CALL RANUNI is executed.

Details

The CALL RANUNI routine updates *seed* and returns a variate *x* that is generated from the uniform distribution on the interval (0,1), using a prime modulus multiplicative generator with modulus $2^{31}-1$ and multiplier 397204094 (Fishman and Moore 1982). For more information, see "References" on page 1609.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or subsequent, DATA steps.

For information about random number functions and CALL routines, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20.

Comparisons

The CALL RANUNI routine gives greater control of the seed and random number streams than does the RANUNI function.

Example: Using the CALL RANUNI Routine

This example uses the CALL RANUNI routine.

```
data u1(keep=x);
   seed=104;
   do i=1 to 5;
      call ranuni(seed, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed=&seed;
   do i=1 to 5;
      call ranuni(seed, x);
      output;
   end;
run;
data all;
   set u1 u2;
   z=ranuni(104);
run;
proc print label;
   label x='Separate Streams' z='Single Stream';
run;
```

Output 3.12 *Output from the CALL RANUNI Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	0.23611	0.23611
2	0.88923	0.88923
3	0.58173	0.58173
4	0.97746	0.97746
5	0.84667	0.84667
6	0.80484	0.80484
7	0.46983	0.46983
8	0.29594	0.29594
9	0.17858	0.17858
10	0.92292	0.92292

See Also

Functions:

- “RAND Function” on page 1304
- “RANUNI Function” on page 1337

CALL SCAN Routine

Returns the position and length of the *n*th word from a character string.

Category: Character

Interaction: When invoked by the %SYSCALL macro statement, CALL SCAN removes the quotation marks from its arguments. For more information, see “Invoking CALL Routines and the %SYSCALL Macro Statement” .

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL SCAN(<*string*>, *count*, *position*, *length* <, <*character-list*> <, <*modifier(s)*>>>);

Required Arguments

count
is a nonzero numeric constant, variable, or expression that has an integer value that specifies the number of the word in the character string that you want the CALL SCAN routine to select. For example, a value of 1 indicates the first word, a value of 2 indicates the second word, and so on. The following rules apply:

- If *count* is positive, CALL SCAN counts words from left to right in the character string.
- If *count* is negative, CALL SCAN counts words from right to left in the character string.

position
specifies a numeric variable in which the position of the word is returned. If *count* exceeds the number of words in the string, the value that is returned in *position* is 0. If *count* is 0 or missing, the value that is returned in *position* is missing.

length
specifies a numeric variable in which the length of the word is returned. If *count* exceeds the number of words in the string, the value that is returned in *length* is 0. If *count* is 0 or missing, the value that is returned in *length* is missing.

Optional Arguments

string
specifies a character constant, variable, or expression.

character-list
specifies an optional character constant, variable, or expression that initializes a list of characters. This list determines which characters are used as the delimiters that separate words. The following rules apply:

- By default, all characters in *character-list* are used as delimiters.
- If you specify the K modifier in the *modifier* argument, all characters that are not in *character-list* are used as delimiters.

Tip You can add more characters to *character-list* by using other modifiers.

modifier
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the CALL SCAN routine. Blanks are ignored. You can 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 (that is, valid first characters in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters. Graphic characters are those 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, 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 0. If the M modifier is not specified, 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 in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *character-list* and *modifier* arguments only once, rather than every time the CALL SCAN routine is called. Using the O modifier in the DATA step can make CALL 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 CALL SCAN routine in your SAS code and does not cause all instances of the CALL SCAN routine 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, scanning from left to right produces different words than scanning from right to left.
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 *character-list* arguments. If you want to remove trailing blanks from just one character argument instead of both character arguments, use the TRIM function instead of the CALL SCAN routine 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.

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

Definition of "Delimiter" and "Word"

A *delimiter* is any of several characters that are used to separate words. You can specify the delimiters in the *character-list* and *modifier* arguments.

If you specify the Q modifier, delimiters inside substrings that are enclosed in quotation marks are ignored.

In the CALL SCAN routine, "word" refers to a substring that has all 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

A word can have a length of 0 if there are delimiters at the beginning or end of the string, or if the string contains two or more consecutive delimiters. However, the CALL SCAN routine ignores words that have a length of 0 unless you specify the M modifier.

Using Default Delimiters in an ASCII Environment

If you use the CALL SCAN routine with only four arguments and your computer uses ASCII characters, here are the default delimiters:

blank ! $ % & ( ) * + , - . / ; < ^ |

In ASCII environments that do not contain the ^ character, the CALL SCAN routine uses the ~ character instead.

If you use the *modifier* argument without specifying any characters as delimiters, the only delimiters used are those that are defined by the *modifier* argument. In this case, the list of default delimiters for ASCII environments are not used. In other words, modifiers add to the list of delimiters that are explicitly specified by the *character-list* argument. Modifiers do not add to the list of default modifiers.

Using the CALL SCAN Routine 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 CALL SCAN routine returns a positive position and a length of 0 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.

If you specify a count that is greater in absolute value than the number of words in the string, the CALL SCAN routine returns a position and length of 0.

Using the CALL SCAN Routine 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 CALL SCAN routine behaves in these ways:

- ignores delimiters at the beginning or end of the string
- 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, the CALL SCAN routine returns a position and length of 0.

Finding the Word as a Character String

To find the designated word as a character string after calling the CALL SCAN routine, use the SUBSTRN function with the *string*, *position*, and *length* arguments.

```
substrn(string, position, length);
```

Because CALL SCAN can return a length of 0, using the SUBSTR function can cause an error.

Using Null Arguments

The CALL SCAN routine allows character arguments to be null. Null arguments are treated as character strings with a length of 0. Numeric arguments cannot be null.

Processing SBCS and DBCS Data

CALL SCAN is designed to process SBCS data, but it can also process DBCS data. Here are the criteria:

- If *string* is not declared as varchar and you are processing single-byte data, CALL SCAN processes SBCS.
- If *string* is declared as varchar and you are processing multi-byte data, CALL SCAN processes DBCS.

Examples

Example 1: Scanning for a Word in a String

This example shows how you can use the CALL SCAN routine to find the position and length of a word in a string.

```
data artists;
   input string $60.;
   drop string;
   do i=1 to 99;
      call scan(string, i, position, length);
      if not position then leave;
      Name=substrn(string, position, length);
      output;
   end;
   datalines;
Picasso Toulouse-Lautrec Turner "Van Gogh" Velazquez
;
proc print data=artists;
run;
```

Output 3.13 *Output from Scanning for a Word in a String*

The SAS System

Obs	i	position	length	Name
1	1	1	7	Picasso
2	2	9	8	Toulouse
3	3	18	7	Lautrec
4	4	26	6	Turner
5	5	33	4	"Van
6	6	38	5	Gogh"
7	7	44	9	Velazquez

Example 2: Finding the First and Last Words in a String

This example scans a string for the first and last words. Note the following information:

- A negative count instructs the CALL SCAN routine 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, so no words are found.

```
data firstlast;
   input String $60.;
```

```
   call scan(string, 1, First_Pos, First_Length);
   First_Word=substrn(string, First_Pos, First_Length);
   call scan(string, -1, Last_Pos, Last_Length);
   Last_Word=substrn(string, Last_Pos, Last_Length);
   datalines4;
Jack and Jill
& Bob & Carol & Ted & Alice &
Leonardo
! $ % & ( ) * + , - . / ;
;;;;
proc print data=firstlast;
   var First: Last:;
run;
```

Output 3.14 *Output from Finding the First and Last Words in a String*

The SAS System

Obs	First_Pos	First_Length	First_Word	Last_Pos	Last_Length	Last_Word
1	1	4	Jack	10	4	Jill
2	3	3	Bob	23	5	Alice
3	1	8	Leonardo	1	8	Leonardo
4	0	0		0	0	

Example 3: Finding All Words in a String without Using the M Modifier

The following example scans a string from left to right until no more words are found. Because the M modifier is not used, the CALL SCAN routine does not return any words that have a length of 0. Because blanks are included among the default delimiters, the CALL SCAN routine returns a position or length of 0 only when the count exceeds the number of words in the string. The loop can be stopped when the returned position is less than or equal to 0. It is safer to use an inequality comparison to end the loop rather than a strict equality comparison with 0, in case an error causes the position to be missing. (In SAS, a missing value is considered to have a lesser value than any nonmissing value.)

```
data all;
   length word $20;
   drop string;
   string=' The quick brown fox jumps over the lazy dog.   ';
   do until(position <= 0);
      count+1;
      call scan(string, count, position, length);
      word=substrn(string, position, length);
      output;
   end;
run;
proc print data=all noobs;
   var count position length word;
```

```
run;
```

Output 3.15 *Output from Finding All Words in a String without Using the M Modifier*

The SAS System

count	position	length	word
1	2	3	The
2	6	5	quick
3	12	5	brown
4	18	3	fox
5	22	5	jumps
6	28	4	over
7	33	3	the
8	37	4	lazy
9	42	3	dog
10	0	0	

Example 4: Finding All Words in a String by Using the M and O Modifiers

The following 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 CALL SCAN routine to return words that have a length of 0.

The O modifier is used for efficiency because the delimiters and modifiers are the same in every call to the CALL SCAN routine.

```
data comma;
   length word $30;
   string=',leading,  trailing,and multiple,,delimiters, ,';
   do until(position <= 0);
      count + 1;
      call scan(string, count, position, length, ', ', 'mo');
      word=substrn(string, position, length);
      output;
   end;
run;
proc print data=comma noobs;
   var count position length word;
run;
```

Output 3.16 *Output from Finding All Words in a String by Using the M and O Modifiers*

The SAS System

count	position	length	word
1	1	0	
2	2	7	leading
3	10	10	trailing
4	21	12	and multiple
5	34	0	
6	35	10	delimiters
7	46	0	
8	47	0	
9	0	0	

Example 5: Using Comma-Separated Values, Substrings in Quotation Marks, and the O Modifier

The following example uses the CALL SCAN routine with the O modifier and a comma as a delimiter.

The O modifier is used for efficiency because in each call of the CALL SCAN routine, the delimiters, and modifiers do not change.

```
data test;
   length word word_r $30;
   string='He said, "She said, ""No!""", not "Yes!"';
   do until(position <= 0);
      count + 1;
      call scan(string, count, position, length, ', ', 'oq');
      word=substrn(string, position, length);
      output;
   end;
run;
proc print data=test noobs;
   var count position length word;
run;
```

Output 3.17 *Output from Comma-Separated Values and Substrings in Quotation Marks*

The SAS System

count	position	length	word
1	1	7	He said
2	9	20	"She said, ""No!"""
3	30	11	not "Yes!"
4	0	0	

Example 6: Finding Substrings of Digits by Using the D and K Modifiers

The following example finds substrings of digits. The *character-list* argument is null, and 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.

```
data digits;
   length digits $20;
   string='Call (800) 555-1234 now!';
   do until(position <= 0);
      count+1;
      call scan(string, count, position, length, , 'dko');
      digits=substrn(string, position, length);
      output;
   end;
run;
proc print data=digits noobs;
   var count position length digits;
run;
```

Output 3.18 *Output from Finding Substrings of Digits by Using the D and K Modifiers*

The SAS System

count	position	length	digits
1	7	3	800
2	12	3	555
3	16	4	1234
4	0	0	

See Also

Functions:

- “COUNTW Function” on page 495
- “FINDW Function” on page 692
- “SCAN Function” on page 1366

CALL SET Routine

Links SAS data set variables to DATA step or macro variables that have the same name and data type.

Category: Variable Control

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL SET(*data-set-id*);

Required Argument

data-set-id
is the identifier that is assigned by the OPEN function when the data set is opened.

Details

Using SET can significantly reduce the coding that is required for accessing variable values for modification or verification when you use functions to read or to manipulate a SAS file. After a CALL SET, whenever a read is performed from the SAS data set, the values of the corresponding DATA step or macro variables are set to the values of the matching SAS data set variables. If the variable lengths do not match, the values are truncated or padded according to need. If you do not use SET, you must use the GETVARC and GETVARN functions to move values explicitly between data set variables and DATA step or macro variables.

As a general rule, use CALL SET immediately following OPEN if you want to link the data set and the DATA step and macro variables.

Example: Using the CALL SET Routine

This example uses the CALL SET routine.

- The following statements automatically set the values of the macro variables PRICE and STYLE when an observation is fetched:

```
%macro setvar;
   %let dsid=%sysfunc(open(sasuser.houses, i));
      /* No leading ampersand with %SYSCALL */
   %syscall set(dsid);
   %let rc=%sysfunc(fetchobs(&dsid, 10));
   %let rc=%sysfunc(close(&dsid));
%mend setvar;
%global price style;
%setvar
%put _global_;
```

- The %PUT statement writes these lines to the SAS log:

```
GLOBAL PRICE 127150
GLOBAL STYLE CONDO
```

- The following statements obtain the values for the first 10 observations in Sasuser.Houses and store them in MYDATA:

```
data mydata;
      /* create variables for assignment */
      /*by CALL SET */
   length style $8 sqfeet bedrooms baths 8
      street $16 price 8;
   drop rc dsid;
   dsid=open("sasuser.houses", "i");
   call set (dsid);
   do i=1 to 10;
      rc=fetchobs(dsid, i);
      output;
   end;
run;
```

See Also

Functions:

- “FETCH Function” on page 600
- “FETCHOBS Function” on page 602
- “GETVARC Function” on page 765
- “GETVARN Function” on page 768

CALL SLEEP Routine

For a specified period of time, suspends the execution of a program that invokes this CALL routine.

Category: Special

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL SLEEP(*n* <, *unit*>);

Required Argument

n

is a numeric constant, variable, or expression that specifies the number of seconds for which you want to suspend execution of a program.

> **TIP** If you do not use the *unit* argument, *n* is the next integer value that is less than *n*.

Range $n \geq 0$

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, `CALL SLEEP(.25);` does not suspend execution. `CALL SLEEP(30.25);` suspends execution for 30 milliseconds.

Optional Argument

unit

specifies the unit of time in seconds, which is applied to *n*. For example, 1 corresponds to 1 second, .001 corresponds to 1 millisecond, and 5 corresponds to 5 seconds.

Default .001

Details

The CALL SLEEP routine suspends the execution of a program that invokes this CALL routine for a period of time that you specify. The program can be a DATA step, macro, IML, SCL, or anything that can invoke a CALL routine.

Examples

Example 1: Suspending Execution for a Specified Period of Time

This example tells SAS to suspend the execution of the DATA step PAYROLL for 1 minute and 10 seconds.

```
data payroll;
   call sleep(7000, .01);
   ...more SAS statements...
run;
```

Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following example tells SAS to suspend the execution of the DATA step BUDGET until March 1, 2013, at 3:00 AM. SAS calculates the length of the suspension based on the target date and the date and time that the DATA step begins to execute.

```
   data budget;
      sleeptime='01mar2013:03:00'dt-datetime();
      call sleep(sleeptime, 1);
      ...more SAS statements...;
   run;
```

Example 3: Using an Expression to Specify a Specific Period of Time

This example shows how to use an expression in the CALL SLEEP routine.

```
   data _null_;
      call sleep(.5+.8,1);
   run;
```

Here are the results from the log:

Example Code 3.1 *Using an Expression in the CALL SLEEP Routine*

```
NOTE: DATA statement used (Total process time):
      real time           1.30 seconds
      cpu time            0.00 seconds
```

See Also

Functions:

- "SLEEP Function" on page 1393

CALL SOFTMAX Routine

Returns the softmax value.

Category: Mathematical

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL SOFTMAX(*argument* <,*argument*, ..., *argument-n*>);

Required Argument

argument
is numeric.

Restriction The CALL SOFTMAX routine accepts variables only as valid arguments. Do not use a constant or SAS expression because the CALL routine is unable to update these arguments.

Details

The CALL SOFTMAX routine replaces each argument with the softmax value of that argument. For example, x_j is replaced by this equation:

$$\frac{\varepsilon^{x}j}{\sum_{i=1}^{i=n} \varepsilon^{x}i}$$

If any argument contains a missing value, CALL SOFTMAX returns missing values for all the arguments. Upon a successful return, the sum of all the values is equal to 1.

Example

```
data one;
   x=0.5;
   y=-0.5;
   z=1;
   call softmax(x, y, z);
   put x= y= z=;
run;
```

The preceding statements produce these results:

```
x=0.3314989604 y=0.1219516523 z=0.5465493873
```

CALL SORT Routine

Sorts the *variable* values.

Syntax

CALL SORT(*variable*)

Required Argument

variable
specifies a character constant, variable, or expression.

Examples

Example 1

This example, using numeric variables, sorts the variables in ascending order.

```
data _null_;
   x=2; y=3; z=1;
   call sort(x,y,z);
   put x= y= z=;
run;
```

The preceding statements produce these results:

```
x=1 y=2 z=3
```

Example 2

This example, using character variables, sorts the variables in ascending order.

```
data _null_;
length x y z $3;
   x='bb';
   y='ccc';
   z='a';
   call sort(x,y,z);
   put x= y= z=;
run;
```

The preceding statements produce these results:

```
x=a y=bb z=ccc
```

Example 3

This example, using varchar variables, sorts the variables in ascending order.

```
data _null_;
length x y z varchar(*);
   x='bb';
   y='ccc';
   z='a';
   call sort(x,y,z);
   put x= y= z=;
run;
```

The preceding statements produce these results:

```
x=a y=bb z=ccc
```

CALL SORTC Routine

Sorts the values of character arguments.

Category: Sort

Interaction: When invoked by the %SYSCALL macro statement, CALL SORTC removes the quotation marks from its arguments. For more information, see “Invoking CALL Routines and the %SYSCALL Macro Statement” on page 12.

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL SORTC(*variable-1* <, ..., *variable-n*>);

Required Argument

variable
specifies a character variable.

Details

The values of *variable* are sorted in ascending order by the CALL SORTC routine.

Comparisons

The CALL SORTC routine is used with character variables, and the CALL SORTN routine is used with numeric variables.

Example

This example sorts the character variables in the array in ascending order.

```
data _null_;
   array x(8) $10
      ('tweedledum' 'tweedledee' 'baboon' 'baby'
       'humpty' 'dumpty' 'banana' 'babylon');
   call sortc(of x(*));
   put +3 x(*);
run;
```

SAS writes the following results to the log:

```
baboon baby babylon banana dumpty humpty tweedledee tweedledum
```

See Also

CALL Routines:

- “CALL SORTN Routine” on page 344

CALL SORTN Routine

Sorts the values of numeric arguments.

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL SORTN(*variable-1* <, ..., *variable-n*>);

Required Argument

variable
specifies a numeric variable.

Details

The values of *variable* are sorted in ascending order by the CALL SORTN routine.

Comparisons

The CALL SORTN routine is used with numeric variables, and the CALL SORTC routine is used with character variables.

Example

This example sorts the numeric variables in the array in ascending order.

```
data _null_;
   array x(10) (0, ., .a, 1e-12, -1e-8, .z, -37, 123456789, 1e20, 42);
   call sortn(of x(*));
   put +3 x(*);
run;
```

SAS writes the following results to the log:

```
  . A Z -37 -1E-8 0 1E-12 42 123456789 1E20
```

See Also

CALL Routines:

- "CALL SORTC Routine" on page 342

CALL STDIZE Routine

Standardizes the values of one or more variables.

Category: Mathematical

Interaction: When invoked by the %SYSCALL macro statement, CALL STDIZE removes the quotation marks from its arguments. For more information, see "Invoking CALL Routines and the %SYSCALL Macro Statement" on page 12.

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL STDIZE(<*option-1*, *option-2, ...*, *option-n*> *variable-1* <,*variable-2, ...*, *variable-n*>);

Required Argument

variable
is numeric. These values are standardized according to the method that you use.

Optional Arguments

option
specifies a character expression whose values can be uppercase, lowercase, or mixed-case letters. Leading and trailing blanks are ignored. *option* includes the following three categories:

- standardization-options
- VARDEF-options
- miscellaneous-options

Restriction: Use a separate argument for each option because you cannot specify more than one option in a single argument.

Tip: Character expressions can end with an equal sign that is followed by another argument that is a numeric constant, variable, or expression.

See PROC STDIZE in *SAS/STAT 9.3 User's Guide* for information about formulas and other details. The options that are used in CALL STDIZE are the same as those used in PROC STDIZE.

standardization-options

specifies how to compute the location and scale measures that are used to standardize the variables. The following standardization options are available:

ABW= must be followed by an argument that is a numeric expression that specifies the tuning constant.

AGK= must be followed by an argument that is a numeric expression that specifies the proportion of pairs to be included in the estimation of the within-cluster variances.

AHUBER= must be followed by an argument that is a numeric expression that specifies the tuning constant.

AWAVE= must be followed by an argument that is a numeric expression that specifies the tuning constant.

EUCLEN specifies the Euclidean length.

IQR specifies the interquartile range.

L= must be followed by an argument that is a numeric expression with a value greater than or equal to 1 that specifies the power to which differences are to be raised in computing an L(p) or Minkowski metric.

MAD specifies the median absolute deviation from the median.

MAXABS specifies the maximum absolute values.

MEAN specifies the arithmetic mean (average).

MEDIAN specifies the middle number in a set of data that is ordered according to rank.

MIDRANGE specifies the midpoint of the range.

RANGE specifies a range of values.

SPACING= must be followed by an argument that is a numeric expression that specifies the proportion of data to be contained in the spacing.

STD specifies the standard deviation.

SUM specifies the result that you obtain when you add numbers.

USTD specifies the standard deviation about the origin, based on the uncorrected sum of squares.

VARDEF-options

specifies the divisor to be used in the calculation of variances. VARDEF options can have the following values:

DF specifies degrees of freedom.

N specifies the number of observations. The default is DF.

miscellaneous-options

Miscellaneous options can have the following values:

ADD=
: is followed by a numeric argument that specifies a number to add to each value after standardizing and multiplying by the value from the MULT= option. The default value is 0.

FUZZ=
: is followed by a numeric argument that specifies the relative fuzz factor.

MISSING=
: is followed by a numeric argument that specifies a value to be assigned to variables that have a missing value.

MULT=
: is followed by a numeric argument that specifies a number by which to multiply each value after standardizing. The default value is 1.

NORM
: normalizes the scale estimator to be consistent for the standard deviation of a normal distribution. This option affects only the methods AGK=, IQR, MAD, and SPACING=.

PSTAT
: writes the values of the location and scale measures in the log.

REPLACE
: replaces missing values with the value 0 in the standardized data. (This value corresponds to the location measure before standardizing.) To replace missing values by other values, see the MISSING= option.

SNORM
: normalizes the scale estimator to have an expectation of approximately 1 for a standard normal distribution. This option affects only the SPACING= method.

Details

The CALL STDIZE routine transforms one or more arguments that are numeric variables by subtracting a location measure and dividing by a scale measure. You can use a variety of location and scale measures. The default location option is MEAN, and the default scale option is STD.

In addition, you can multiply each standardized value by a constant, and you can add a constant. Here is the final output value: $result = add + mult * \left(\frac{(original - location)}{scale} \right)$.

These are the descriptions of the variables:

result
: specifies the final value that is returned for each variable.

add
: specifies the constant to add (ADD= option).

mult
: specifies the constant to multiply by (MULT= option).

original
: specifies the original input value.

location
specifies the location measure.

scale
specifies the scale measure.

You can replace missing values by any constant. If you do not specify the MISSING= option or the REPLACE option, variables that have missing values are not altered. The initial estimation method for the ABW=, AHUBER=, and AWAVE= methods is MAD. Percentiles are computed using definition 5. For more information about percentile calculations, see SAS Elementary Statistics Procedures in *Base SAS Procedures Guide*.

Comparisons

The CALL STDIZE routine is similar to the STDIZE procedure in the SAS/STAT product. However, the CALL STDIZE routine is primarily useful for standardizing the rows of a SAS data set, whereas the STDIZE procedure can standardize only the columns of a SAS data set. For more information, see PROC STDIZE in *SAS/STAT User's Guide*.

Example

```
data _null_;
   retain x 1 y 2 z 3;
   call stdize(x,y,z);
   put x= y= z=;
run;
```

The preceding statements produce these results:

```
x=-1 y=0 z=1
```

```
data _null_;
   retain w 10 x 11 y 12 z 13;
   call stdize('iqr',w,x,y,z);
   put w= x= y= z=;
run;
```

The preceding statements produce these results:

```
w=-0.75 x=-0.25 y=0.25 z=0.75
```

```
data _null_;
   retain w . x 1 y 2 z 3;
   call stdize('range',w,x,y,z);
   put w= x= y= z=;
run;
```

The preceding statements produce these results:

```
w=. x=0 y=0.5 z=1
```

```
data _null_;
      retain w . x 1 y 2 z 3;
      call stdize('mult=',10,'missing=',-1,'range',w,x,y,z);
      put w= x= y= z=;
   run;
```

The preceding statements produce these results:

```
w=-1 x=0 y=5 z=10
```

CALL STREAM Routine

Specifies a random-number stream to use for subsequent calls to the RAND function.

Category: Random Number

Syntax

CALL STREAM(*key*);

Argument

key
an integer that identifies which stream of pseudorandom numbers from a random-number generator (RNG) is used for subsequent calls to the RAND function.

Range The key value is a nonnegative integer. The range varies among RNGs, but all RNGs support more than one billion keys.

Details

Basics

The STREAM subroutine enables you to create and access multiple copies of an RNG, each of which was initialized separately by a nonnegative integer called a key. After you call the STREAM subroutine, subsequent calls to the RAND function use that stream until a later call to the STREAM subroutine specifies a different key value.

For most applications, you do not need to use the STREAM subroutine. You can generate multiple variables that contain random numbers by using a single stream,

as explained in "CALL STREAMINIT Routine" on page 352. Each stream that you create requires storing the state of an RNG in memory.

In certain circumstances, it is desirable to generate random numbers from multiple independent streams. Here are three use cases:

- A macro that calls a DATA step multiple times. If the DATA step generates random numbers, you might want each iteration of the macro loop to generate a unique stream. For a description of a macro that calls the STREAM subroutine, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20 .
- A program that runs in parallel on multiple nodes of a grid of computers. If the program generates random values, you can use the node ID as the key value in the CALL STREAM routine.
- An existing DATA step that generates random numbers as part of a simulation study or bootstrap computation. You might need to modify the program to generate additional random variables without changing the values of the existing random variables.

For most purposes, different streams are statistically independent, especially streams from the PCG or Threefry RNGs. Different streams are also computationally independent. That is, generating pseudorandom numbers from one stream has no effect on pseudorandom numbers in another stream.

Calling STREAM with a key value of 0 yields the same stream that would be obtained if the STREAM subroutine were never called.

For a hardware-based RNG, CALL STREAM has no effect. A warning message in the log alerts you that the call was ignored.

For more information about multiple streams, see "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20.

Details about the Range for the Argument *key*

The *key* value is a nonnegative integer. The range varies among RNGs, and all RNGs support more than four billion keys.

- For the MTHybrid, MT32, and MT1998 random-number generators (RNGs), the *key* value can be an integer from 0 to $2^{32}-1=4294967295$.
- For the MT64, Threefry2x64, and Threefry4x64 RNGs, the *key* value can be an integer from 0 to $2^{64}-1025=18446744073709549568$.
- For PCG RNG, *key* can be an integer from 0 to $2^{20}-1=1048575$.

Some large integers do not have an exact representation in double-precision floating-point numbers.

If the STREAM routine is called with an invalid key value, the DATA step stops with an error message.

Example: Modify an Existing Program That Generates Random Numbers

The DATA step in this example simulates data for a linear regression model Y = 1 + 2*X1 + E, where E is a variable that contains standard normal random variates. The STREAMINIT subroutine sets the PCG RNG for generating pseudorandom numbers and sets the initial seed. The DATA step then generates X1 with a uniform distribution and E with a normal distribution. The variable Y is linearly related to X1, but random noise is added.

```
data Sim;
call streaminit( 'pcg', 54321 );
do i = 1 to 10;
   X1 = rand( 'uniform' );
   E = rand( 'normal' );
   Y = 1 + 2*X1 + E;
   output;
end;
run;
```

The pseudorandom values for X1 and E depend on the RNG, the seed, the probability distributions, and the order in which you call the RAND function. The values also depend on the stream. Because the DATA step does not call the STREAM subroutine, the stream is the same for CALL STREAM(0). If you decide to modify the program to include a new random variable X2, but you do not want to change the values of X1 and E, you can generate X2 from a separate stream. The stream is independent of the first stream, as shown in this example:

```
data Sim2;
    call streaminit( 'pcg', 54321 );
    do i = 1 to 10;
       call stream( 0 );            * Use the default stream;
       X1 = rand( 'uniform' );      * X1 and E are unchanged;
       E = rand( 'normal' );
       call stream( 1 );            * Use a new stream for X2;
       X2 = rand( 'normal' );
       Y = 1 + 2*X1 + 0.3*X2 + E;
       output;
    end;
run;

proc compare base=Sim compare=Sim2 brief;
var X1 E;
run;
```

The output from PROC COMPARE shows that the X1 and E variables are unchanged. For the PCG RNG, the original stream and the second stream are guaranteed not to overlap. Furthermore, generating numbers from the second stream does not change the first stream.

```
                                    The COMPARE Procedure
                                Comparison of WORK.SIM with WORK.SIM2
                                        (Method=EXACT)

NOTE: No unequal values were found. All values compared are
exactly equal.
```

See Also

Call Routines

- “CALL STREAMREWIND Routine” on page 358
- “CALL STREAMINIT Routine” on page 352

Routine

- “RAND Function” on page 1304

CALL STREAMINIT Routine

Specifies a random-number generator and seed value for generating random numbers.

Category: Random Number

Syntax

CALL STREAMINIT<(*RNG*)>;

CALL STREAMINIT <(*seed*)>;

CALL STREAMINIT <(*RNG,seed*)>;

Optional Arguments

RNG
is 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 Threefry 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" on page 20.

Restriction The HARDWARE RNG is available only on Intel processors that support the RdRand instruction: Ivy Bridge and later processors.

Tips All the streams in a DATA step use the *RNG* that is specified in the first call to the STREAMINIT subroutine. Subsequent calls to STREAMINIT 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.

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.

Tip All the streams in the DATA step are initialized by using the seed value in the first call to STREAMINIT. Subsequent calls to STREAMINIT are ignored.

Details

Basics

A sequence of random or pseudorandom values that are generated by an *RNG* is called a *stream*. The STREAMINIT subroutine uses a positive seed to initialize an *RNG*. The first call to the STREAMINIT subroutine initializes the *RNG*; subsequent calls do not initialize. The *RNG* remains in effect for the remainder of the DATA step or procedure.

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 routine with a positive seed value and specify any of the pseudorandom *RNG*s. If the STREAMINIT routine is called before the RAND function, the resulting stream of random numbers is reproducible. Every time you run the DATA step, 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 STREAMINIT with a missing 0 or negative *seed* argument.
- Call STREAMINIT without the *seed* argument.
- Call the RAND function without previously calling STREAMINIT.

STREAMINIT can be used as a function. If STREAMINIT initializes the stream or streams, it returns the value of the seed. If the stream or streams have already been initialized by calling STREAMINIT or RAND, STREAMINIT returns 0.

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 0, SAS generates a seed value as described in “Using Random-Number Functions and CALL Routines in the DATA Step” on page 20.

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: Generate Random Integers

The following DATA step shows how to randomly generate five random integers that range from 1 to 10. The STREAMINIT subroutine specifies a *seed* value. Because an RNG is not specified, the default RNG is used. The RAND function is called five times inside a DO loop. Each call returns a random uniformly distributed number in the interval (0,1). Multiplying by 10 and calling the CEILZ function results in a random integer in the range 1 to 10.

If you change the *RNG* or the *seed*, the random integers potentially change. Because a *seed* value is used, this program generates the same integers every time it runs. If you omit the *seed* value or specify the *seed* value 0, the program generates a new set of integers every time it runs.

```
data Integers;
   call streaminit( 4321 );
   do i = 1 to 5;
      n = ceilz(10*rand('uniform'));
      output;
   end;
run;

proc print noobs; run;
```

The preceding statements produce these results:

i	n
1	8
2	4
3	10
4	3
5	3

Example 2: Randomly Sorting Rows

The following DATA step shows how to randomly permute the male students in the SASHELP.CLASS data set. The STREAMINIT subroutine specifies an *RNG* and a *seed* value. Because of the implicit loop over the observations, the RAND function generates a random uniform value in the interval (0,1) for each observation. The SORT procedure then sorts the data by the random value, which results in a random permutation of the rows of the original data.

```
/* Random sort
   The RAND function generates a uniform random variable that
   is sorted by PROC SORT. */
data Class;
   set Sashelp.Class(where=(sex='M'));
   call streaminit( 'mt64', 27182818284590 );
```

```
   Random = rand( 'uniform' );
run;

/* sort data in random order */
proc sort data=Class; by random; run;

title 'Data Sorted in Random Order';
proc print;
run;
```

OBS	Name	Sex	Age	Height	Weight	Random
1	Philip	M	16	72.0	150.0	0.08683
2	Robert	M	12	64.8	128.0	0.66322
3	Henry	M	14	63.5	102.5	0.73285
4	William	M	15	66.5	112.0	0.75144
5	Jeffrey	M	13	62.5	84.0	0.75723
6	Alfred	M	14	69.0	112.5	0.78130
7	John	M	12	59.0	99.5	0.83785
8	Thomas	M	11	57.5	85.0	0.84470
9	Ronald	M	15	67.0	133.0	0.84870
10	James	M	12	57.3	83.0	0.95149

Example 3: Simulate Random Coin Tosses

The following DATA step shows how to simulate one million coin tosses. The STREAMINIT subroutine specifies an *RNG* and *seed*. The RAND function generates random binary values with equal probability, which are used to assign the value **`Head`** or **`Tail`** to the COIN variable. The FREQ procedure tabulates how many heads and tails were generated. You should expect approximately an equal number of heads and tails.

```
data CoinToss;
call streaminit( 'pcg', 186284 );
do Toss = 1 to 1000000;
   if rand('Bernoulli',0.5) then
      Coin = 'Head';
   else
      Coin = 'Tail';
   output;
end;
run;

/* Count heads and tails */
title 'Heads and Tails';
proc freq data=CoinToss;
   tables Coin / nocum;
```

```
run;
```

Coin	Frequency	Percent
Head	501132	50.11
Tail	498868	49.89

Example 4: Generate Uncorrelated Random Normal Variables

You can generate multiple variables that contain random numbers by using a single stream. The variables are statistically independent and uncorrelated, as shown in this example:

```
data Rand(drop=i);
call streaminit('ThreeFry2x64', 97531);
do i = 1 to 100000;
   x1 = rand('normal');
   x2 = rand('normal', 10, 2);
   x3 = rand('normal', 15, 3);
   output;
end;
run;

proc corr data=Rand noprob;
var x1-x3;
run;
```

3 Variables:	x1 x2 x3

Simple Statistics						
Variable	N	Mean	Std Dev	Sum	Minimum	Maximum
x1	100000	-0.00262	0.99923	-261.51120	-4.58527	4.55185
x2	100000	9.98468	1.99422	998468	1.20960	19.00989
x3	100000	15.01080	2.98425	1501080	1.63577	28.16925

Pearson Correlation Coefficients, N = 100000			
	x1	x2	x3
x1	1.00000	-0.00112	-0.00047
x2	-0.00112	1.00000	0.00146
x3	-0.00047	0.00146	1.00000

The DATA step generates 100,000 random values from uncorrelated normal variables. The CORR procedure computes simple descriptive statistics and the correlations between variables. The X1 variable is generated from a probability distribution that has mean 0 and unit standard deviation. The sample mean and

standard deviation are very close to those values. Similarly, the sample means and standard deviations for the X2 and X3 variables are close to their population values. The matrix of sample correlations is very close to the identity matrix. The off-diagonal elements, which indicate correlations between variables, are very small.

See Also

Functions:

- "RAND Function" on page 1304
- "CALL STREAM Routine" on page 349
- "CALL STREAMREWIND Routine" on page 358

CALL STREAMREWIND Routine

Rewinds a stream to its initial state for subsequent random-number generation.

Category: Random Number

Syntax

CALL STREAMREWIND;

CALL STREAMREWIND<(*key*)>;

Optional Argument

key
a numeric expression that specifies the key value for a stream of pseudorandom numbers.

Range Key values in CALL STREAMREWIND correspond to key values in CALL STREAM.

See "CALL STREAM Routine" on page 349

Details

CALL STREAMREWIND resets an RNG to its initial state.

For most applications, there is no need to use the STREAMREWIND subroutine to rewind a stream. The usual way to generate random numbers is to set a single seed

value and use a single stream, as explained in "Using Random-Number Functions and CALL Routines in the DATA Step" on page 20. Rewinding a stream can lead to random numbers that are correlated, so think carefully before calling the STREAMREWIND subroutine.

The STREAMREWIND subroutine rewinds a stream to its initial state. If you do not specify an argument, the CALL STREAMREWIND routine rewinds every stream. If you specify an argument, the stream identified by that key value is rewound, and that stream becomes the active stream; other streams are not affected. For more information about key values, see "CALL STREAM Routine" on page 349.

Hardware-based RNGs do not support streams. It is an error to call the STREAMINIT subroutine if you are using a hardware-based RNG.

Examples

Example 1: Reset a Random-Number Stream

This DATA step demonstrates how to call the STREAMREWIND subroutine. The STREAMINIT call initializes a Threefry RNG and initializes it. The RAND function then generates random observations from the uniform and normal distributions. The call to the STREAMREWIND subroutine re-initializes the RNG to its initial state. Consequently, the RAND function generates the same pseudorandom numbers when **iter=2** that were previously generated when **iter=1**.

```
data rewind;
call streaminit( 'tf2', 12345 );
do iter = 1 to 2;
   do i = 1 to 3;
      u = rand( 'uniform' );
      n = rand( 'normal' );
      output;
   end;
   call streamrewind;
end;

proc print data=rewind;
run;
```

Obs	iter	i	u	n
1	1	1	0.33476	-0.18806
2	1	2	0.29159	0.73548
3	1	3	0.26579	0.20907
4	2	1	0.33476	-0.18806
5	2	2	0.29159	0.73548
6	2	3	0.26579	0.20907

Example 2: Reset Multiple Random-Number Streams

This DATA step demonstrates how to use two streams and rewind one or both streams. The STREAMINIT call sets and initializes a Threefry RNG. The STREAM subroutine creates two streams: one with **key=0** and one with **key=1**. The RAND function generates three random observations from each stream.

After generating three observations, the program rewinds the stream for **key=0**. Consequently, the values of the U variable for observations 4–6 are the same as for observations 1–3.

After generating six observations, the program rewinds both streams. Consequently, the values of the U and N variables, for observations 7–9, are the same for observations 1–3.

```
data rewind2;
call streaminit( 'tf4', 54321 );
do iter = 1 to 3;
   do i = 1 to 3;
      call stream(0);
      u = rand( 'uniform' );
      call stream(1);
      n = rand( 'normal' );
      output;
   end;
   if iter = 1 then
      call streamrewind(0);
   else if iter = 2 then
      call streamrewind;
end;

proc print data=rewind2;
run;
```

Obs	iter	i	u	n
1	1	1	0.70032	-1.25566
2	1	2	0.16419	0.83939
3	1	3	0.83045	0.33226
4	2	1	0.70032	-0.30012
5	2	2	0.16419	0.35669
6	2	3	0.83045	0.94662
7	3	1	0.70032	-1.25566
8	3	2	0.16419	0.83939
9	3	3	0.83045	0.33226

See Also

Call Routines

- “CALL STREAM Routine” on page 349
- “CALL STREAMINIT Routine” on page 352

Functions

- “RAND Function” on page 1304

CALL SYMPUT Routine

Assigns a value produced in a DATA step to a macro variable.

Category: Macro

Type: DATA step call routine

See: “SYMGET Function” in *SAS Macro Language Reference for SAS Viya Workbench* and “CALL SYMPUTX Routine” on page 366

Syntax

CALL SYMPUT(*macro-variable*, *value*);

Required Arguments

macro-variable
can be one of the following items:

- a character string that is a SAS name, enclosed in quotation marks. For example, to assign the character string `testing` to macro variable NEW, submit the following statement:

```
call symput('new','testing');
```

- the name of a character variable whose values are SAS names. For example, this DATA step creates the three macro variables SHORTSTP, PITCHER, and FRSTBASE and respectively assigns them the values ANN, TOM, and BILL.

```
data team1;
   input position : $8. player : $12.;
   call symput(position,player);
datalines;
shortstp Ann
pitcher Tom
frstbase Bill
;
```

```
%put Short stop: &shortstp;
%put Pitcher:    &pitcher;
%put First Base: &frstbase;
```

- a character expression that produces a macro variable name. This form is useful for creating a series of macro variables. For example, the CALL SYMPUT statement builds a series of macro variable names by combining the character string POS and the left-aligned value of _N_ . Values are assigned to the macro variables POS1, POS2, and POS3.

```
data team2;
   input position : $12. player $12.;
   call symput('POS'||left(_n_), position);
   datalines;
shortstp Ann
pitcher Tom
frstbase Bill
;

%put Position 1: &pos1;
%put Position 2: &pos2;
%put Position 3: &pos3;
```

value

is the value to be assigned, which can be

- a string enclosed in quotation marks. For example, this statement assigns the string `testing` to the macro variable NEW:

```
call symput('new','testing');
```

- the name of a numeric or character variable. The current value of the variable is assigned as the value of the macro variable. If the variable is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log. Later sections on formatting rules describe the rules that SYMPUT follows in assigning character and numeric values of DATA step variables to macro variables.

  Note: This form is most useful when *macro-variable* is also the name of a SAS variable or a character expression that contains a SAS variable. A unique macro variable name and value can be created from each observation, as shown in the previous example for creating the data set Team1.

  If *macro-variable* is a character string, SYMPUT creates only one macro variable, and its value changes in each iteration of the program. Only the value assigned in the last iteration remains after program execution is finished.

- a DATA step expression. The value returned by the expression in the current observation is assigned as the value of *macro-variable*. In this example, the macro variable named HOLDATE receives the value `July 4,1997`:

```
data c;
   input holiday mmddyy.;
   call symput('holdate',trim(left(put(holiday,worddate.))));
```

```
datalines;
070497
;
run;
```

If the expression is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log. Later sections on formatting rules describe the rules that SYMPUT follows in assigning character and numeric values of expressions to macro variables.

Details

If *macro-variable* exists in any enclosing scope, *macro-variable* is updated. If *macro-variable* does not exist, SYMPUT creates it. (See below to determine in which scope SYMPUT creates *macro-variable*.) SYMPUT makes a macro variable assignment when the program executes.

SYMPUT can be used in all SAS language programs, including SCL programs. Because it resolves variables at program execution instead of macro execution, SYMPUT should be used to assign macro values from DATA step views, SQL views, and SCL programs.

Scope of Variables Created with SYMPUT

SYMPUT puts the macro variable in the most local symbol table that it finds. If there are no local tables, then the macro variable is put in the global symbol table.

For more information about creating a variable with SYMPUT, see "Scopes of Macro Variables" in *SAS Macro Language Reference for SAS Viya Workbench*.

Note: Macro references execute immediately and SAS statements do not execute until after a step boundary. You cannot use CALL EXECUTE to invoke a macro that contains references for macro variables that are created by CALL SYMPUT in that macro. For a workaround, see the following TIP:

TIP If CALL SYMPUT is contained within a macro and that macro is invoked within a CALL EXECUTE routine, the macro variable will not resolve inside the macro. To get around this issue, invoke the macro in one of these two ways:

```
call execute('%nrstr(%test('||temp||'))');
```

Use the %NRSTR macro quoting function to mask the macro statement.

```
rc=dosubl('%test('||temp||')');
```

Problem Trying to Reference a SYMPUT-Assigned Value Before It Is Available

One of the most common problems in using SYMPUT is trying to reference a macro variable value assigned by SYMPUT before that variable is created. The failure generally occurs because the statement referencing the macro variable compiles

before execution of the CALL SYMPUT statement that assigns the variable's value. The most important fact to remember in using SYMPUT is that it assigns the value of the macro variable during program execution. Macro variable references resolve during the compilation of a step, a global statement used outside a step, or an SCL program. As a result:

- You cannot use a macro variable reference to retrieve the value of a macro variable in the same program (or step) in which SYMPUT creates that macro variable and assigns it a value.
- You must specify a step boundary statement to force the DATA step to execute before referencing a value in a global statement following the program (for example, a TITLE statement). The boundary could be a RUN statement or another DATA or PROC statement. For example:

```
data x;
   x='December';
   call symput('var',x);
proc print;
title "Report for &var";
run;
```

Processing provides details about compilation and execution.

Formatting Rules For Assigning Character Values

If *value* is a character variable, SYMPUT writes it using the $*w*. format, where *w* is the length of the variable. Therefore, a value shorter than the length of the program variable is written with trailing blanks. For example, in the following DATA step the length of variable C is 8 by default. Therefore, SYMPUT uses the $8. format and assigns the letter x followed by seven trailing blanks as the value of CHAR1. To eliminate the blanks, use the TRIM function as shown in the second SYMPUT statement.

```
data char1;
   input c $;
   call symput('char1',c);
   call symput('char2',trim(c));
   datalines;
x
;
run;
%put char1 = ***&char1***;
%put char2 = ***&char2***;
```

When this program executes, these lines are written to the SAS log:

```
char1 = ***x       ***
char2 = ***x***
```

Formatting Rules For Assigning Numeric Values

If *value* is a numeric variable, SYMPUT writes it using the BEST12. format. The resulting value is a 12-byte string with the value right-aligned within it. For example, this DATA step assigns the value of numeric variable X to the macro variables NUM1 and NUM2. The last CALL SYMPUT statement deletes undesired leading blanks by using the LEFT function to left-align the value before the SYMPUT routine assigns the value to NUM2.

```
data _null_;
   x=1;
   call symput('num1',x);
   call symput('num2',left(x));
   call symput('num3',trim(left(put(x,8.)))); /*preferred technique*/
run;
%put num1 = ***&num1***;
%put num2 = ***&num2***;
%put num3 = ***&num3***;
```

When this program executes, these lines are written to the SAS log:

```
num1 = ***           1***
num2 = ***1           ***
num3 = ***1***
```

Comparisons

- SYMPUT assigns values produced in a DATA step to macro variables during program execution, but the SYMGET function returns values of macro variables to the program during program execution.
- SYMPUT is available in DATA step and SCL programs, but SYMPUTN is available only in SCL programs.
- SYMPUT assigns character values, but SYMPUTN assigns numeric values.

Example: Creating Macro Variables and Assigning Them Values from a Data Set

```
data dusty;
   input dept $ name $ salary @@;
   datalines;
bedding Watlee 18000    bedding Ives 16000
bedding Parker 9000     bedding George 8000
bedding Joiner 8000     carpet Keller 20000
carpet Ray 12000        carpet Jones 9000
gifts Johnston 8000     gifts Matthew 19000
kitchen White 8000      kitchen Banks 14000
kitchen Marks 9000      kitchen Cannon 15000
tv Jones 9000           tv Smith 8000
tv Rogers 15000         tv Morse 16000
;
proc means noprint;
   class dept;
   var salary;
   output out=stats sum=s_sal;
run;
data _null_;
   set stats;
   if _n_=1 then call symput('s_tot',trim(left(s_sal)));
```

```
   else call symput('s'||dept,trim(left(s_sal)));
run;
%put _user_;
```

When this program executes, this list of variables is written to the SAS log:

```
GLOBAL SCARPET 41000
GLOBAL SKITCHEN 46000
GLOBAL STV 48000
GLOBAL SGIFTS 27000
GLOBAL SBEDDING 59000
GLOBAL S_TOT 221000
```

CALL SYMPUTX Routine

Assigns a value to a macro variable, and removes both leading and trailing blanks.

Category: Macro

See: "CALL SYMPUTX Routine" in *SAS Functions and CALL Routines for SAS Viya Workbench*

Syntax

CALL SYMPUTX(*macro-variable*, *value* <, *symbol-table*>);

Required Arguments

macro-variable
can be one of the following items:

- a character string that is a SAS name, enclosed in quotation marks.
- the name of a character variable whose values are SAS names.
- a character expression that produces a macro variable name. This form is useful for creating a series of macro variables.
- a character constant, variable, or expression. Leading and trailing blanks are removed from the value of name, and the result is then used as the name of the macro variable.

value
is the value to be assigned, which can be

- a string enclosed in quotation marks.
- the name of a numeric or character variable. The current value of the variable is assigned as the value of the macro variable. If the variable is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log.

Note: This form is most useful when *macro-variable* is also the name of a SAS variable or a character expression that contains a SAS variable. A unique macro variable name and value can be created from each observation.

- a DATA step expression. The value returned by the expression in the current observation is assigned as the value of *macro-variable*.

  If the expression is numeric, SAS performs an automatic numeric-to-character conversion and writes a message in the log.

Optional Argument

symbol-table
specifies a character constant, variable, or expression. The value of *symbol-table* is not case sensitive. The first non-blank character in *symbol-table* specifies the symbol table in which to store the macro variable. The following values are valid as the first non-blank character in *symbol-table*:

G
specifies that the macro variable is stored in the global symbol table, even if a local symbol table exists.

L
specifies that the macro variable is stored in the most local symbol table that exists. If a local symbol table does not exist, which occurs when CALL SYMPUTX is used outside a macro, the macro variable is stored in the global symbol table.

F
specifies that if the macro variable exists in any symbol table, CALL SYMPUTX uses the version in the most local symbol table in which it exists. If the macro variable does not exist, CALL SYMPUTX stores the variable in the most local symbol table that it finds. If CALL SYMPUTX cannot find a local symbol table, it stores the macro variable in the global symbol table.

Note: This is the same method that the CALL SYMPUT routine uses.

Default F

Note If you specify a value other than G, L, or F, the default (F) is used.

Comparisons

CALL SYMPUTX is similar to CALL SYMPUT. Here are the differences.

- CALL SYMPUTX does not write a note to the SAS log when the second argument is numeric. CALL SYMPUT, however, writes a note to the log stating that numeric values were converted to character values.

- CALL SYMPUTX uses a field width of up to 32 characters when it converts a numeric second argument to a character value. CALL SYMPUT uses a field width of up to 12 characters.
- CALL SYMPUTX left-justifies both arguments and trims trailing blanks. CALL SYMPUT does not left-justify the arguments, and trims trailing blanks from the first argument only. Leading blanks in the value of name cause an error.
- CALL SYMPUTX enables you to specify the symbol table in which to store the macro variable, whereas CALL SYMPUT does not.

Example

The following example shows the results of using CALL SYMPUTX.

```
%let x=1;
%let items=2;
%macro test(val);

data _null_;
 call symputx('items', ' leading and trailing blanks removed ', 'L');
 call symputx('  x   ', 123.456);
run;

%put local items &items;
%mend test;

%test(100)

%put items=!&items!;
%put x=!&x!;
```

The following lines are written to the SAS log:

```
82   %let x=1;
83   %let items=2;
84   %macro test(val);
85
86   data _null_;
87    call symputx('items', ' leading and trailing blanks removed ', 'L');
88    call symputx('  x   ', 123.456);
89   run;
90
91   %put local items &items
92   %mend test;
93
94   %test(100)
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

local items leading and trailing blanks removed
95
96   %put items=!&items!;
items=!2!
97   %put x=!&x!;
x=!123.456!
```

CALL SYSTEM Routine

Submits an operating environment command for execution.

Category: Special

Interaction: When invoked by the %SYSCALL macro statement, CALL SYSTEM removes quotation marks from its arguments. For more information, see "Invoking CALL Routines and the %SYSCALL Macro Statement" on page 12.

Operating environment: For the Linux OS, *command* must evaluate to a valid Linux command

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL SYSTEM(*command*);

Required Argument

command
specifies any of the following commands: a system command that is enclosed in quotation marks (character string), an expression whose value is a system command, or the name of a character variable whose value is a system command that is executed.

Restriction The length of the command cannot be greater than 1024 characters, including trailing blanks.

Details

General Information

The behavior of the CALL SYSTEM routine is similar to that of the X command, the X statement, and the SYSTEM function. CALL SYSTEM is useful in certain situations because it can be conditionally executed, it accepts an expression as an argument, and it is executed at run time.

Information That Is Specific to Linux

Under Linux, the output of the command appears in the window from which you invoked SAS.

The value of the XSYNC system option affects how the CALL SYSTEM routine works.

Note: The CALL SYSTEM routine can be executed within a DATA step. However, neither the X statement nor the %SYSEXEC macro program statement is intended for use during the execution of a DATA step.

Example: Using the CALL SYSTEM Routine to Send a Message under Linux

In this example, for each record in `answer.week`, if the `resp` variable is `y`, the CALL SYSTEM routine mails a message.

```
data _null_;
   set answer.week;
   if resp='y' then
      do;
         call system('mail mgr < $HOME/msg');
      end;
   run;
```

See Also

Functions:

- "SYSTEM Function" on page 1446

CALL TANH Routine

Returns the hyperbolic tangent.

Category: Mathematical

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL TANH(*argument* <, *argument, ..., argument-n*>);

Required Argument

argument
is numeric.

Restriction The CALL TANH routine accepts variables only as valid arguments. Do not use a constant or a SAS expression because the CALL routine is unable to update these arguments.

Details

The subroutine TANH replaces each argument by the tanh of that argument. For example, x_j is replaced by the following equation:

$$tanh(x_j) = \frac{\varepsilon^{x_j} - \varepsilon^{-x_j}}{\varepsilon^{x_j} + \varepsilon^{-x_j}}$$

If any argument contains a missing value, CALL TANH returns missing values for all the arguments.

Example

```
data one;
   x=0.5;
   y=-0.5;
   call tanh(x, y);
   put x= y=;
run;
```

The preceding statements produce these results:

```
x=0.4621171573 y=-0.462117157
```

See Also

Functions:

- “TANH Function” on page 1448

CALL VNAME Routine

Assigns a variable name as the value of a specified variable.

Category: Variable Control

Note: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

Syntax

CALL VNAME(*variable-1*, *variable-2*);

Required Arguments

variable-1
specifies any SAS variable.

variable-2
specifies any SAS character variable. Because SAS variable names can contain up to 32 characters, the length of *variable-2* should be at least 32.

Details

The CALL VNAME routine assigns the name of the *variable-1* variable as the value of the *variable-2* variable.

Example: Using the CALL VNAME Routine

This example uses the CALL VNAME routine with array references to return the names of all variables in the data set Old.

```
data new(keep=name);
   set old;
      /* all character variables in old */
   array abc{*} _character_;
      /* all numeric variables in old */
   array def{*} _numeric_;
      /* name is not in either array */
   length name $32;
   do i=1 to dim(abc);
         /* get name of character variable */
      call vname(abc{i}, name);
         /* write name to an observation */
      output;
   end;
   do j=1 to dim(def);
         /* get name of numeric variable */
      call vname(def{j}, name);
         /* write name to an observation */
      output;
   end;
```

```
   stop;
run;
```

See Also

Functions:

- “VNAME Function” on page 1554
- “VNAMEX Function” on page 1556

CALL VNEXT Routine

Returns the name, type, and length of a variable that is used in a DATA step.

Category: Variable Information

Notes: Argument types for arguments that are updated must match in CALL routines. All argument types must be CHAR, VARCHAR, or NUMERIC. If the argument types do not match, a warning is issued to the log.

This function supports the VARCHAR type.

Syntax

CALL VNEXT(*variable-name* <, *variable-type* <, *variable-length*> >);

Required Argument

variable-name
is a character variable that is updated by the CALL VNEXT routine. The following conditions apply:

- If the input value of *variable-name* is blank, the value that is returned in *variable-name* is the name of the first variable in the DATA step's list of variables.
- If the CALL VNEXT routine is executed for the first time in the DATA step, the value that is returned in *variable-name* is the name of the first variable in the DATA step's list of variables.

If neither of these conditions exists, the input value of *variable-name* is ignored. Each time the CALL VNEXT routine is executed, the value that is returned in *variable-name* is the name of the next variable in the list.

After the names of all the variables in the DATA step are returned, the value that is returned in *variable-name* is blank.

Optional Arguments

variable-type
is a character variable whose input value is ignored. The value that is returned is "N" or "C". The following rules apply:

- If the value that is returned in *variable-name* is the name of a numeric variable, the value that is returned in *variable-type* is "N".
- If the value that is returned in *variable-name* is the name of a character variable, the value that is returned in *variable-type* is "C".
- If the value that is returned in *variable-name* is blank, the value that is returned in *variable-type* is also blank.

variable-length
is a numeric variable. The input value of *variable-length* is ignored.

The value that is returned is the length of the variable whose name is returned in *variable-name*. If the value that is returned in *variable-name* is blank, the value that is returned in *variable-length* is 0.

Details

The variable names that are returned by the CALL VNEXT routine include automatic variables such as _N_ and _ERROR_. If the DATA step contains a BY statement, the variable names that are returned by CALL VNEXT include the FIRST.*variable* and LAST.*variable* names. CALL VNEXT also returns the names of the variables that are used as arguments to CALL VNEXT.

Note: The order in which variable names are returned by CALL VNEXT can vary in different releases of SAS and in different operating environments.

Example: Using the CALL VNEXT Routine

This example shows the results from using the CALL VNEXT routine.

```
data test;
   x=1;
   y='abc';
   z=.;
   length z 5;
run;
data attributes;
   set test;
   by x;
   input a b $ c;
   length name $38 type $3;
   name=' ';
   length=666;
   do i=1 to 99 until(name=' ');
```

```
      call vnext(name, type, length);
      put i= name @40 type= length=;
   end;
   this_is_a_long_variable_name=0;
   datalines;
1 q 3
;
```

The following partial output is from the SAS log:

```
i=1 x                                type=N length=8
i=2 y                                type=C length=3
i=3 z                                type=N length=5
i=4 FIRST.x                          type=N length=8
i=5 LAST.x                           type=N length=8
i=6 a                                type=N length=8
i=7 b                                type=C length=8
i=8 c                                type=N length=8
i=9 name                             type=C length=38
i=10 type                            type=C length=3
i=11 length                          type=N length=8
i=12 i                               type=N length=8
i=13 this_is_a_long_variable_name    type=N length=8
i=14 _ERROR_                         type=N length=8
i=15 _N_                             type=N length=8
i=16                                 type=  length=0
```

CAT Function

Does not remove leading or trailing blanks and returns a concatenated character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: DBCS equivalent function is KSTRCAT in *SAS National Language Support (NLS): Reference Guide*.

Syntax

CAT(*item-1* <, ..., *item-n*>)

Required Argument

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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

In a DATA step, if the CAT function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes. If the concatenation operator (||) returns a value to a variable that has not previously been assigned a length, 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 one of these lengths:

- 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 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 performs these actions:

- 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

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 operator (||) 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 376.

Using the CAT, CATS, CATT, and CATX functions is faster than using the TRIM and LEFT functions, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter such as a blank or comma.

Function	Equivalent Code
CAT(OF X1-X4)	X1||X2||X3||X4
CATS(OF X1-X4)	TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))
CATT(OF X1-X4)	TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4)
CATX(SP, OF X1-X4)	TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4))

Example

This example shows how the CAT function concatenates strings.

```
data _null_;
   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 $char.;
run;
```

SAS writes the following results to the log:

```
The 2012 Olympic Arts Festi val included works by D ale Chihuly.
```

See Also

Functions:

- “CATQ Function” on page 378
- “CATS Function” on page 383
- “CATT Function” on page 385
- “CATX Function” on page 388

CALL Routines:

- "CALL CATS Routine" on page 225
- "CALL CATT Routine" on page 228
- "CALL CATX Routine" on page 230

CATQ Function

Concatenates character and 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 assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

CATQ(*modifiers* <, *delimiter*>, *item-1* <, ..., *item-n*>)

Required 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, CATQ does not insert delimiters for item arguments that have a length of 0 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, the value of that variable is not changed by CATQ unless the result is assigned to that variable.

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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.

Optional Argument

delimiter
specifies a character constant, variable, or expression that is used as a delimiter between concatenated strings. If you specify this argument, you must also specify the D modifier.

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 one of these lengths:

- 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, SAS performs these actions:

- 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, 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, CATQ uses a blank as a delimiter.

If you do not specify a quotation mark in *modifier* or the 1 or 2 modifiers, 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 0, and then performs the following actions for each item argument:

1 If *item* is not a character string, CATQ converts *item* to a character string by using the BEST*w*. format and removes leading blanks.

2 If you used the S modifier, CATQ removes leading blanks from the string.

3 If you used the S or T modifiers, CATQ removes trailing blanks from the string.

4 CATQ determines whether to add quotation marks based on these 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, the string is converted to a name literal if either of these 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 N modifiers, CATQ adds quotation marks to the string if any of these conditions are 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 these conditions is true:

- You used the M modifier.
- The string has a length greater than 0 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

This example shows how the CATQ function concatenates strings.

```
options ls=110;
data _null_;
```

```
    result1=CATQ(' ',
                 'noblanks',
                 'one blank',
                 12345,
                 '  lots of blanks    ');
    result2=CATQ('CS',
                 'Period (.)                ',
                 'Ampersand (&)             ',
                 'Comma (,)                 ',
                 'Double quotation marks (") ',
                 '   Leading Blanks');
    result3=CATQ('BCQT',
                 'Period (.)                ',
                 'Ampersand (&)             ',
                 'Comma (,)                 ',
                 'Double quotation marks (") ',
                 '   Leading Blanks');
    result4=CATQ('ADT',
                 '#=#',
                 'Period (.)                ',
                 'Ampersand (&)             ',
                 'Comma (,)                 ',
                 'Double quotation marks (") ',
                 '   Leading Blanks');
    result5=CATQ('N',
                 'ABC_123   ',
                 '123    ',
                 'ABC 123');
    put (result1-result5) (=/);
run;
```

SAS writes the following results 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 375
- “CATS Function” on page 383
- “CATT Function” on page 385
- “CATX Function” on page 388

CALL Routines:

- "CALL CATS Routine" on page 225
- "CALL CATT Routine" on page 228
- "CALL CATX Routine" on page 230

CATS Function

Removes leading and trailing blanks, and returns a concatenated character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

CATS(*item-1* <, ..., *item-n*>)

Required Argument

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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.

Details

Length of Returned Variable

In a DATA step, if the CATS function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes. If the concatenation operator (||) returns a value to a variable that has not previously been assigned a length, 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 one of these lengths:

- 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 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 performs these actions:

- 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

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 operator (||) 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 383.

Using the CAT, CATS, CATT, and CATX functions is faster than using the TRIM and LEFT functions, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter such as a blank or comma.

Function	Equivalent Code
`CAT(OF X1-X4)`	`X1||X2||X3||X4`
`CATS(OF X1-X4)`	`TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))`
`CATT(OF X1-X4)`	`TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4)`
`CATX(SP, OF X1-X4)`	`TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4))`

Example

This example shows how the CATS function concatenates strings.

```
data _null_;
   x='  The   Olym';
   y='pic Arts Festi';
   z='  val includes works by D  ';
   a='ale Chihuly.';
   result=cats(x, y, z, a);
   put result $char.;
run;
```

SAS writes the following results to the log:

```
The   Olympic Arts Festival includes works by Dale Chihuly.
```

See Also

Functions:

- “CAT Function” on page 375
- “CATQ Function” on page 378
- “CATT Function” on page 385
- “CATX Function” on page 388

CALL Routines:

- “CALL CATS Routine” on page 225
- “CALL CATT Routine” on page 228
- “CALL CATX Routine” on page 230

CATT Function

Removes trailing blanks, and returns a concatenated character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

CATT(*item-1* <, ... *item-n*>)

Required Argument

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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

In a DATA step, if the CATT function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes. If the concatenation operator (||) returns a value to a variable that has not previously been assigned a length, 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 one of these lengths:

- 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 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 performs these actions:

- 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

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 operator (||) 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 386.

Using the CAT, CATS, CATT, and CATX functions is faster than using the TRIM and LEFT functions, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter such as a blank or comma.

Function	Equivalent Code
CAT(OF X1-X4)	X1||X2||X3||X4
CATS(OF X1-X4)	STRIP(X1)||STRIP(X2)||STRIP(X3)||STRIP(X4)
CATT(OF X1-X4)	TRIMN(X1)||TRIMN(X2)||TRIMN(X3)||TRIMN(X4)
CATX(SP, OF X1-X4)	STRIP(X1)||SP||STRIP(X2)||SP||STRIP(X3)||SP||STRIP(X4)

Example

This example shows how the CATT function concatenates strings.

```
data _null_;
   x='  The   Olym';
   y='pic Arts Festi';
   z='  val includes works by D  ';
   a='ale Chihuly.';
   result=catt(x, y, z, a);
   put result $char.;
run;
```

SAS writes the following results to the log:

```
The   Olympic Arts Festi  val includes works by Dale Chihuly.
```

See Also

Functions:

- "CAT Function" on page 375

- "CATQ Function" on page 378
- "CATS Function" on page 383
- "CATX Function" on page 388

CALL Routines:

- "CALL CATS Routine" on page 225
- "CALL CATT Routine" on page 228
- "CALL CATX Routine" on page 230

CATX Function

Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

CATX(*delimiter*, *item-1* <, ... *item-n*>)

Required 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, 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 388.

Details

The Basics

The CATX function first copies *item-1* to the result, omitting leading and trailing blanks. Next, for each subsequent argument *item-i*, i=2, ..., n, if *item-i* contains at least one non-blank character, 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

In a DATA step, if the CATX function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes. If the concatenation operator (||) returns a value to a variable that has not previously been assigned a length, 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 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 has one of these lengths:

- 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 CATX is called from the macro processor

If CATX 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 CATX 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 performs these actions:

- 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

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 operator (||) 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 389.

Using the CAT, CATS, CATT, and CATX functions is faster than using the TRIM and LEFT functions, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

Note: In the case of variables that have missing values, the concatenation produces different results. See "Example 2: Concatenating Strings That Have Missing Values" on page 390.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter such as a blank or comma.

Function	Equivalent Code
CAT(OF X1-X4)	X1||X2||X3||X4
CATS(OF X1-X4)	TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))
CATT(OF X1-X4)	TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4)
CATX(SP, OF X1-X4)	TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4))

Examples

Example 1: Concatenating Strings That Have No Missing Values

This example shows how the CATX function concatenates strings that have no missing values.

```
data _null_;
   separator='%%$%%';
   x='The Olympic  ';
   y='   Arts Festival ';
   z='   includes works by ';
   a='Dale Chihuly.';
   result=catx(separator, x, y, z, a);
   put result $char.;
run;
```

SAS writes the following result to the log:

```
The Olympic%%$%%Arts Festival%%$%%includes works by%%$%%Dale Chihuly.
```

Example 2: Concatenating Strings That Have Missing Values

This example shows how the CATX function concatenates strings that contain missing values.

```
data one;
   length x1-x4 $1;
   input x1-x4;
   datalines;
A B C D
```

```
E . F G
H . . J
;
run;
data two;
   set one;
   SP='^';
   test1=catx(sp, of x1-x4);
   test2=trim(left(x1)) || sp || trim(left(x2)) || sp ||
trim(left(x3)) || sp ||
      trim(left(x4));
run;

proc print data=two;
run;
```

Output 3.19 *Using CATX with Missing Values*

The SAS System

Obs	x1	x2	x3	x4	SP	test1	test2
1	A	B	C	D	^	A^B^C^D	A^B^C^D
2	E		F	G	^	E^F^G	E^ ^F^G
3	H			J	^	H^J	H^ ^ ^J

See Also

Functions:

- “CAT Function” on page 375
- “CATQ Function” on page 378
- “CATS Function” on page 383
- “CATT Function” on page 385

CALL Routines:

- “CALL CATS Routine” on page 225
- “CALL CATT Routine” on page 228
- “CALL CATX Routine” on page 230

CDF Function

Returns a value from a cumulative probability distribution.

Category: 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 1293.

Syntax

CDF(*distribution*, *quantile* <, *parameter-1,–parameter-k*>)

Required Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Valid distributions are as follows:

Table 3.2 *Distributions for the CDF Function*

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

Distribution	Argument
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.

Optional Argument

parameter
are optional 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 394 for complete information about these parameters.

Details

The CDF function computes the left cumulative distribution function from various continuous and discrete probability distributions.

See the Table 3.10 on page 392 for links to the CDF Function's distributions.

See Also

Functions:

- "LOGCDF Function" on page 1081
- "LOGPDF Function" on page 1084
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "QUANTILE Function" on page 1293
- "SDF Function" on page 1376
- "SQUANTILE Function" on page 1404

CDF Bernoulli Distribution Function

Returns a value from the Bernoulli cumulative probability distribution.

Category: Probability

Syntax

CDF('BERNOULLI', *x*, *p*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

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

```
data one;
   y=cdf('BERN', 0, .25);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.75
```

CDF Beta Distribution Function

Returns a value from the beta cumulative probability distribution.

Category: Probability

Syntax

CDF('BETA', *x*, *a*, *b* *v*<, *l*, *r*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range *a* > 0

b

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $b > 0$

v

is a numeric constant, variable, or expression that specifies a dispersion parameter

Range If *l* and *r* are specified, *l* <=*v*<=*r*; otherwise, 0<=*v*<=1.

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range $r > l$

Details

The CDF function of 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

```
data one;
   y=cdf('BETA', 0.2, 3, 4);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.09888
```

CDF Binomial Distribution Function

Returns a value from the binomial cumulative probability distribution.

Category: Probability

Syntax

CDF('BINOMIAL', *m*, *p*, *n*)

Arguments

m

is an integer random variable that counts the number of successes.

Range $m = 0, 1, \ldots$

p

is a numeric constant, variable, or expression that specifies a probability of success parameter.

Range $0 \le p \le 1$

n

is a numeric constant, variable, or expression that specifies an integer parameter that counts the number of independent Bernoulli trials.

Range $n = 0, 1, \ldots$

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

```
data one;
   y=cdf('BINOM', 4, .5, 10);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.37695
```

CDF Cauchy Distribution Function

Returns a value from the Cauchy cumulative probability distribution.

Category: Probability

Syntax

CDF('CAUCHY', *x* <,*θ*> <, *λ*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

θ
is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

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

```
data one;
   y=cdf('CAUCHY', 2);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.85242
```

CDF Chi Square Distribution Function

Returns a value from the Chi-square cumulative probability distribution.

Category: Probability

Syntax

CDF('CHISQUARE', *x*, *df* <, *nc*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

df
is a numeric constant, variable, or expression that specifies a degrees of freedom parameter.

Range *df* > 0

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range *nc* ≥ 0

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 noninteger degrees of freedom. If *nc* is omitted or equal to 0, 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

```
data one;
   y=cdf('CHISQ', 11.264, 11);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.57858
```

CDF Conway-Maxwell-Poisson Distribution Function

Returns a value from the Conway-Maxwell-Poisson cumulative probability distribution.

Category: Probability

Syntax

CDF('CONMAXPOI', *y*, *λ*,*v*)

Arguments

y
is a numeric constant, variable, or expression that specifies a nonnegative integer that represents counts data.

λ
is similar to the mean, as in the Poisson distribution.

v
is a numeric constant, variable, or expression that specifies a dispersion parameter.

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 1198.

Example

```
data one;
   y=cdf('CONMAXPOI', 5, 2.3, .4);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.2445411535
```

CDF Exponential Distribution Function

Returns a value from the Exponential probability distribution.

Category: Probability

Syntax

CDF('EXPONENTIAL', *x* <,*λ*>)

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 scale parameter.

Default 1

Range λ > 0

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

```
data one;
   y=cdf('EXPO', 1);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.63212
```

CDF F Distribution Function

Returns a value from the F distribution.

Category: Probability

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

ddf
is a numeric constant, variable, or expression that specifies a denominator degrees of freedom parameter.

Range *ddf* > 0

nc
is a numeric constant, variable, or expression that specifies a noncentrality parameter.

Range *nc* ≥ 0

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 0, 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:

$$CDF('F', x, v_1, v_2, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} P_F(x, v_1 + 2j, v_2) & x \geq 0 \end{cases}$$

In the equation, $P_f(f,u_1,u_2)$ is the probability from the central *F* distribution.

$$P_F(x, u_1, u_2) = P_B\left(\frac{u_1 x}{u_1 x + u_2}, \frac{u_1}{2}, \frac{u_2}{2}\right)$$

$P_B(x,a,b)$ is the probability from the standard beta distribution.

Note: There are no *location* or *scale* parameters for the *F* distribution.

Example

```
data one;
   y=cdf('F', 3.32, 2, 3);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.82639
```

CDF Gamma Distribution Function

Returns a value from the gamma distribution.

Syntax

CDF('GAMMA', *x*, *a* <, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

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 > 0

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

```
data one;
   y=cdf('GAMMA', 1, 3);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.080301
```

CDF Generalized Poisson Distribution Function

Returns information about the generalized Poisson distribution.

Syntax

CDF('GENPOISSON', *x*, *θ*, *η*)

Arguments

x
is a numeric constant, variable, or expression that specifies an integer random variable.

θ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≤5 and >0

η

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≥0 and <0.95

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, the generalized Poisson distribution becomes the standard Poisson distribution with the shape parameter θ.

Example

```
data one;
   y=cdf('GENPOISSON', 9, 1, .7);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.906162963
```

CDF Geometric Distribution Function

Returns information from the geometric distribution.

Syntax

CDF('GEOMETRIC', *m*, *p*)

Arguments

m

is a numeric random variable that specifies the number of failures.

Note 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 $0 \le p \le 1$

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

```
data one;
   y=cdf('GEOM',2,1);
   put y=;
run;
```

The preceding statements produce this result:

```
y=1
```

CDF Hypergeometric Distribution Function

Returns information about the hypergeometric distribution.

Syntax

CDF('HYPER', *x*, *N*, *R*, *n* <, *o*>)

Arguments

x

is a numeric constant, variable, or expression that specifies an integer random variable.

N

is a numeric constant, variable, or expression that specifies an integer population size parameter.

Range *N*=1, 2, ...

R

is a numeric constant, variable, or expression that specifies an integer number of items in the category of interest.

Range *R* = 0, 1, ..., *N*

n

is a numeric constant, variable, or expression that specifies an integer sample size parameter.

Range *n* = 1, 2, ..., *N*

o

is a numeric constant, variable, or expression that specifies an optional numeric odds ratio parameter.

Range *o* > 0

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

```
data one;
   y=cdf('HYPER', 2, 200, 50, 10);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.52367
```

CDF Laplace Distribution Function

Returns information about the Laplace distribution.

Syntax

CDF('LAPLACE', *x* <,*θ*, *λ*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

θ
is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

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

```
data one;
   y=cdf('LAPLACE', 1);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.81606
```

CDF Logistic Distribution Function

Returns information about the logistic distribution.

Syntax

CDF('LOGISTIC', *x* <, *θ*, *λ*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

θ
is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

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

```
data one;
   y=cdf('LOGISTIC', 1);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.73106
```

CDF Lognormal Distribution Function

Returns information about the lognormal distribution.

Syntax

CDF('LOGNORMAL', *x* <, *θ*, *λ*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

θ
is a numeric constant, variable, or expression that specifies a log scale parameter. e(θ) is a scale parameter.

Default 0

λ
is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

Range λ > 0

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

```
data one;
   y=cdf('LOGNORMAL', 1);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.5
```

CDF Negative Binomial Distribution Function

Returns information about the negative binomial distribution.

Syntax

CDF('NEGBINOMIAL', *m*, *p*, *n*)

Argument

m

is a numeric constant, variable, or expression that specifies a positive integer random variable that counts the number of failures.

Range *m* = 0, 1, ...

p
is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

n
is a numeric constant, variable, or expression that specifies a value that counts the number of successes.

Range $n > 0$

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

```
data one;
   y=cdf('NEGB', 1, .5, 2);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.5
```

CDF Normal Distribution Function

Returns information about the normal distribution.

Syntax

CDF('NORMAL', *x* <, *θ*, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

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

```
data one;
   y=cdf('NORMAL', 1.96);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.97500
```

CDF Normal Mixture Distribution Function

Returns information about the normal mixture distribution.

Syntax

CDF('NORMALMIX', *x*, *n*, *p*, *m*, *s*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

n

is a numeric constant, variable, or expression that specifies the number of mixtures.

Range *n* = 1, 2, ...

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, ...

m

is a numeric constant, variable, or expression that specifies the *n* means $m_1, m_2, \ldots, m_n$.

s

is a numeric constant, variable, or expression that specifies the *n* standard deviations $s_1, s_2, \ldots, s_n$.

Range *s* > 0

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, 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

```
data one;
   y=cdf('NORMALMIX',2.3, 3,.33, .33, .34,
       .5, 1.5, 2.5, .79, 1.6, 4.3);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.7181
```

CDF Pareto Distribution Function

Returns information about the Pareto distribution.

Syntax

CDF('PARETO', *x*, *a* <, *k*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

k

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $k > 0$

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

```
data one;
   y=cdf('PARETO', 1 ,1);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0
```

CDF Poisson Distribution Function

Returns information from the Poisson distribution.

Syntax

CDF('POISSON', *n*, *m*)

Arguments

n

is a numeric constant, variable, or expression that specifies an integer random variable.

Range *n* = 0, 1, ...

m

is a numeric constant, variable, or expression that specifies a mean parameter.

Range *m* > 0

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

```
data one;
   y=cdf('POISSON', 2, 1);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.91970
```

CDF T Distribution Function

Returns information about the CDF t distribution.

Syntax

CDF('T', *t*, *df* <, *nc*>)

Arguments

t
is a numeric constant, variable, or expression that specifies a random variable.

df
is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Details

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 0, 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

```
data one;
   y=cdf('T', .9, 5);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.79531
```

CDF Tweedie Distribution Function

Returns information about the Tweedie distribution.

Syntax

CDF ('TWEEDIE', *y*, *p* <, *μ*, *ϕ*>)

Arguments

y

is a numeric constant, variable, or expression that specifies a random variable.

Range $y \geq 0$

Notes This argument is required.

When $p>1$, y is numeric. When $p=1$, y is an integer.

p

is a numeric constant, variable, or expression that specifies the power parameter.

Range $p \geq 1$

Note This argument is required.

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu>0$

ϕ

is a numeric constant, variable, or expression that specifies the dispersion parameter.

Default 1

Range $\phi>0$

Details

The CDF function for the Tweedie distribution returns an exponential dispersion model with variance and mean related by the equation *variance*$= \phi \times \mu^p$.

$$\int_0^y \frac{1}{y} \sum_{j=1}^{\infty} \left(\frac{y^{-j\alpha}(p-1)^{j\alpha}}{\phi^{j}(1-\alpha)(2-p)^{j} j!\,\Gamma(-j\alpha)} \right) e\left(\frac{1}{\phi} \left(y \frac{\mu^{1-p}-1}{1-p} - \frac{\mu^{2-p}-1}{2-p} \right) \right) dy$$

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

```
data one;
   y=cdf('TWEEDIE', .8, 5);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.5917629164
```

CDF Uniform Distribution Function

Returns information about the uniform distribution.

Syntax

CDF('UNIFORM', *x* <, *l*, *r*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

l
is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

r
is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range $r > l$

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

```
data one;
   y=cdf('UNIFORM', 0.25);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.25
```

CDF Wald Distribution Function

Returns information about the Wald distribution.

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.

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 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

```
data one;
   y=cdf('WALD', 1, 2);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.62770
```

CDF Weibull Distribution Function

Returns information about the Weibull distribution.

Syntax

CDF('WEIBULL', *x*, *a* <, *λ*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

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 > 0

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

```
data one;
   y=cdf('WEIBULL', 1, 2);
   put y=;
run;
```

The preceding statements produce this result:

```
y=0.63212
```

CEIL Function

Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results.

Category: Rounding and Truncation

Syntax

CEIL(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

If the argument is within 1E-12 of an integer, the function returns that integer.

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, you might get unexpected results with the CEILZ function.

Example

```
data one;
   var1=2.1;
   a=ceil(var1);
   put a=;
   b=ceil(-2.4);;
   put b=;
   c=ceil(1+1.e-11);
   put c=;
   d=ceil(-1+1.e-11);
   put d=;
   e=ceil(1+1.e-13);
   put e=;
   f=ceil(223.456);
   put f=;
   g=ceil(763);
   put g=;
   h=ceil(-223.456);
   put h=;
run;
```

The preceding statements produce these results:

```
a=3
b=-2
c=2
d=0
e=1
f=224
g=763
h=-223
```

See Also

Functions:

■ "CEILZ Function" on page 426

CEILZ Function

Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.

Category: Rounding and Truncation

Syntax

CEILZ(*argument*)

Required Argument

argument
is a numeric constant, variable, or expression.

Details

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, you might get unexpected results with the CEILZ function.

Example

```
data one;
   a=ceilz(2.1);
   put a=;
   b=ceilz(-2.4);
   put b=;
   c=ceilz(1+1.e-11);
   put c=;
   d=ceilz(-1+1.e-11);
   put d=;
   e=ceilz(1+1.e-13);
   put e=;
   f=ceilz(223.456);
   put f=;
   g=ceilz(763);
   put g=;
   h=ceilz(-223.456);
   put h=;
run;
```

The preceding statements produce these results:

```
a=3
b=-2
c=2
d=0
e=2
f=224
g=763
h=-223
```

See Also

Functions:

- “CEIL Function” on page 424
- “FLOOR Function” on page 710
- “FLOORZ Function” on page 712
- “INT Function” on page 963
- “INTZ Function” on page 1020
- “ROUND Function” on page 1348
- “ROUNDE Function” on page 1356
- “ROUNDZ Function” on page 1359

CEXIST Function

Verifies the existence of a SAS catalog or SAS catalog entry.

Category: SAS File I/O

Syntax

CEXIST(*entry*<, 'U'>)

Required Argument

entry
is a character constant, variable, or expression that specifies a SAS catalog, or the name of an entry in a catalog. If the *entry* value is a one- or two-level name, then it is assumed to be the name of a catalog. Use a three- or four-level name to test for the existence of an entry within a catalog.

Optional Argument

'U'
tests whether the catalog can be opened for updating.

Details

CEXIST returns 1 if the SAS catalog or catalog entry exists, or 0 if the SAS catalog or catalog entry does not exist.

CEXIST(*entry*) verifies whether the catalog exists and can be read. For example, `cexist(myref.mycal)`. CEXIST(*entry, U*) verifies whether the catalog exists and can be updated. For example, `cexist(myref.mtcat,'U')`.

Comparisons

The EXIST function verifies the existence of a data set, view, catalog, and access file. The CEXIST function verifies the existence of a catalog with the added functionality of verifying whether the catalog can be updated.

Examples

Example 1: Verifying the Existence of an Entry in a Catalog

This example verifies the existence of the entry X.PROGRAM in LIB.CAT1:

```
data _null_;
   if cexist("lib.cat1.x.program") then
   put "Entry X.PROGRAM exists";
run;
```

Example 2: Determining If a Catalog Can Be Opened for Update

This example tests whether the catalog LIB.CAT1 exists and can be opened for update. If the catalog does not exist, a message is written to the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro test;
%if %sysfunc(cexist(lib.cat1, u)) %then
%put The catalog LIB.CAT1 exists and can be opened for update.;
%else
%put %sysfunc(sysmsg());
%mend;
%test
```

See Also

Functions:

- "EXIST Function" on page 582

CHAR Function

Returns a single character from a specified position in a character string.

Category: Character

Restriction: The internationalization level is 2.

Syntax

CHAR(*string*, *position*)

Required Arguments

string
specifies a character constant, variable, or expression.

position
is an integer that specifies the position of the character to be returned.

Details

In a DATA step, the default length of the target variable for the CHAR function is 1 character.

If *position* has a missing value, then CHAR returns a string with a length of 0. Otherwise, CHAR returns a string with a length of 1.

If *position* is less than or equal to 0, or greater than the length of the string, then CHAR returns a blank. Otherwise, CHAR returns the character at the specified position in the string.

Comparisons

The CHAR function returns the same result as SUBPAD(*string*, *position*, 1). The results are the same, but the default length of the target variable is different.

Example

The following example shows the results of using the CHAR function.

```
data test;
   retain string "abc";
   do position = -1 to 4;
      result=char(string, position);
      output;
   end;
run;

proc print noobs data=test;
run;
```

Output 3.20 *Output from the CHAR Function*

The SAS System

string	position	result
abc	-1	
abc	0	
abc	1	a
abc	2	b
abc	3	c
abc	4	

See Also

Functions:

- “FIRST Function” on page 709

CHOOSEC Function

Returns a character value that represents the results of choosing from a list of arguments.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

CHOOSEC(*index-expression*, *selection-1* <, ...*selection-n*>)

Required Arguments

index-expression
specifies a numeric constant, variable, or expression.

selection
specifies a character constant, variable, or expression. The value of this argument is returned by the CHOOSEC function.

Details

Length of Returned Variable

In a DATA step, if the CHOOSEC function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes.

The Basics

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 whereas CHOOSEN returns a numeric value.

Example

This example shows how CHOOSEC chooses from a series of values.

```
data _null_;
   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=;
run;
```

The preceding statements produce these results:

```
Fruit=apple Color=green Planet=Mercury Sport=baseball
```

See Also

Functions:

- "CHOOSEN Function" on page 432

CHOOSEN Function

Returns a numeric value that represents the results of choosing from a list of arguments.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

CHOOSEN(*index-expression*, *selection-1* <, ...*selection-n*>)

Required Arguments

index-expression
specifies a numeric constant, variable, or expression.

selection
specifies a numeric constant, variable, or expression. The value of this argument is returned by the CHOOSEN function.

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 CHOOSEN returns a numeric value whereas CHOOSEC returns a character value.

Example

This example shows how CHOOSEN chooses from a series of values.

```
data _null_;
   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= Rank= Score= Value=;
run;
```

The preceding statements produce these results:

```
ItemNumber=679 Rank=4 Score=33 Value=1014
```

See Also

Functions:

- “CHOOSEC Function” on page 430

CINV Function

Returns a quantile from the chi-square distribution.

Category: Quantile

Syntax

CINV(*p*, *df*<, *nc*>)

Required Arguments

p
is a numeric probability.

Range $0 \le p < 1$

df
is a numeric degrees of freedom parameter.

Range *df* > 0

Optional Argument

nc
is a numeric noncentrality parameter.

Range *nc* ≥ 0

Details

The CINV function returns the *p*th quantile from the chi-square distribution with degrees of freedom *df* and a noncentrality parameter *nc*. The probability that an observation from a chi-square distribution is less than or equal to the returned quantile is *p*. This function accepts a noninteger degrees of freedom parameter *df*.

If the optional parameter *nc* is not specified or has the value 0, the quantile from the central chi-square distribution is returned. The noncentrality parameter *nc* is defined such that if X is a normal random variable with mean μ and variance 1, X^2 has a noncentral chi-square distribution with *df*=1 and *nc* = μ^2.

CAUTION
For large values of nc, the algorithm could fail. In that case, a missing value is returned.

Note: CINV is the inverse of the PROBCHI function.

Example

The first statement that follows shows how to find the 95th percentile from a central chi-square distribution with 3 degrees of freedom. The second statement shows how to find the 95th percentile from a noncentral chi-square distribution with 3.5 degrees of freedom and a noncentrality parameter equal to 4.5.

```
data one;
   q1=cinv(.95, 3);
   a2=cinv(.95, 3.5, 4.5);
   put q1=;
   put a2=;
run;
```

The preceding statements produce these results:

```
q1=7.8147279033
a2=17.504582117
```

See Also

Functions:

- "QUANTILE Function" on page 1293

CLOSE Function

Closes a SAS data set.

Syntax

CLOSE(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

Details

CLOSE returns zero if the operation was successful, or returns a nonzero value if it was not successful. Close all SAS data sets as soon as they are no longer needed by the application.

Note: All data sets opened within a DATA step are closed automatically at the end of the DATA step.

Example

This example uses OPEN to open the SAS data set PAYROLL. If the data set opens successfully, indicated by a positive value for the variable PAYID, the example uses CLOSE to close the data set.

```
%macro test;
```

```
%let payid=%sysfunc(open(payroll, is));
/* macro statements */
%if &payid > 0 %then
%let rc=%sysfunc(close(&payid));
%mend;
%test
```

See Also

Functions:

- "OPEN Function" on page 1183

CMISS Function

Counts the number of missing arguments.

Categories: Descriptive Statistics
Missing Values

Note: This function supports the VARCHAR type.

Syntax

CMISS(*argument-1* <, *argument-2*,..., *argument-n*>)

Required Argument

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,.));
```

Comparisons

The CMISS function does not convert any argument. The NMISS function converts all arguments to numeric values.

See Also

Functions:

- “MISSING Function” on page 1108
- “NMISS Function” on page 1139

CNONCT Function

Returns the noncentrality parameter from a chi-square distribution.

Category: Mathematical

Syntax

CNONCT(*x*, *df*, *probability*)

Required Arguments

x
is a numeric random variable.

Range $x \geq 0$

df
is a numeric degrees of freedom parameter.

Range $df > 0$

probability
is a probability.

Range 0 < *probability* < 1

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 following 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:

$$P_g(x|a)$$

This relationship is the probability from the gamma distribution given by this equation:

$$P_g(x|a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} \varepsilon^{-t} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

```
data work;
   x=2;
   df=4;
   do nc=1 to 3 by .5;
      probability=probchi(x, df, nc);
      ncc=cnonct(x, df, probability);
      output;
   end;
run;
proc print;
run;
```

Output 3.21 *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

COALESCE Function

Returns the first nonmissing value from a list of numeric arguments.

Categories: Mathematical
Missing Values

Syntax

COALESCE(*argument-1*<..., *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The Basics

COALESCE accepts one or more numeric arguments. The COALESCE function checks the value of each argument in the order in which the values are listed and returns the first nonmissing value. If only one value is listed, the COALESCE function returns the value of that argument. If all the values of all the arguments are missing, the COALESCE function returns a missing value.

Comparisons

The COALESCE function searches numeric arguments, whereas the COALESCEC function searches character arguments.

Example

```
data one;
   x = COALESCE(42, .);
   y = COALESCE(.A, .B, .C);
   z = COALESCE(., 7, ., ., 42);
   put x=;
   put y=;
   put z=;
run;
```

The preceding statements produce these results:

```
x=42
y=.
z=7
```

See Also

Functions:

- "COALESCEC Function" on page 440

COALESCEC Function

Returns the first nonmissing value from a list of character arguments.

Categories: Character
Missing Values

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

COALESCEC(*argument-1* <..., *argument-n*>)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the COALESCEC function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes.

The Basics

COALESCEC accepts one or more character arguments. The COALESCEC function checks the value of each argument in the order in which the values are listed and returns the first nonmissing value. If only one value is listed, the COALESCEC function returns the value of that argument. A character value is considered missing if it has a length of zero or if all the characters are blank. If all the values of all the arguments are missing, the COALESCEC function returns a string with a length of zero.

Comparisons

The COALESCEC function searches character arguments, whereas the COALESCE function searches numeric arguments.

Example

```
data one;
   x = COALESCEC('', 'Hello');
   y = COALESCEC ('', 'Goodbye', 'Hello');
   put x=;
   put y=;
run;
```

The preceding statements produce these results:

```
x=Hello
y=Goodbye
```

See Also

Functions:

■ "COALESCE Function" on page 439

COLLATE Function

Returns a character string in the ASCII collating sequence.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Operating environment: The Linux OS uses the ASCII collating sequence

Syntax

COLLATE(*start-position* <, *end-position*>) | (*start-position* <, , *length*>)

Required Argument

start-position
specifies the numeric position in the collating sequence of the first character to be returned.

Interaction If you specify only *start-position*, COLLATE returns consecutive characters from that position to the end of the collating sequence or up to 255 characters, whichever comes first.

Optional Arguments

end-position
specifies the numeric position in the collating sequence of the last character to be returned.

For ASCII collating sequences, the characters that correspond to *end-position* values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to *end-position* values between 128 and 255 are available in certain ASCII operating environments, but the information that those characters represent varies with the operating environment.

Tips *end-position* must be larger than *start-position*.

If you omit *end-position* and use *length*, mark the *end-position* place with a comma.

length
specifies the number of characters in the collating sequence.

If you specify both *end-position* and *length*, COLLATE ignores *length*.

Default 200

Tip If you omit *end-position*, use *length* to specify the length of the result explicitly.

Details

A *collating sequence* is the order in which a set of characters are sorted. For example, when the SORT procedure is executed, the collating sequence determines the sort order (higher, lower, or equal to) of a particular character in relation to other characters in the set. The collating sequence is based on the session encoding.

The COLLATE function returns a string of ASCII characters. The ASCII collating sequence contains 256 positions, referenced with the position numbers 0 through 255. Characters above position 127 correspond to characters that are used in European languages that are defined in the ISO 8859 character set.

The string that the COLLATE function returns begins with the ASCII character that is specified by the *start-position* argument. If you specify the *end-position* argument, the COLLATE function returns a string that contains all the ASCII characters between the *start-position* and *end-position* arguments, inclusively. If you specify the *length* argument instead of the *end-position* argument, the COLLATE function returns a string that contains the specified number of characters, beginning at the *start-position*.

In a DATA step, if the COLLATE function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes. If you request a string longer than the remainder of the sequence, COLLATE returns a string through the end of the sequence and pads the end of the string with spaces. Because the full ASCII collating sequence is longer than 200 characters, the default return string is truncated to 200 characters. To return a string with a length of 201 to 256 ASCII characters, use one of these methods in the DATA step:

- Use a format such as $256. for the return string variable.
- Define the length of the return string variable by using a LENGTH statement.

Examples

Example 1

The following statements use the ASCII collating sequence:

```
data one;
   x=collate(48, , 10);
   y=collate(48, 57);
```

```
      put @1 x= @14 y=;
run;
```

The preceding statements produce these results:

```
x=0123456789 y=0123456789
```

Example 2: Returning an ASCII String by Using the Return Variable Default String Length

In this example, the return code variable **x** uses the default return string length of 200. Therefore, the COLLATE function returns 200 characters of the collating sequence to **x**. The length is set for **y** to 256 characters. You can see in the output that the length of **y** contains more characters.

```
data order;
   length y $256;
   x = collate(0,255);
   put x;
   y = collate(0,255);
   put y;
run;

proc contents data=order;
run;
```

Output 3.22 *Output to the Log from the COLLATE Function*

```
43   data order;
44      length y $256;
45      x = collate(0,255);
46      put x;
47      y = collate(0,255);
48      put y;
49   run;

 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`a
bcdefghijklmnopqrstuvwxyz{|}~□€|–┌┐┼┤└┴┘□‘’‖═╓╖╙╜□ ¡¢£¤¥¦§ ¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃ
ÄÅÆÇ
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`a
bcdefghijklmnopqrstuvwxyz{|}~□€|–┌┐┼┤└┴┘□‘’‖═╓╖╙╜□ ¡¢£¤¥¦§ ¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃ
ÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ
NOTE: The data set WORK.ORDER has 1 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Output 3.23 *Partial Output from the CONTENTS Procedure*

Alphabetic List of Variables and Attributes

#	Variable	Type	Len
2	x	Char	200
1	y	Char	256

Example 3: Returning an ASCII String of a Specific Length

In this example, the return code variable **y** uses a return string length of 56, and the COLLATE function returns the first 56 characters of the ASCII collating sequence. The remainder of the 200-character string is padded with spaces.

```
data _null_;
   y = collate(,,56);
   put y;
run;
```

Output 3.24 *Output of a Specific Length from the COLLATE Function*

```
31    data _null_;
32       y = collate(,,56);
33       put y;
34    run;

                                !"#$%&'()*+,-./01234567
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

See Also

Functions:

- "BYTE Function" on page 214
- "RANK Function" on page 1330

COMB Function

Computes the number of combinations of *n* elements taken *r* at a time.

Category: Combinatorial

Syntax

COMB(*n*, *r*)

Required Arguments

n

is a nonnegative integer that represents the total number of elements from which the sample is chosen.

r
is a nonnegative integer that represents the number of chosen elements.

Restriction $r \le n$

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

```
data one;
   x=comb(5,1);
   put x=;
run;
```

The preceding statements produce this result:

```
x=5
```

See Also

Functions:

- “FACT Function” on page 587
- “LCOMB Function” on page 1047
- “PERM Function” on page 1217

COMPARE Function

Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Tip: The DBCS equivalent function is KCOMPARE in *SAS National Language Support (NLS): Reference Guide*. See also "DBCS Compatibility" on page 448.

Syntax

COMPARE(*string-1*, *string-2* <, *modifier(s)*>)

Required Arguments

string-1
specifies a character constant, variable, or expression.

string-2
specifies a character constant, variable, or expression.

Optional Argument

modifier
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 SAS data set 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 1, 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.

> **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 0. If the arguments differ, here are the conditions:

- 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 Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*. There are minor differences between the COMPARE and KCOMPARE functions. Both functions accept varying numbers of arguments. Usage of the third argument is not compatible. This example shows the differences in the syntax.

COMPARE(*string-1*, *string-2* <, *modifier(s)*>)

KCOMPARE(*string-1* <, *position* <, *count*> >, *string-2*)

Examples

Example 1: Understanding the Order of Comparisons When Comparing Two Strings

This example compares two strings by using the COMPARE function.

```
data test;
   infile datalines missover;
   input string1 $char8. string2 $char8. modifiers $char8.;
   result=compare(string1, string2, modifiers);
```

```
   datalines;
1234567812345678
123     abc
abc     abx
xyz     abcdef
aBc     abc
aBc     AbC     i
   abc  abc
   abc  abc     l
 abc       abx
 abc       abx  l
ABC     'abc'n
ABC     'abc'n  n
 '$12'n $12     n
 '$12'n $12     nl
 '$12'n $12     ln
;

proc print data=test;
run;
```

Output 3.25 *Results of Comparing Two Strings by Using the COMPARE Function*

The SAS System

Obs	string1	string2	modifiers	result
1	12345678	12345678		0
2	123	abc		-1
3	abc	abx		-3
4	xyz	abcdef		1
5	aBc	abc		-2
6	aBc	AbC	i	0
7	abc	abc		-1
8	abc	abc	l	0
9	abc	abx		2
10	abc	abx	l	-3
11	ABC	'abc'n		1
12	ABC	'abc'n	n	0
13	'$12'n	$12	n	-1
14	'$12'n	$12	nl	1
15	'$12'n	$12	ln	0

Example 2: Truncating Strings by Using the COMPARE Function

This example uses the : (colon) modifier to truncate strings.

```
data test2;
   pad1=compare('abc', 'abc            ');
   pad2=compare('abc', 'abcdef         ');
   truncate1=compare('abc', 'abcdef',':');
   truncate2=compare('abcdef', 'abc',':');
   blank=compare('', 'abc',          ':');
run;

proc print data=test2 noobs;
run;
```

Output 3.26 *Results of Using the Truncation Modifier*

The SAS System

pad1	pad2	truncate1	truncate2	blank
0	-4	0	0	-1

See Also

Functions:

- “COMPGED Function” on page 456
- “COMPLEV Function” on page 463

CALL Routines:

- “CALL COMPCOST Routine” on page 232

COMPBL Function

Removes multiple blanks from a character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Interaction: This function supports the VARCHAR type.

Syntax

COMPBL(*source*)

Required Argument

source
specifies a character constant, variable, or expression to compress.

Details

Length of Returned Variable

In a DATA step, if the COMPBL function returns a value to a variable that has not previously been assigned a length, the length of that variable defaults to the length of the first argument.

The Basics

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 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

```
data one;
   string='Hey
   Diddle Diddle';
   string=compbl(string);
   put string=;
run;
```

The preceding statements produce this result:

```
string=HeyDiddle Diddle
```

Example 2

```
data one;
string='125 E Main St';
length address $10;
address=compbl(string);
put address=;
run;
```

The preceding statements produce this result:

```
address=125 E Main
```

See Also

Functions:

- “COMPRESS Function” on page 467

COMPFUZZ Function

Performs a fuzzy comparison of two numeric values.

Category: Mathematical

Syntax

COMPFUZZ(*value-1*, *value-2* <, *fuzz* <, *scale*>>)

Required Arguments

value-1
specifies the first of two numeric values to be compared.

value-2
specifies the second numeric value to be compared.

Optional Arguments

fuzz
is a nonnegative numeric value that specifies the relative threshold for comparisons. Values greater than or equal to 1 are treated as multiples of the machine precision.

Default 1024

scale
specifies the scale factor.

Default MAX (ABS (*value-1*), ABS (*value-2*))

Details

The COMPFUZZ function returns the following values if you specify all four arguments:

- –1 if *value-1* < *value-2* – threshold
- 0 if ABS(*value-1* - *value-2*) ≤ threshold
- 1 if *value-1* > *value-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.

If any argument has a missing value, COMPFUZZ returns a missing value.

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 use *n* as the fuzz value and the sum of the absolute values as the scale factor, as shown in this DATA step:

```
data _null_;
   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= / sum2= / 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.                                                    */
   compfuzz = compfuzz(sum1, sum2, 8);
   put "fuzz only (wrong):         " compfuzz=;
      /* Using a fuzz factor and a scale value gives the correct
result. */
   scale = abs(x1) + abs(x2) + abs(x3) + abs(x4);
   compfuzz = compfuzz(sum1, sum2, 8, scale);
   put "fuzz and scale (correct):  " compfuzz=;
run;
```

SAS writes the following results to the log:

Example Code 3.2 *Partial SAS Output for the COMPFUZZ Function*

```
sum1=1.8095238209
sum2=1.8095238095
diff=1.1353266E-8
fuzz only (wrong):         compfuzz=1
fuzz and scale (correct):  compfuzz=0
```

See Also

Functions:

- “FUZZ Function” on page 743
- “ROUND Function” on page 1348

COMPFUZZ_MISS Function

Performs comparisons of missing numeric values and fuzzy comparisons of nonmissing numeric values.

Category: Mathematical

See: "COMPFUZZ Function"

Syntax

COMPFUZZ_MISS(*value-1*, *value-2* <<, *fuzz*<<*scale*>>>>)

Required Arguments

value-1
specifies the first of two numeric values to be compared.

value-2
specifies the second numeric value to be compared.

Optional Arguments

fuzz
is a nonnegative numeric value that specifies the relative threshold for comparisons. Values greater than or equal to 1 are treated as multiples of the machine precision.

Default 1024

scale
specifies the scale factor.

Default MAX (ABS (*value-1*), ABS (*value-2*))

Details

The COMPFUZZ function returns the following values if you specify all four arguments:

- –1 if *value-1* < *value-2* – threshold
- 0 if ABS(*value-1* - *value-2*) ≤ threshold
- 1 if *value-1* > *value-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')

If value-1 and value-2 are both missing, COMPFUZZ_MISS compares the missing values the same way that the less-than and greater-than operators do; see "Order of Missing Values" in *SAS Language Programmer's Guide for SAS Viya Workbench* for the order of SAS missing values.

- If value-1 < value-2, COMPFUZZ_MISS returns -1.
- If value-1 = value-2, COMPFUZZ_MISS returns -0
- If value-1 > value-2, COMPFUZZ_MISS returns 1.

If value-1 is missing and value-2 is not missing, COMPFUZZ_MISS returns -1, because all missing values are considered less than all nonmissing values. Likewise, If value-1 is not missing and value-2 is missing, COMPFUZZ_MISS returns 1.

If neither value-1 nor value-2 is missing, COMPFUZZ function returns the following values:

- `-1 if value-1 < value-2 – threshold`
- `0 if ABS(value-1 - value-2) ≤ threshold`
- `1 if value-1 > value-2 + threshold`

Where:

- `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.

If any argument has a missing value, COMPFUZZ returns a missing value.

Comparisons

COMPFUZZ is the same as COMPFUZZ_MISS except that COMPFUZZ returns a missing value If any argument has a missing value.

COMPGED Function

Returns the generalized edit distance between two strings.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Syntax

COMPGED(*string-1*, *string-2* <, *cutoff*> <, *modifier(s)*>)

Required Arguments

string-1
specifies a character constant, variable, or expression.

string-2
specifies a character constant, variable, or expression.

Optional Arguments

cutoff
is a numeric constant, variable, or expression. 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 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.

Operation	Default Cost in Units	Description of Operation
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 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.

Operation	Default Cost in Units	Description of Operation
		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 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 CALL COMPCOST routine 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 CALL COMPCOST routine 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 CALL COMPCOST routine and the COMPGED function. 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.

Example

This example uses the default costs to calculate the generalized edit distance.

```
data test;
   infile datalines missover;
   input String1 $char8. +1 String2 $char8. +1 Operation $40.;
   GED=compged(string1, string2);
   datalines;
baboon   baboon   match
baXboon  baboon   insert
baoon    baboon   delete
baXoon   baboon   replace
baboonX  baboon   append
baboo    baboon   truncate
babboon  baboon   double
babon    baboon   single
baobon   baboon   swap
bab oon  baboon   blank
bab,oon  baboon   punctuation
bXaoon   baboon   insert+delete
bXaYoon  baboon   insert+replace
bXoon    baboon   delete+replace
Xbaboon  baboon   finsert
aboon    baboon   trick question: swap+delete
Xaboon   baboon   freplace
axoon    baboon   fdelete+replace
axoo     baboon   fdelete+replace+truncate
axon     baboon   fdelete+replace+single
baby     baboon   replace+truncate*2
```

```
balloon  baboon   replace+insert
;

proc print data=test label;
   label GED='Generalized Edit Distance';
   var String1 String2 GED Operation;
run;
```

Output 3.27 *Generalized Edit Distance Based on Operation*

The SAS System

Obs	String1	String2	Generalized Edit Distance	Operation
1	baboon	baboon	0	match
2	baXboon	baboon	100	insert
3	baoon	baboon	100	delete
4	baXoon	baboon	100	replace
5	baboonX	baboon	50	append
6	baboo	baboon	10	truncate
7	babboon	baboon	20	double
8	babon	baboon	20	single
9	baobon	baboon	20	swap
10	bab oon	baboon	10	blank
11	bab,oon	baboon	30	punctuation
12	bXaoon	baboon	200	insert+delete
13	bXaYoon	baboon	200	insert+replace
14	bXoon	baboon	200	delete+replace
15	Xbaboon	baboon	200	finsert
16	aboon	baboon	120	trick question: swap+delete
17	Xaboon	baboon	200	freplace
18	axoon	baboon	300	fdelete+replace
19	axoo	baboon	310	fdelete+replace+truncate
20	axon	baboon	320	fdelete+replace+single
21	baby	baboon	120	replace+truncate*2
22	balloon	baboon	200	replace+insert

See Also

Functions:

- “COMPARE Function” on page 447
- “COMPLEV Function” on page 463

CALL Routines:

- “CALL COMPCOST Routine” on page 232

COMPLEV Function

Returns the Levenshtein edit distance between two strings.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Syntax

COMPLEV(*string-1*, *string-2* <, *cutoff*> <, *modifier(s)*>)

Required Arguments

string-1
specifies a character constant, variable, or expression.

string-2
specifies a character constant, variable, or expression.

Optional Arguments

cutoff
specifies a numeric constant, variable, or expression. 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 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.

```
data test;
   infile datalines missover;
   input string1 $char8. string2 $char8. modifiers $char8.;
   result=complev(string1, string2, modifiers);
```

```
   datalines;
1234567812345678
abc      abxc
ac       abc
aXc      abc
aXbZc    abc
aXYZc    abc
WaXbYcZ  abc
XYZ      abcdef
aBc      abc
aBc      AbC      i
  abc    abc
  abc    abc      l
AxC      'abc'n
AxC      'abc'n   n
;

proc print data=test;
run;
```

Output 3.28 *Output from Comparing Two Strings by Computing the Levenshtein Edit Distance*

The SAS System

Obs	string1	string2	modifiers	result
1	12345678	12345678		0
2	abc	abxc		1
3	ac	abc		1
4	aXc	abc		1
5	aXbZc	abc		2
6	aXYZc	abc		3
7	WaXbYcZ	abc		4
8	XYZ	abcdef		6
9	aBc	abc		1
10	aBc	AbC	i	0
11	abc	abc		2
12	abc	abc	l	0
13	AxC	'abc'n		6
14	AxC	'abc'n	n	1

See Also

Functions:

- “COMPARE Function” on page 447
- “COMPGED Function” on page 456

CALL Routines:

- “CALL COMPCOST Routine” on page 232

COMPOUND Function

Returns compound interest parameters.

Category: Financial

Syntax

COMPOUND(*a*, *f*, *r*, *n*)

Required Arguments

a

is numeric and specifies the initial amount.

Range $a \geq 0$

f

is numeric and specifies the future amount (at the end of *n* periods).

Range $f \geq 0$

r

is numeric and specifies the periodic interest rate expressed as a fraction.

Range $r \geq 0$

n

is an integer and specifies the number of compounding periods.

Range $n \geq 0$

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

$$(1 + r)^n$$

are equal to 1.

Note: If you choose *r* as your missing value, COMPOUND returns an error.

Example

The accumulated value of an investment of USD 2000 at a nominal annual interest rate of 9%, compounded monthly after 30 months, can be expressed as shown in the example.

The second argument has been set to missing, which indicates 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 one;
   future=compound(2000, ., 0.09/12, 30);
   put future=;
run;
```

The preceding statements produce this result:

```
future=2502.5435276
```

COMPRESS Function

Returns a character string with specified characters removed from the original string.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: The DBCS equivalent function is KCOMPRESS.

Syntax

COMPRESS(*source* ,< *characters*> ,< *modifier(s)*>)

Required Argument

source
specifies a character constant, variable, or expression from which specified characters are removed.

Optional Arguments

characters
specifies a character constant, variable, or expression that initializes a list of characters.

By default, the characters in this list are removed from the *source* argument. If you specify the K modifier in the third argument, only the characters in this list are kept in the result.

> **TIP** You can add more characters to this list by using other modifiers in the third argument.

> **TIP** Enclose a literal string of characters in quotation marks.

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 quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

Details

Length of Returned Variable

In a DATA step, if the COMPRESS function returns a value to a variable that has not previously been assigned a length, that variable is given the length of the first argument.

The Basics

The COMPRESS function allows null arguments. A null argument is treated as a string that has a length of 0.

Based on the number of arguments, the COMPRESS function works as follows:

Number of Arguments	Result
only the first argument, *source*	All blanks have been removed from the argument. If the argument is completely

Number of Arguments	Result
	blank, the result is a string with a length of 0. If you assign the result to a character variable with a fixed length, the value of that variable is padded with blanks to fill its defined length.
the first two arguments, *source* and *characters*	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.

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. For example, 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 can use either of these function calls:

```
compress(source, "1234567890+-");
compress(source, "+-", "d");
```

Examples

Example 1: Compressing Blanks

```
data one;
   a='AB C D ';
   b=compress(a);
   put b=;
run;
```

The preceding statements produce this result:

```
b=ABCD
```

Example 2: Compressing Lowercase Letters

```
data _null_;
   x='123-4567-8901 B 234-5678-9012 c';
   y=compress(x, 'ABCD', 'l');
   put y=;
run;
```

The preceding statements produce this result:

```
y=123-4567-8901  234-5678-9012
```

Example 3: Compressing Space Characters

```
data one;
   x='1 2 3 4 5';
   y=compress(x, , 's');
   put y=;
run;
```

The preceding statements produce this result:

```
y=12345
```

Example 4: Keeping Characters in the List

```
data one;
   x='Math A English B Physics A';
   y=compress(x, 'ABCD', 'k');
   put y=;
run;
```

The preceding statements produce this result:

```
y=ABA
```

Example 5: Compressing a String and Returning a Length of 0

```
data _null_;
   x='';
   l=lengthn(compress(x));
   put l=;
run;
```

The preceding statements produce this result:

```
l=0
```

See Also

Functions:

- “COMPBL Function” on page 450
- “LEFT Function” on page 1048
- “TRIM Function” on page 1470

COMPSRV_OVAL Function

Returns the original, possibly unsafe, value of an input parameter or global macro variable that is passed into the Compute server.

Category: Compute Server

Alias: APPSRV_UNSAFE

Note: A returned value is specified to an input variable name: `variable-name=COMPSRV_OVAL(macro-variable-name);`

CAUTION: **Improper use of this function could allow a malicious code injection. Use this function only in the DATA step to prevent code injection.**

Syntax

COMPSRV_OVAL(*macro-variable-name*);

Required Argument

macro-variable-name
specifies the name of the input parameter or global macro variable (with no leading ampersands).

Details

The COMPSRV_OVAL function accepts the name of an input parameter or any global macro variable and returns the original, unmasked value including semicolons (;). This function removes delta characters from Compute server input parameters so that the value can be used in statements that do not remove delta characters.

Examples

Example 1

In this example, the NAME_VALUE input parameter has the value **`Year to Date Revenue; 123456`**. The SAS program uses the semicolon as a delimiter and parses the value into two variables. The NAME variable should have the value **`This year's revenue`**, and the VALUE variable should have the value **`123456`**.

```
data work.test;
length name $256 value 8;
   name=scan("&NAME_VALUE", 1, ';');
   value=input(scan("&NAME_VALUE", 2, ';'), best.);
   putlog name= value=;
run;
```

You get unexpected results because semicolons are removed from the input parameter to prevent malicious code injections.

```
name=Year to Date Revenue 123456 value=.
```

Example 2

In this example, COMPSRV_OVAL returns the original value of the NAME_VALUE macro variable, including the semicolon.

```
data work.test;
   length unsafe_param $32767 name $256 value 8;
   unsafe_param = compsrv_oval('NAME_VALUE');
   name=scan(unsafe_param, 1, ';');
   value=input(scan(unsafe_param, 2, ';'), best.);
   putlog name= value=;
run;
```

Here are the expected results from the code:

```
name=Year to Date Revenue value=123456
```

Example 3

Avoid using COMPSRV_OVAL or unsafe macro variable values outside the DATA step to prevent malicious code injections. In this example, the value **Program Executed on &SYSDATE9** is specified for the TITLE_TEXT input parameter.

```
title "&TITLE_TEXT";
proc print...;
run;
```

In the output, the **SYSDATE9** value appears as text in the title because the ampersand character (&) is masked in the input parameter to help prevent malicious code injections.

```
Program Executed on &SYSDATE9
```

COMSRV_OVAL resolves this issue.

```
%let UNSAFE_TITLE_TEXT=%sysfunc(compsrv_oval(TITLE_TEXT));
title "&UNSAFE_TITLE_TEXT";
proc print ... ;
```

```
run;
```

Here is the desired output from the code:

```
Program Executed on <date program executed>
```

CAUTION
Avoid using this %LET technique because the %LET statement is vulnerable to a malicious code injection. For example, if this technique is specified for the input parameter, the following code for the value **`TITLE_TEXT, PROCDATASETS`** executes when the TITLE_TEXT macro variable is resolved.

```
;proc datasets lib=work;
 run;
quit;*
```

Here are the results that are written to the log:

```
2    %let UNSAFE_TITLE_TEXT=%sysfunc(compsrv_oval(TITLE_TEXT));
NOTE: PROCEDURE DATASETS used (Total process time):
      real time           0.35 seconds
      cpu time            0.02 seconds
```

COMPSRV_UNQUOTE2 Function

Unmasks matched pairs of quotation marks in an input parameter or global macro variable.

Category: Compute Server

Alias: STPSRV_UNQUOTE2

Syntax

%SYSCALL STPSRV_UNQUOTE2(*macro-variable-name*)

Required Argument

macro-variable-name
specifies the name of the input parameter or global macro variable (with no leading ampersands).

Details

The COMPSRV_UNQUOTE2 function accepts the name of an input parameter or any global macro variable, and unmasks matched pairs of single or double quotation marks. The function does not return a value. Instead, the value of the macro variable is changed. This function removes delta characters from compute server input parameters so that the parameters can be used in statements that require quotation marks.

Examples

Example 1

The ODS_TEXT input parameter has the value **R&D Expenditures** (including double quotation marks) and is used in this code.

COMPSRV_UNQUOTE2 is typically called with %SYSCALL in open macro code.

```
ods html text=&ODS_TEXT ... ;
(SAS Procedure code goes here);
ods htmlclose;
```

An error occurs because the value of ODS_TEXT contains delta characters.

```
2    ods html text=&ODS_TEXT ... ;
NOTE: Line generated by the macro variable "ODS_TEXT".
2    "R&D Expenditures"
     -
     22
      -
      200
ERROR 22-322: Expecting a quoted string.
ERROR 200-322: The symbol is not recognized and will be ignored.
```

Call COMPSRV_UNQUOTE2 before using ODS_TEXT to correct the error.

COMPSRV_UNQUOTE2 removes the delta characters that mask the double quotation marks so that the macro variable can be used in the ODS_TEXT option.

```
%syscall compsrv_unquote2(ODS_TEXT);
ods html text=&ODS_TEXT ... ;
   (SAS-procedure code goes here);
ods html close;
```

Example 2

In this example, the TITLE_TEXT input parameter has the value **"R&D Expenditures"**, including double quotation marks, and is used in this code:

```
title &TITLE_TEXT;
proc print ...;
run;
```

The code runs without error, but the double quotation marks appear in the title.

```
"R&D Expenditures"
```

Call COMPSRV_UNQUOTE2 before using TITLE_TEXT to resolve the issue of double quotation marks.

COMPSRV_UNQUOTE2 removes the delta characters that mask the double quotation marks, so that the TITLE_TEXT macro variable can be used in the TITLE statement.

```
%syscall compsrv_unquote2(TITLE_TEXT);
title &TITLE_TEXT
proc print ...;
run;
```

```
R&D Expenditures
```

CONSTANT Function

Computes machine and mathematical constants.

Category: Mathematical

Syntax

CONSTANT(*constant* <, *parameter*>)

Required Argument

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
'EULER'	Euler constant

Constant	Description
'GOLDEN'	Numerical optimization and root finding
'PI'	Pi
'EXACTINT' <,nbytes>	Exact integer
'BIG'	The largest double-precision number
'BIGRECIP'	The largest double-precision floating-point number
'LOGBIG' <,base>	The log with respect to *base* of BIG
'LOGBIGRECIP' <,base>	The log with respect to base of the largest double-precision floating-point number
'SQRTBIG'	The square root of BIG
'SMALL'	The smallest double-precision number
'SMALLRECIP'	The smallest double-precision floating-point number
'LOGSMALL' <,base>	The log with respect to *base* of SMALL
'LOGSMALLRECIP' <,base>	The log with respect to base of the smallest double-precision floating-point number
'SQRTSMALL'	The square root of SMALL
'MACEPS'	Machine precision constant, also called machine epsilon
'LOGMACEPS' <,base>	The log with respect to *base* of MACEPS
'SQRTMACEPS'	The square root of MACEPS

Optional Argument

parameter
is an optional numeric parameter. Some of the constants specified in *constant* have an optional argument that 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')` does 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:

$$\lim_{n \to \infty} \left\{ \sum_{j=1}^{j=n} \frac{1}{j} - \log(n) \right\} \approx 0.577215664901532860$$

Golden

CONSTANT('GOLDEN')

GOLDEN is a common constant in numerical optimization and root finding. GOLDEN is described by the following equation:

$$GOLDEN = (sqrt(5) - 1)/2)$$

The value `g= constant('Golden')` is one of two ways to define the GOLDEN ratio. The inverse quantity `phi = 1/g = (sqrt(5) + 1)/2` is the conventional definition of the GOLDEN ratio. `phi = 1 + g` also defines the GOLDEN ratio.

Table 3.3 *Two Ways to Define the GOLDEN Ratio.*

Expression	Value
`g = constant('GOLDEN')`	`0.61803`
`phi = 1/g`	`1.61803{{}}`

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.

CONSTANT('BIGRECIP')

This case returns the largest double precision floating point number (8-bytes) that is representable on the machine, that can be inverted, and its inverse can also be inverted.

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 the *base* of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

You can 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.

You can exponentiate any floating-point number less than or equal to CONSTANT('LOGBIG') by using the exponential function, EXP, without causing any overflows.

CONSTANT('LOGBIGRECIP' <, *base*>)

Arguments

base
is a numeric value that is the base of the logarithm.

Default The natural base, E.

Restriction The *base* must be greater than the value of 1+SQRTMACEPS.

Note $\log \frac{1}{a}x = -\log_a x$

This case returns the logarithm with respect to the *base* of the largest double-precision floating-point number (8-bytes) that is representable on your computer, can be inverted, and its inverse can be inverted.

You can exponentiate the given *base* raised to a power less than or equal to CONSTANT('LOGBIGRECIP', *base*) by using the power operation (**) without causing any overflows.

You can exponentiate any floating-point number less than or equal to CONSTANT('LOGBIGRECIP') 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.

CONSTANT('SMALLRECIP')

This case returns the smallest double precision floating point number (8-bytes) that is representable on the machine, can be inverted, and its inverse can also be inverted.

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 the *base* of the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

You can 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.

You can exponentiate any floating-point number greater than or equal to CONSTANT('LOGSMALL') by using the exponential function, EXP, without causing any underflows or 0.

CONSTANT('LOGSMALLRECIP' <, *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.

Note $\log \frac{1}{a} x = -\log_a x$

This case returns the logarithm with respect to the *base* of the smallest double-precision floating-point number (8-bytes) that is representable on your computer machine, can be inverted, and its inverse can also be inverted.

You can exponentiate the given *base* raised to a power greater than or equal to CONSTANT('LOGSMALLRECIP', *base*) by using the power operation (**) without causing any underflows or 0.

You can exponentiate any floating-point number greater than or equal to CONSTANT('LOGSMALLRECIP') by using the exponential function, EXP, without causing any underflows 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, which is called machine epsilon, 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 1+SQRTMACEPS.

This case returns the logarithm with respect to the *base* of CONSTANT('MACEPS').

The Square Root of MACEPS

CONSTANT('SQRTMACEPS')

This case returns the square root of CONSTANT('MACEPS').

CONVX Function

Returns the convexity for an enumerated cash flow.

Category: Financial

Syntax

CONVX(*y*, *f*, *c(1), ..., c(k)*)

Required Arguments

y
specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range 0 < *y* < 1

f
specifies the frequency of cash flows per period.

Range $f > 0$

c(1), ..., c(k)
specifies a list of cash flows.

Details

The CONVX function returns this value:

$$C = \sum_{k=1}^{K} \frac{k(k+f)\frac{c(k)}{(1+y)^{\frac{k}{f}}}}{P\left((1+y)^2\right)f^2}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example

```
data _null_;
   c=convx(1/20,
1, .33, .44, .55, .49, .50, .22, .4, .8, .01, .36, .2, .4);
   put c;
run;
```

SAS writes the following result to the log:

```
42.3778
```

CONVXP Function

Returns the convexity for a periodic cash flow stream such as a bond.

Category: Financial

Syntax

CONVXP(*A*, *c*, *n*, *K*, k_0, *y*)

Required Arguments

A

specifies the par value.

Range $A > 0$

c

specifies the nominal per-period coupon rate, expressed as a fraction.

Range $0 \le c < 1$

n

specifies the number of coupons per period.

Range $n > 0$ and is an integer

K

specifies the number of remaining coupons.

Range $K > 0$ and is an integer

k_o

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_0 \le \frac{1}{n}$

y

specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Details

The CONVXP function returns this value:

$$C = \frac{1}{n^2}\left(\frac{\sum_{k=1}^{K} t_k(t_k+1)\frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)^2}\right)$$

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:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{tk}}$$

Example

In the following example, 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 _null_;
   y=convxp(1000, .01, 4, 14, .33/2, .08);
   put y;
run;
```

SAS writes the following result to the log:

```
11.729001987
```

COS Function

Returns the cosine.

Category: Trigonometric

Syntax

COS(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression and is expressed in radians. If the magnitude of *argument* is so great that `mod(argument,pi)` is accurate to less than about three decimal places, COS returns a missing value.

Example

```
data one;
   x=cos(0.5);
   put x=;
```

```
   y=cos(0);
   put y=;
   z=cos(3.14159/3);
   put z=;
run;
```

The preceding statements produce these results:

```
x=0.8775825619
y=1
z=0.500000766
```

COSH Function

Returns the hyperbolic cosine.

Category: Hyperbolic

Syntax

COSH(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The COSH function returns the hyperbolic cosine of the argument given by this equation:

$$(\varepsilon^{argument} + \varepsilon^{-argument})/2$$

Example

```
data one;
   x=cosh(0);
   put x=;
   y=cosh(-5.0);
   put y=;
   z=cosh(0.5);
   put z=;
run;
```

The preceding statements produce these results:

```
x=1
y=74.209948525
z=1.1276259652
```

COT Function

Returns the cotangent.

Category: Trigonometric

Syntax

COT(*argument*)

Required Argument

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

```
data one;
   x=cot(0.5);
   put x=;
   y=cot(1);
   put y=;
   z=cot(3.14159/3);
   put z=;
run;
```

The preceding statements produce these results:

```
x=1.8304877217
y=0.6420926159
z=0.5773514486
```

Note: If you use `x=cot(0);`, 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 485
- "CSC Function" on page 499
- "SEC Function" on page 1380
- "SIN Function" on page 1390
- "TAN Function" on page 1447

COUNT Function

Counts the number of times that a specified substring appears within a character string.

Category: Character

Restriction: This function is assigned an I18N Level 0 status and is designed for SBCS data. However, if the first argument, *string*, is specified as varchar and there are multi-byte characters, the COUNT function processes the multi-byte data. For more information, see "Internationalization Compatibility for SAS String Functions" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Tip: You can use the KCOUNT function for DBCS processing, but the functionality is different. See DBCS Compatibility on page 490.

Syntax

COUNT(*string*, *substring* <, *modifier(s)*>)

Required Arguments

string

specifies a character constant, variable, or expression in which substrings are to be counted.

Tip Enclose a literal string of characters in quotation marks.

substring
is a character constant, variable, or expression that specifies the substring of characters to count in *string*.

Tip Enclose a literal string of characters in quotation marks.

Optional Argument

modifier
is a character constant, variable, or expression that specifies one or more modifiers.

The following *modifiers* can be in uppercase or lowercase:

i ignores character case during the count. If this modifier is not specified, COUNT counts character substrings only with the same case as the characters in *substring*.

t trims trailing blanks from *string* and *substring*.

Tip If the *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.

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.

Processing SBCS and DBCS Data

The COUNT function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria for SBCS and DBCS processing:

- If *string* is not declared as varchar and you are processing single-byte data, then COUNT processes SBCS.
- If *string* is declared as varchar and you are processing multi-byte data, then COUNT processes DBCS.

KCOUNT DBCS Compatibility

For DBCS processing, you can use the "KCOUNT Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*, but the functionality is different.

If the value of *substring* in the COUNT function is longer than 2 bytes, the COUNT function can handle DBCS strings. The following examples show the differences in syntax:

COUNT(*string*, *substring* <, *modifier(s)*>)

KCOUNT(*string*)

Comparisons

The COUNT function counts substrings of characters in a character string, whereas the COUNTC function counts individual characters in a character string.

Example

```
data _null_;
   xyz='This is a thistle? Yes, this is a thistle.';
   howmanythis=count(xyz,'this');
   put howmanythis=;
run;
```

The preceding statements produce this result:

```
howmanythis=3
```

```
data one;
   xyz='This is a thistle? Yes, this is a thistle.';
   howmanyis=count(xyz,'is');
   put howmanyis=;
run;
```

The preceding statements produce this result:

```
howmanyis=6
```

```
data one;
   howmanythis_i=count('This is a thistle? Yes, this is a
thistle.','this','i');
   put howmanythis_i=;
run;
```

The preceding statements produce this result:

```
howmanythis_i=4
```

```
data one;
   variable1='This is a thistle? Yes, this is a thistle.';
   variable2='is ';
   variable3='i';
   howmanyis_i=count(variable1,variable2,variable3);
   put howmanyis_i=;
run;
```

The preceding statements produce this result:

```
howmanyis_i=4
```

```
data one;
   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=;
run;
```

The preceding statements produce this result:

```
howmanyis_it=6
```

See Also

Functions:

- “COUNTC Function” on page 491
- “COUNTW Function” on page 495
- “FIND Function” on page 679
- “INDEX Function” on page 947

COUNTC Function

Counts the number of characters that appear or do not appear in a list of characters.

Category: Character

Restriction: This function is assigned an I18N Level 0 status and is designed for SBCS data. However, if the first argument, *string*, is specified as varchar and there are multi-byte characters, the COUNTC function processes the multi-byte data. For more information, see “Internationalization Compatibility for SAS String Functions” in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Syntax

COUNTC(*string*, *character-list* <, *modifier(s)*>)

Required Arguments

string
specifies a character constant, variable, or expression in which characters are counted.

Tip Enclose a literal string of characters in quotation marks.

character-list
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 *modifier* argument. If you specify the V modifier, all characters that are not in this list are counted. You can add more characters to the list by using other modifiers.

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.

Optional Argument

modifier
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.
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 *character-list* and *modifier* arguments only once, at the first call to this instance of COUNTC. If you change the value

of *character-list* 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 *character-list*. 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, 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.

Tip If *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks.

Details

Basics

The COUNTC function allows character arguments to be null. Null arguments are treated as character strings with a length of 0. If there are no characters in the list of characters to be counted, COUNTC returns 0.

Processing SBCS and DBCS Data

The COUNTC function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria for SBCS and DBCS processing:

- If *string* is not declared as varchar and you are processing single-byte data, then COUNTC processes SBCS.
- If *string* is declared as varchar and you are processing multi-byte data, then COUNTC processes DBCS.

Comparisons

The COUNTC function counts individual characters in a character string, whereas the COUNT function counts substrings of characters in a character string.

Example

This example uses the COUNTC function with and without modifiers to count the number of characters in a string.

```
data test;
   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');
run;

proc print data=test noobs;
run;
```

Output 3.29 *Output 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	16	11

See Also

Functions:

- “ANYALNUM Function” on page 147
- “ANYALPHA Function” on page 150
- “ANYCNTRL Function” on page 152
- “ANYDIGIT Function” on page 154
- “ANYGRAPH Function” on page 158
- “ANYLOWER Function” on page 161
- “ANYPRINT Function” on page 165
- “ANYPUNCT Function” on page 168
- “ANYSPACE Function” on page 170
- “ANYUPPER Function” on page 173
- “ANYXDIGIT Function” on page 175

- “NOTALNUM Function” on page 1142
- “INDEXC Function” on page 949
- “VERIFY Function” on page 1512
- “NOTALPHA Function” on page 1144
- “NOTCNTRL Function” on page 1147
- “NOTDIGIT Function” on page 1149
- “NOTGRAPH Function” on page 1156
- “NOTLOWER Function” on page 1159
- “NOTPRINT Function” on page 1163
- “NOTPUNCT Function” on page 1167
- “NOTSPACE Function” on page 1169
- “NOTUPPER Function” on page 1172
- “NOTXDIGIT Function” on page 1174
- “FINDC Function” on page 683

COUNTW Function

Counts the number of words in a character string.

Category: Character

Restriction: This function is assigned an I18N Level 0 status and is designed for SBCS data. However, if the first argument, *string*, is specified as varchar and there are multi-byte characters, the COUNTW function processes the multi-byte data. For more information, see “Internationalization Compatibility for SAS String Functions” in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Syntax

COUNTW(*string* <, <*character(s)*> <, <*modifier(s)*>>>)

Required Argument

string
specifies a character constant, variable, or expression in which words are counted.

Optional Arguments

character
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, all characters that are not in this list are delimiters. You can add more characters to the list by using other modifiers.

Tip Character arguments can be null. Null arguments are treated as character strings with a length of 0. Numeric arguments cannot be null.

modifier
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, 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 0. If the M modifier is not specified, 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 *characters* 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, 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.

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 one argument, 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, the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

Using the M Modifier

If you do not use the M modifier, a word must contain at least one character. If you use the M modifier, a word can have a length of 0. 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.

Processing SBCS and DBCS Data

The COUNTW function is designed to process SBCS data, but it can process DBCS data with certain conditions. Here are the criteria for SBCS and DBCS processing:

- If *string* is not declared as varchar or you are processing single-byte data, then COUNTW processes SBCS.
- If *string* is declared as varchar and you are processing multi-byte data, then COUNTW processes DBCS.

Example

The following example shows how to use the COUNTW function with the M and P modifiers.

```
data test;
   length default blanks mp 8;
   input string $char60.;
   default=countw(string);
   blanks=countw(string, ' ');
   mp=countw(string, , 'mp');
   datalines;
The quick brown fox jumps over the lazy dog.
2+2=4
//unix/path/names/use/slashes
\\Windows\Path\Names\Use\Backslashes
;
run;

proc print noobs data=test;
run;
```

Output 3.30 *Output from the COUNTW Function*

The SAS System

default	blanks	mp	string
9	9	2	The quick brown fox jumps over the lazy dog.
2	1	3	2+2=4
5	1	7	//unix/path/names/use/slashes
1	1	7	\\Windows\Path\Names\Use\Backslashes

See Also

Functions:

- “COUNT Function” on page 488
- “COUNTC Function” on page 491
- “FINDW Function” on page 692
- “SCAN Function” on page 1366

CALL Routines:

- “CALL SCAN Routine” on page 326

CSC Function

Returns the cosecant.

Category: Trigonometric

Syntax

CSC(*argument*)

Required Argument

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 CSC function is related to the SIN function in this way:

```
csc(x)=1/sin(x)
```

Example

```
data one;
   x=csc(0.5);
   put x=;
   y=csc(1);
```

```
   put y=;
   z=csc(3.14159/3);
   put z=;
run;
```

The preceding statements produce these results:

```
x=2.0858296429
y=1.1883951058
z=1.1547011281
```

Note: If you use `x=csc(0);`, 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 485
- "COT Function" on page 487
- "SEC Function" on page 1380
- "SIN Function" on page 1390
- "TAN Function" on page 1447

CSS Function

Returns the corrected sum of squares.

Category: Descriptive Statistics

Syntax

CSS(*argument-1*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one nonmissing argument is required. Otherwise, the function returns a missing value. If you have more than one argument, the argument list can consist of a variable list, which is preceded by OF.

Example

```
data one;
   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=;
run;
```

The preceding statements produce these results:

```
x1=18.75
x2=10
x3=4.6666666667
x4=101.11574074
```

CUMIPMT Function

Returns the cumulative interest paid on a loan between the start and end periods.

Category: Financial

Syntax

CUMIPMT(*rate*, *number-of-periods*, *principal-amount*, <*start-period*>, <*end-period*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

number-of-periods
specifies the number of payment periods. *Number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. If a missing value is specified, 0 is assumed.

Optional Arguments

start-period
specifies the start period for the calculation.

end-period
specifies the end period for the calculation.

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. If *type* is omitted or if a missing value is specified, 0 is assumed.

Examples

Example 1

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 with these statements:

```
data one;
   TotalInterest=CUMIPMT(0.09/12, 360, 125000, 13, 24, 0);
   put TotalInterest=;
run;
```

The preceding statements produce this result:

```
TotalInterest=11135.232131
```

Example 2

The interest that is paid on the first period of the same loan is computed with these statements:

```
data one;
   first_period_interest=CUMIPMT(0.09/12, 360, 125000, 1, 1, 0);
   put first_period_interest=;
run;
```

The preceding statements produce this result:

```
first_period_interest=937.5
```

CUMPRINC Function

Returns the cumulative principal paid on a loan between the start and end periods.

Category: Financial

Syntax

CUMPRINC(*rate*, *number-of-periods*, *principal-amount*, <*start-period*>, <*end-period*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

number-of-periods
specifies the number of payment periods. *Number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. If a missing value is specified, 0 is assumed.

Optional Arguments

start-period
specifies the start period for the calculation.

end-period
specifies the end period for the calculation.

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. If *type* is omitted or if a missing value is specified, 0 is assumed.

Examples

Example 1

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 with these statements:

```
data one;
   PrincipalYear2=CUMPRINC(0.09/12, 360, 125000, 12, 24, 0);
   put PrincipalYear2=;
run;
```

The preceding statements produce this result:

```
PrincipalYear2=1008.2344127
```

Example 2

The principal that is paid on the second year of the same loan with beginning-of-period payments is computed with these statements:

```
data one;
   PrincipalYear2b=CUMPRINC(0.09/12, 360, 125000, 12, 24, 1);
   put PrincipalYear2b=;
run;
```

The preceding statements produce this result:

```
PrincipalYear2b=1000.7289456
```

CUROBS Function

Returns the observation number of the current observation.

Category: SAS File I/O

Requirement: Use this function only with an uncompressed SAS data set that is accessed using a native library engine.

Syntax

CUROBS(*data-set-id*)

Required Argument

data-set-id
is a numeric value that specifies the data set identifier that the OPEN function returns.

Details

If the engine that is being used does not support observation numbers, the function returns a missing value.

With a SAS view, the function returns the relative observation number, which is the number of the observation within the SAS view (as opposed to the number of the observation within any related SAS data set).

Example

This example uses the FETCHOBS function to fetch the 10th observation in the data set MYDATA. The value of OBSNUM returned by CUROBS is 10.

```
%let dsid=%sysfunc(open(mydata, i));
%let rc=%sysfunc(fetchobs(&dsid, 10));
%let obsnum=%sysfunc(curobs(&dsid));
```

See Also

Functions:

- “FETCHOBS Function” on page 602
- “OPEN Function” on page 1183

CV Function

Returns the coefficient of variation.

Category: Descriptive Statistics

Syntax

CV(*argument-1*,*argument-2* <, ...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two arguments are required. The argument list can consist of a variable list, which is preceded by OF.

Example

```
data one;
   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=;
run;
```

The preceding statements produce these results:

```
x1=43.47826087
x2=26.082026548
x3=19.924242152
x4=40.953539216
```

DACCDB Function

Returns the accumulated declining balance depreciation.

Category: Financial

Syntax

DACCDB(*p*, *v*, *y*, *r*)

Required Arguments

p
is numeric, and specifies the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

r
is numeric, the rate of depreciation expressed as a decimal.

Range $r > 0$

Details

The DACCDB function returns the accumulated depreciation by using a declining balance method. Here is the formula:

$$\text{DACCDB}(p, v, y, r) = \begin{cases} 0 & p \le 0 \\ v\left(1 - \left(1 - \frac{r}{y}\right)^{int(p)}\right)\left(1 - (p - int(p))\frac{r}{y}\right) & p > 0 \end{cases}$$

Note that int(*p*) is the integer part of *p*. The *p* and *y* arguments must be expressed by using the same units of time. A double-declining balance is obtained by setting *r* equal to 2.

Example

An asset has a depreciable initial value of USD 1000 and a 15-year lifetime. Using a 200% declining balance, the depreciation throughout the first 10 years can be expressed with these statements. The value that is returned is 760.93. The first and third arguments are expressed in years.

```
data one;
   a=daccdb(10,1000,15,2);
   put a=;
run;
```

Thee preceding statements produce this result:

```
a=760.93
```

DACCDBSL Function

Returns the accumulated declining balance with conversion to a straight-line depreciation.

Category: Financial

Syntax

DACCDBSL(*p*, *v*, *y*, *r*)

Required Arguments

p
is numeric, and specifies the period for which the calculation is to be done.

v
is numeric, the depreciable initial value of the asset.

y
is an integer, the lifetime of the asset.

Range $y > 0$

r
is numeric, the rate of depreciation that is expressed as a fraction.

Range $r > 0$

Details

The DACCDBSL function returns the accumulated depreciation by using a declining balance method, with conversion to a straight-line depreciation function that is defined by the following equation:

$$\text{DACCDBSL}(p, v, y, r) = \sum_{i=1}^{p} \text{DEPDBSL}(i, v, y, r)$$

The declining balance with conversion to a straight-line depreciation chooses for each time period the method of depreciation (declining balance or straight-line on the remaining balance) that gives the larger depreciation. The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset has a depreciable initial value of $1,000 and a ten-year lifetime. Using a declining balance rate of 150%, the accumulated depreciation of that asset in its fifth year can be expressed with these statements. The value that is returned is 564.99. The first and the third arguments are expressed in years.

```
data
one;

   y5=daccdbsl(5, 1000, 10,
1.5);

   put
y5=;

run;
```

These statements produce this result:

```
y5=564.99479167
```

DACCSL Function

Returns the accumulated straight-line depreciation.

Category: Financial

Syntax

DACCSL(*p*, *v*, *y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For fractional *p*, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

Details

The DACCSL function returns the accumulated depreciation by using the straight-line method, which is given by

$$\text{DACCSL}(p, v, y) = \begin{cases} 0 & p < 0 \\ v\left(\frac{p}{y}\right) & 0 \le p \le y \\ v & p > y \end{cases}$$

The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01APR86, has a depreciable initial value of $1000 and a ten-year lifetime. The accumulated depreciation in the value of the asset through 31DEC87 can be expressed with these statements. The value that is returned is 175.00. The first and the third arguments are expressed in years.

```
data
one;

   a=daccsl(1.75, 1000,
10);

   put
a=;

run;
```

These statements produce this result:

```
a=175
```

DACCSYD Function

Returns the accumulated sum-of-years-digits depreciation.

Category: Financial

Syntax

DACCSYD(*p*, *v*, *y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

Details

The DACCSYD function returns the accumulated depreciation by using the sum-of-years-digits method. The formula is

$$\text{DACCSYD}(p, v, y) = \begin{cases} 0 & p < 0 \\ v\dfrac{int(p)\left(y - \dfrac{int(p)-1}{2}\right) + (p - int(p))(y - int(p))}{int(y)\left(y - \dfrac{int(y)-1}{2}\right) + (y - int(y))^2} & 0 \le p \le y \\ v & p > y \end{cases}$$

Note that int(*y*) indicates the integer part of *y*. The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01OCT86, has a depreciable initial value of $1,000 and a five-year lifetime. The accumulated depreciation of the asset throughout 01JAN88 can be expressed with these statements.

The value that is returned is 400.00. The first and the third arguments are expressed in years.

```
data
one;

   y2=daccsyd(15/12, 1000,
5);

   put
y2=;

run;
```

These statements produce this result:

```
y2=400
```

DACCTAB Function

Returns the accumulated depreciation from specified tables.

Category: Financial

Syntax

DACCTAB(*p*, *v*, *t1, ..., tn*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

t1, ..., tn
are numeric, and are the fractions of depreciation for each time period with $t1+t2+...tn \leq 1$.

Details

The DACCTAB function returns the accumulated depreciation by using user-specified tables. The formula for this function is

$$DACCTAB(p, v, t_1, t_2, ..., t_n) = \begin{cases} 0 & p \le 0 \\ v\left(t_1 + t_2 + ... + t_{int(p)} + (p - int(p))t_{int(p)+1}\right) & 0 < p < n \\ v & p \ge n \end{cases}$$

For a given *p*, only the arguments $t_1, t_2, \ldots, t_k$ need to be specified with `k=ceil(p)`.

Example

An asset has a depreciable initial value of $1000 and a five-year lifetime. Using a table of the annual depreciation rates of .15, .22, .21, .21, and .21 during the first, second, third, fourth, and fifth years, respectively, the accumulated depreciation throughout the third year can be expressed with these statements. The value that is returned is 580.00. The fourth rate, .21, and the fifth rate, .21, can be omitted because they are not needed in the calculation.

```
data
one;

   y3=dacctab(3,
1000, .15, .22, .21, .21, .21);

   put
y3=;

run;
```

These statements produce this result:

```
y3=580
```

DAIRY Function

Returns the derivative of the AIRY function.

Category: Mathematical

Syntax

DAIRY(*x*)

Required Argument

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 on page 1609.)

Example

```
data
one;


x=dairy(2.0);


y=dairy(-2.0);

   put x=
y=;

run;
```

These statements produce these results:

```
x=-0.053090384 y=0.6182590207
```

DATDIF Function

Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.

Category: Date and Time

Syntax

DATDIF(*start-date*, *end-date*, *basis*)

Required Arguments

start-date
specifies a SAS date value that identifies the starting date.

Tip If *start-date* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

end-date
specifies a SAS date value that identifies the ending date.

Tip If *end-date* 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)" on page 515 .

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.

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 example, 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 _null;
   sdate='16oct20'd;
   edate='16feb21'd;
   actual=datdif(sdate, edate, 'act/act');
   days360=datdif(sdate, edate, '30/360');
   put actual=;
   put days360=;
run;
```

These statements produce these results:

```
actual=123
days360=120
```

See Also

Functions:

- “YRDIF Function” on page 1576

References

Securities Industry Association 1994. *Standard Securities Calculation Methods - Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, USA: . Securities Industry Association.

DATE Function

Returns the current date as a SAS date value.

Category: Date and Time

Alias: TODAY

Interaction: If the value of the TIMEZONE= system option is set to a time zone name or time zone ID, the date and time values that are returned for this function are determined by the time zone.

See: “TODAY Function” on page 1457

Syntax

DATE()

Details

The DATE function produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

Examples

Example 1: Determining a Date Value for an America/Los_Angeles Time Zone

This example shows how the DATE function returns a date value based on the value of the TIMEZONE= system option.

```
option timezone='America/Los_Angeles';
data _null_;
   d1=date();
   put d1=nldate.;
run;
```

SAS writes the following results to the log:

```
d1=February 03, 2021
```

Example 2: Determining a Date Value for an Asia/Shanghai Time Zone

This example shows how the DATE function returns a date value based on the value of the TIMEZONE= system option.

```
option timezone='Asia/Shanghai';
data _null_;
   d2=date();
   put d2=nldate.;
run;
```

SAS writes these results to the log:

```
d2=February 03, 2021
```

DATEJUL Function

Converts a Julian date to a SAS date value.

Category: Date and Time

See: "About SAS Date, Time, and Datetime Values" in *SAS Formats and Informats for SAS Viya Workbench*

Syntax

DATEJUL(*julian-date*)

Required Argument

julian-date
specifies a SAS numeric expression that represents a Julian date. A Julian date in SAS is a date in the form *yyddd* or *yyyyddd*, where *yy* or *yyyy* is a two-digit or four-digit integer 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).

Example

```
Data one;

Xstart=datejul(19365);

   put Xstart / Xstart
date9.;




Xend=datejul(2021001);

   put Xend / Xend
date9.;

run;
```

SAS writes these results to the log:

```
21914
31DEC2019
22281
01JAN2021
```

See Also

Functions:

- "JULDATE Function" on page 1030

DATEPART Function

Extracts the date from a SAS datetime value.

Category: Date and Time

Syntax

DATEPART(*datetime*)

Required Argument

datetime
specifies a SAS expression that represents a SAS datetime value.

Example

```
data
one;


conn='03feb21:8:45'dt;


servdate=datepart(conn);

   put servdate=
worddate.;

run;
```

SAS writes these results to the log:

```
servdate=February 3, 2021
```

See Also

Functions:

- “DATETIME Function” on page 520
- “TIMEPART Function” on page 1451

DATETIME Function

Returns the current date and time of day as a SAS datetime value.

Category: Date and Time

Interaction: If the value of the TIMEZONE= system option is set to a time zone name or time zone ID, the date and time values that are returned for this function are determined by the time zone.

Syntax

DATETIME()

Examples

Example 1: Returning the Number of Seconds with the DATETIME Function

This example returns a SAS value that represents the number of seconds between January 1, 1960, and the current time.

```
data
one;


when=datetime();

   put
when=;

run;
```

SAS writes these results to the log:

```
when=1928418844.4
```

Example 2: Determining a Datetime Value for an America/Los_Angeles Time Zone

This example shows how the DATETIME function returns datetime values based on the value of the TIMEZONE= system option.

```
option timezone='America/Los_Angeles';
data _null_;
   dt1=datetime();
   put dt1=nldatm.;
run;
```

SAS writes these results to the log:

```
dt1=09Feb2021:11:55:30
```

Example 3: Determining a Datetime Value for a Europe/Berlin Time Zone

This example shows how the DATETIME function returns datetime values based on the value of the TIMEZONE= system option.

```
option timezone='Europe/Berlin';
data _null_;
   dt2=datetime();
   put dt2=nldatm.;
run;
```

SAS writes these results to the log:

```
dt2=09Feb2021:12:07:36
```

See Also

Functions:

- “DATE Function” on page 516
- “TIME Function” on page 1450
- “TODAY Function” on page 1457

DAY Function

Returns the day of the month from a SAS date value.

Category: Date and Time

Syntax

DAY(*date*)

Required Argument

date
specifies a SAS expression that represents a SAS date value.

Details

The DAY function produces an integer from 1 to 31 that represents the day of the month.

Example

```
data one;
   now='09feb21'd;
   d=day(now);
   put d=;
run;
```

These statements produce this result:

```
d=9
```

See Also

Functions:

- “MONTH Function” on page 1120
- “YEAR Function” on page 1573

DCLOSE Function

Closes a directory that was opened by the DOPEN function.

Category: External Files

Syntax

DCLOSE(*directory-id*)

Required Argument

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

Details

DCLOSE returns 0 if the operation was successful, ≠0 if it was not successful. The DCLOSE function closes a directory that was previously opened by the DOPEN function. DCLOSE also closes any open members.

Note: All directories or members opened within a DATA step are closed automatically when the DATA step ends.

Examples

Example 1: Using DCLOSE to Close a Directory

This example opens the directory to which the fileref MYDIR has previously been assigned, returns the number of members, and then closes the directory:

```
%macro memnum(filrf, path);
%let rc=%sysfunc(filename(filrf, &path));
%if %sysfunc(fileref(&filrf))=0 %then
   %do;
         /* Open the directory. */
      %let did=%sysfunc(dopen(&filrf));
      %put did=&did;
         /* Get the member count. */
      %let memcount=%sysfunc(dnum(&did));
      %put &memcount members in &filrf.;
         /* Close the directory. */
      %let rc=%sysfunc(dclose(&did));
   %end;
%else %put Invalid FILEREF;
%mend;
%memnum(MYDIR, physical-filename)
```

Example 2: Using DCLOSE within a DATA Step

This example uses the DCLOSE function within a DATA step:

```
%let filrf=MYDIR;
```

```
data _null_;
  rc=filename("&filrf", "physical-filename");
  if fileref("&filrf")=0 then
     do;
           /* Open the directory. */
        did=dopen("&filrf");
           /* Get the member count. */
        memcount=dnum(did);
        put memcount "members in &filrf";
           /* Close the directory. */
        rc=dclose(did);
     end;
  else put "Invalid FILEREF";
run;
```

See Also

Functions:

- “DOPEN Function” on page 556
- “FCLOSE Function” on page 591
- “FOPEN Function” on page 720
- “MOPEN Function” on page 1121

DCREATE Function

Returns the complete pathname of a new, external directory.

Category: External Files

Syntax

DCREATE(*directory-name* <, *parent-directory*>)

Required Argument

directory-name
is a character constant, variable, or expression that specifies the name of the directory to create. This value cannot include a pathname.

Optional Argument

parent-directory
is a character constant, variable, or expression that contains the complete pathname of the directory in which to create the new directory. If you do not

supply a value for *parent-directory*, then the current directory is the parent directory.

Details

The DCREATE function enables you to create a directory in your operating environment. If the directory cannot be created, then DCREATE returns an empty string.

Example

```
data
one;

/* To create a new directory in the UNIX operating environment, using
the name
   that is stored in the variable DirectoryName, follow this form:
*/



 NewDirectory=dcreate(DirectoryName, '/local/u/
abcdef/');



run;
```

DEPDB Function

Returns the declining balance depreciation.

Category: Financial

Syntax

DEPDB(*p*, *v*, *y*, *r*)

Required Arguments

p

is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

r
is numeric, the rate of depreciation that is expressed as a fraction.

Range $r \geq 0$

Details

The DEPDB function returns the depreciation by using the declining balance method, which is given by

$$\begin{aligned}\text{DEPDB}(p, v, y, r) &= \text{DACCDB}(p, v, y, r) \\ &\quad -\text{DACCDB}(p-1, v, y, r)\end{aligned}$$

The *p* and *y* arguments must be expressed by using the same units of time. A double-declining balance is obtained by setting *r* equal to 2.

Example

An asset has an initial value of $1,000 and a fifteen-year lifetime. Using a declining balance rate of 200%, the depreciation of the value of the asset for the 10th year can be expressed with these statements.

The value that is returned is 36.78. The first and the third arguments are expressed in years.

```
data
one;

   y10=depdb(10, 1000, 15,
2);

   put
y10=;

run;
```

These statements produce this result:

```
y10=36.779648624
```

DEPDBSL Function

Returns the declining balance with conversion to a straight-line depreciation.

Category: Financial

Syntax

DEPDBSL(*p*, *v*, *y*, *r*)

Required Arguments

p
is an integer, the period for which the calculation is to be done.

v
is numeric, the depreciable initial value of the asset.

y
is an integer, the lifetime of the asset.

Range $y > 0$

r
is numeric, the rate of depreciation that is expressed as a fraction.

Range $r \geq 0$

Details

The DEPDBSL function returns the depreciation by using the declining balance method with conversion to a straight-line depreciation, which is given by the following equation:

$$\text{DEPDBSL}(p, v, y, r) = \begin{cases} 0 & p \leq 0 \\ v\frac{r}{y}\left(1 - \frac{r}{y}\right)^{p-1} & 0 < p \leq t \\ \frac{v\left(1 - \frac{r}{y}\right)^{t}}{(y - t)} & t < p \leq y \\ 0 & p > y \end{cases}$$

The following relationship applies to the preceding equation:

$$t = int\left(y - \frac{y}{r} + 1\right)$$

int( ) denotes the integer part of a numeric argument.

The *p* and *y* arguments must be expressed by using the same units of time. The declining balance that changes to a straight-line depreciation chooses for each time period the method of depreciation (declining balance or straight-line on the remaining balance) that gives the larger depreciation.

Example

An asset has a depreciable initial value of $1,000 and a ten-year lifetime. Using a declining balance rate of 150%, the depreciation of the value of the asset in the fifth year can be expressed with these statements. The value that is returned is 87.001041667. The first and the third arguments are expressed in years.

```
data
one;

   y5=depdbsl(5, 1000, 10,
1.5);

   put
y5=;

run;
```

These statements produce this result:

```
y5=87.001041667
```

DEPSL Function

Returns the straight-line depreciation.

Category: Financial

Syntax

DEPSL(*p*, *v*, *y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For fractional *p*, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

Details

The DEPSL function returns the straight-line depreciation, which is given by

$$\begin{aligned}\text{DEPSL}(p, v, y) &= \text{DACCSL}(p, v, y) \\ &\quad -\text{DACCSL}(p-1, v, y)\end{aligned}$$

The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01APR86, has a depreciable initial value of $1,000 and a ten-year lifetime. The depreciation in the value of the asset for the year 1986 can be expressed with these statements.

The value that is returned is 75.00. The first and the third arguments are expressed in years.

```
data
one;

   d=depsl(9/12, 1000,
10);

   put
d=;

run;
```

These statements produce this result:

```
d=75
```

DEPSYD Function

Returns the sum-of-years-digits depreciation.

Category: Financial

Syntax

DEPSYD(*p*, *v*, *y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset in number of depreciation periods.

Range $y > 0$

Details

The DEPSYD function returns the sum-of-years-digits depreciation, which is given by

$$\begin{aligned}\text{DEPSYD}(p, v, y) &= \text{DACCSYD}(p, v, y) \\ &\quad -\text{DACCSYD}(p-1, v, y)\end{aligned}$$

The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01OCT86, has a depreciable initial value of $1,000 and a five-year lifetime. The depreciations in the value of the asset for the years 1986 and 1987 can be expressed with these statements:

The values that are returned are 83.33 and 316.67, respectively. The first and the third arguments are expressed in years.

```
data
one;

   y1=depsyd(3/12, 1000,
5);

   y2=depsyd(15/12, 1000,
5);

   put y1=
y2=;
```

```
run;
```

These statements produce these results:

```
y1=83.333333333 y2=316.66666667
```

DEPTAB Function

Returns the depreciation from specified tables.

Category: Financial

Syntax

DEPTAB(*p*, *v*, *t1, ..., tn*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

t1, t2, ..., tn
are numeric, the fractions of depreciation for each time period with *t1*+*t2*+...*tn* ≤ 1.

Details

The DEPTAB function returns the depreciation by using specified tables. The formula is

$$DEPTAB(p, v, t_1, t_2, ..., t_n) = DACCTAB(p, v, t_1, t_2, ..., t_n) - DACCTAB(p-1, v, t_1, t_2, ..., t_n)$$

For a given *p*, only the arguments $t_1, t_2, \ldots, t_k$ need to be specified with `k=ceil(p)`.

Example

An asset has a depreciable initial value of $1,000 and a five-year lifetime. Using a table of the annual depreciation rates of .15, .22, .21, .21, and .21 during the first, second, third, fourth, and fifth years, respectively, the depreciation in the third year can be expressed with these statements. The value that is returned is 210.00. The fourth rate, .21, and the fifth rate, .21, can be omitted because they are not needed in the calculation.

```
data
one;

   y3=deptab(3,
1000, .15, .22, .21, .21, .21);

   put
y3=;

run;
```

These statements produce this result:

```
y3=210
```

DEQUOTE Function

Removes matching quotation marks from a character string that begins with a quotation mark, and deletes all characters to the right of the closing quotation mark.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

DEQUOTE(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the DEQUOTE function returns a value to a variable that has not been previously assigned a length, then that variable is given the length of the argument.

The Basics

The value that is returned by the DEQUOTE function is determined as follows:

- If the first character of *string* is not a single or double quotation mark, DEQUOTE returns *string* unchanged.
- If the first two characters of *string* are both single quotation marks or both double quotation marks, and the third character is not the same type of quotation mark, then DEQUOTE returns a result with a length of zero.
- If the first character of *string* is a single quotation mark, the DEQUOTE function removes that single quotation mark from the result. DEQUOTE then scans *string* from left to right, looking for more single quotation marks. Each pair of consecutive, single quotation marks is reduced to one single quotation mark. The first single quotation mark that does not have an ending quotation mark in *string* is removed and all characters to the right of that quotation mark are also removed.
- If the first character of *string* is a double quotation mark, the DEQUOTE function removes that double quotation mark from the result. DEQUOTE then scans *string* from left to right, looking for more double quotation marks. Each pair of consecutive, double quotation marks is reduced to one double quotation mark. The first double quotation mark that does not have an ending quotation mark in *string* is removed and all characters to the right of that quotation mark are also removed.

Note: If *string* is a constant enclosed in quotation marks, those quotation marks are not part of the value of *string*. Therefore, you do not need to use DEQUOTE to remove the quotation marks that denote a constant.

Example

This example demonstrates how quotation marks in the DEQUOTE function are handled within a DATA step.

```
data test;
   input string $80.;
   result=dequote(string);
   datalines;
No quotation marks, no change
No "leading" quotation marks, no change
"Matching double quotation marks are removed"
```

```
'Matching single quotation marks are removed'
"Paired ""quotation marks"" are reduced"
'Paired '' quotation marks '' are reduced'
"Single 'quotation marks' inside '' double'' quotation marks are
unchanged"
'Double "quotation marks" inside ""single"" quotation marks are
unchanged'
"No matching quotation mark, no problem
Don't remove this apostrophe
"Text after the matching quotation mark" is "deleted"
;
proc print noobs;
title 'Input Strings and Output Results from DEQUOTE';
run;
```

Output 3.31 *Working with Quotation Marks in the DEQUOTE Function*

Input Strings and Output Results from DEQUOTE

string	result
No quotation marks, no change	No quotation marks, no change
No "leading" quotation marks, no change	No "leading" quotation marks, no change
"Matching double quotation marks are removed"	Matching double quotation marks are removed
'Matching single quotation marks are removed'	Matching single quotation marks are removed
"Paired ""quotation marks"" are reduced"	Paired "quotation marks" are reduced
'Paired '' quotation marks '' are reduced'	Paired ' quotation marks ' are reduced
"Single 'quotation marks' inside '' double'' quotation marks	Single 'quotation marks' inside '' double'' quotation marks
'Double "quotation marks" inside ""single"" quotation marks	Double "quotation marks" inside ""single"" quotation marks
"No matching quotation mark, no problem	No matching quotation mark, no problem
Don't remove this apostrophe	Don't remove this apostrophe
"Text after the matching quotation mark" is "deleted"	Text after the matching quotation mark

DEVIANCE Function

Returns the deviance based on a probability distribution.

Category: Mathematical

Syntax

DEVIANCE(*distribution*, *variable*, *shape-parameters* <, *ε*>)

Required Arguments

distribution
is a character constant, variable, or expression that identifies the distribution.

Valid distributions are listed in the following table:

Distribution	Argument
Bernoulli	`'BERNOULLI' | 'BERN'`
Binomial	`'BINOMIAL' | 'BINO'`
Gamma	`'GAMMA'`
Inverse Gauss (Wald)	`'IGAUSS' | 'WALD'`
Normal	`'NORMAL' | 'GAUSSIAN'`
Poisson	`'POISSON' | 'POIS'`

variable
is a numeric constant, variable, or expression.

shape-parameter
are one or more distribution-specific numeric parameters that characterize the shape of the distribution.

Optional Argument

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 $1 - \varepsilon$.

If eps is not specified, a value of 1e-12 is used.

If eps is less than 1e-12, a value of 1e-12 is used.

If eps is greater than 0.01, a value of 0.01 is used.

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.

$$\text{DEVIANCE}('BERN', variable, p, \varepsilon) = \begin{cases} -2\log(1-p) & x = 0 \\ -2\log(p) & x = 1 \\ . & otherwise \end{cases}$$

This example demonstrates the functionality of the Bernoulli distribution.

```
data Bern;
    var=1;
    p=0.6;
    d=deviance("Bern", var, p);
    put d=;
run;
```

The preceding statements produce this result:

```
d=1.0216512475
```

The Binomial Distribution

DEVIANCE('BINO', *variable*, *μ*, *n* <, *ε*>)

Arguments

variable
is a numeric random variable that contains the number of successes.

Range $0 \le variable \le 1$

μ
is a numeric mean parameter.

Range $n\varepsilon \le \mu \le n(1-\varepsilon)$

n
is an integer 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 $n\varepsilon$. Any value of μ in the interval $n(1-\varepsilon) \le \mu \le n$ is replaced by $n(1-\varepsilon)$.

If eps is not specified, a value of 1e−12 is used.

If eps is less than 1e−12, a value of 1e−12 is used.

If eps is greater than 0.01, a value of 0.01 is used.

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.

$$\text{DEVIANCE}('BINO', x, \mu, n) = \begin{cases} . & x < 0 \\ 2\left(x\log\left(\frac{x}{\mu}\right) + (n-x)\log\left(\frac{n-x}{n-\mu}\right)\right) & 0 \le x \le n \\ . & x > n \end{cases}$$

This example demonstrates the functionality of the Binomial distribution.

```
data Binom;
    p=0.6; n=50; mu=n*p;
    d=deviance("Binom", 27, mu, n);
    put d=;
run;
```

The preceeding statements produce this result:

```
d=0.7395815037
```

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.

If eps is not specified, a value of 1e−12 is used.

If eps is less than 1e−12, a value of 1e−12 is used.

If eps is greater than 0.01, a value of 0.01 is used.

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:

$$\text{DEVIANCE}('GAMMA', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(-\log\left(\frac{x}{\mu}\right) + \frac{x-\mu}{\mu}\right) & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

This example demonstrates the functionality of the Gamma distribution.

```
data Gamma;
    var=1;
    p=0.6;
    d=deviance("Gamma", var, p);
    put d=;
run;
```

The preceeding statements produce this result:

```
d=0.3116820858
```

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.

If eps is not specified, a value of 1e−12 is used.

If eps is less than 1e−12, a value of 1e−12 is used.

If eps is greater than 0.01, a value of 0.01 is used.

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:

$$\text{DEVIANCE}('IGAUSS', x, \mu) = \begin{cases} . & x < 0 \\ \frac{(x-\mu)^2}{\mu^2 x} & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

This example demonstrates the functionality of the Wald distribution.

```
data Wald;
    var=1.2;
    mu=0.6;
    d=deviance("Wald", var, mu);
    put d=;
run;
```

The preceeding statements produce this result:

```
d=0.8333333333
```

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:

$$\text{DEVIANCE}('NORMAL', x, \mu) = (x - \mu)^2$$

This example demonstrates the functionality of the Normal distribution.

```
data Normal;
    var=10.1;
    mu=10;
    d=deviance("Normal", var, mu);
    put d=;
run;
```

The preceeding statements produce this result:

```
d=0.01
```

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.

If eps is not specified, a value of 1e−12 is used.

If eps is less than 1e−12, a value of 1e−12 is used.

If eps is greater than 0.01, a value of 0.01 is used.

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:

$$\text{DEVIANCE}('POISSON', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(x\log\left(\frac{x}{\mu}\right) - (x-\mu)\right) & x \geq 0, \mu \geq \varepsilon \end{cases}$$

This example demonstrates the functionality of the Poisson distribution.

```
data Poisson;
    var=1.4;
    mu=0.7;
    d=deviance("Poisson", var, mu);
    put d=;
run;
```

The preceeding statements produce this result:

```
d=0.5408121056
```

DHMS Function

Returns a SAS datetime value from date, hour, minute, and second values.

Category: Date and Time

Syntax

DHMS(*date*, *hour*, *minute*, *second*)

Required Arguments

date
specifies a SAS expression that represents a SAS date value.

hour
is numeric.

minute
is numeric.

second
is numeric.

Details

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

You can also use the DHMS function to combine a SAS date value with a SAS time value to produce a SAS time value. Because a SAS time value is stored in seconds, you can specify 0 for the hour variable and 0 for the minute variable to return the correct value. Here is the syntax:

```
DHMS(SAS date, 0, 0, SAS time).
```

Example

```
data
one;

   dtid=dhms('09feb21'd, 15, 30,
15);

   put
dtid;

   put dtid
datetime.;



   dtid2=dhms('09feb21'd, 15, 30,
61);

   put
dtid2;

   put dtid2
datetime.;



   dtid3=dhms('09feb21'd, 15, .5,
15);

   put
dtid3;
```

```
   put dtid3
datetime.;

run;
```

SAS writes the following results to the log:

```
1928503815
09FEB21:15:30:15
1928503861
09FEB21:15:31:01
1928502045
09FEB21:15:00:45
```

The following SAS statements show how to combine a SAS date value with a SAS time value into a SAS datetime value.

```
data
one;


day=date();


time=time();

   sasdt=dhms(day, 0, 0,
time);

   put sasdt
datetime.;

run;
```

SAS writes the following results to the log:

```
09FEB21:12:37:38
```

See Also

Functions:

- “HMS Function” on page 910

DIF Function

Returns differences between an argument and its *n*th lag.

Category: Special

Syntax

DIF<*n*> (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Optional Argument

n
specifies the number of lags.

Details

The DIF functions, DIF1, DIF2, ..., DIF*n*, return the first differences between the argument and its *n*th lag. DIF1 can also be written as DIF. DIF*n* is defined as DIF*n*(*x*)=*x*-LAG*n*(*x*).

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 second difference DIF(DIF(X)).

Example

This example demonstrates the difference between the LAG and DIF functions.

```
data two;
   input X @@;
   Z=lag(x);
   D=dif(x);
   datalines;
1 2 6 4 7
;
proc print data=two;
run;
```

Output 3.32 *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

See Also

Functions:

- "LAG Function" on page 1033

DIGAMMA Function

Returns the value of the digamma function.

Category: Mathematical

Syntax

DIGAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Restriction Nonpositive integers are invalid.

Details

The DIGAMMA function returns the ratio that is given by

$$\Psi(x) = \Gamma'(x)/\Gamma(x)$$

where $\Gamma(.)$ and $\Gamma'(.)$ denote the Gamma function and its derivative, respectively. For *argument*>0, the DIGAMMA function is the derivative of the LGAMMA function.

Example

```
data
one;


x=digamma(1.0);

   put
x=;

run;
```

These statements produce this result:

```
x=-0.577215665
```

DIM Function

Returns the number of elements in an array.

Category: Array

Syntax

DIM<*n*> (*array-name*)

DIM(*array-name*, *bound-n*)

Required Arguments

array-name
specifies the name of an array that was previously defined in the same DATA step. This argument cannot be a constant, variable, or expression.

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. Use *bound-n* only when *n* is not specified.

Optional Argument

n

specifies the dimension, in a multidimensional array, for which you want to know the number of elements. If no *n* value is specified, the DIM function returns the number of elements in the first dimension of the array.

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 when the lower bound of the dimension is 1. Use DIM in array processing to avoid changing the upper bound of an iterative DO group each time you change the number of array elements.

Comparisons

- DIM always returns a total count of the number of elements in an array dimension.
- HBOUND returns the literal value of the upper bound of an array dimension.

Note: This distinction is important when the lower bound of an array dimension has a value other than 1 and the upper bound has a value other than the total number of elements in the array dimension.

Examples

Example 1: One-dimensional Array

In this example, DIM returns a value of 5. Therefore, SAS repeats the statements in the DO loop five times.

```
data
mydata;

input weight sex height state
city;

datalines;

1 2 3 4
5
```

```
6 7 8 9
0

;



data
one;

set
mydata;









   array big{5} weight sex height state
city;

   do i=1 to dim(big);
   put
big(i);


end;

run;
```

These statements produce these results:

```
1
2
3
4
5
6
7
8
9
0
```

Example 2: Multidimensional Array

This example shows two ways of specifying the DIM function for multidimensional arrays. Both methods return the same value for DIM, as shown in the table that follows the SAS code example.

```
data
one;

   array mult{5, 10, 2} mult1-
mult100;
```

```
v1=DIM(MULT);


v2=DIM(MULT,1);

   put v1=
v2=;

run;
```

These statements produce these results:

```
v1=5 v2=5
```

Syntax	Alternative Syntax	Value
DIM(MULT)	DIM(MULT,1)	5
DIM2(MULT)	DIM(MULT,2)	10
DIM3(MULT)	DIM(MULT,3)	2

See Also

Functions:

- “HBOUND Function” on page 907
- “LBOUND Function” on page 1044

Statements:

- “ARRAY Statement” in *SAS DATA Step Statements for SAS Viya Workbench*
- “Array Reference Statement” in *SAS DATA Step Statements for SAS Viya Workbench*

Other References:

- “Using Arrays” in *SAS Language Programmer’s Guide for SAS Viya Workbench*

DINFO Function

Returns information about a directory.

Category: External Files

Operating environment: For the Linux OS, directory pathname, owner, group, permissions, and time last modified information items are available.

Syntax

DINFO(*directory-id*, *information-item*)

Required Arguments

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

information-item
is a character constant, variable, or expression that specifies the information item to be retrieved. DINFO returns a blank if the value of the *information-item* argument is invalid. The information available varies according to the operating environment.

Details

Overview

Directories that are opened with the DOPEN function are identified by a *directory-id* value. Use the DOPTNAME function to determine the names of the available system-dependent directory information items. Use the DOPTNUM function to determine the number of directory information items that are available.

If *directory-id* points to a list of concatenated directories, then the directory is the list of concatenated directory names.

Linux Specifics: The following information items are available: directory pathname, which is the pathname of *directory-id*, owner, group, permissions, and the time last modified.

Examples

Example 1: Using DINFO to Return Information about a Directory

This example opens the directory MYDIR, determines the number of directory information items available, and retrieves the value of the last one:

```
%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf, "physical-name"));
%let did=%sysfunc(dopen(&filrf));
%let numopts=%sysfunc(doptnum(&did));
```

```
%let foption=%sysfunc(doptname(&did, &numopts));
%let charval=%sysfunc(dinfo(&did, &foption));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DINFO within a DATA Step

This example creates a data set that contains the name and value of each directory information item:

```
data diropts;
   length optname $ 32 optval $ 40;
   rc=filename("mydir", "physical-name");
   put "rc = 0 if the directory exists: " rc=;
   did=dopen("mydir");
   numopts=doptnum(did);
   do i=1 to numopts;
      optname=doptname(did, i);
      put i= optname=;
      optval=dinfo(did, optname);
      put optval=;
   end;
run;
```

Output 3.33 *Sample SAS Log for Linux*

```
65   data diropts;
66      length optname $ 32 optval $ 40;
67      rc=filename("mydir", "/u");
68      put "rc = 0 if the directory exists: " rc=;
69      did=dopen("mydir");
70      numopts=doptnum(did);
71      do i=1 to numopts;
72         optname=doptname(did, i);
73         put i= optname=;
74         optval=dinfo(did, optname);
75         put optval=;
76      end;
77   run;

rc = 0 if the directory exists: rc=0
i=1 optname=Directory
optval=/u
i=2 optname=Owner Name
optval=root
i=3 optname=Group Name
optval=root
i=4 optname=Access Permission
optval=drwxr-xr-x
i=5 optname=Last Modified
optval=02Dec2014:23:07:22
NOTE: The data set WORK.DIROPTS has 1 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time           0.08 seconds
      cpu time            0.05 seconds
```

See Also

Functions:

- “DOPEN Function” on page 556
- “DOPTNAME Function” on page 558
- “DOPTNUM Function” on page 560
- “FINFO Function” on page 699
- “FOPTNAME Function” on page 723
- “FOPTNUM Function” on page 726

DIVIDE Function

Returns the result of a division that handles special missing values for ODS output.

Category: Mathematical

Syntax

DIVIDE(*x*, *y*)

Required Arguments

x
is a numeric constant, variable, or expression that represents the numerator.

y
is a numeric constant, variable, or expression that represents the denominator.

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 3.1 *Values That Are Returned by the DIVIDE Function*

		x						
		positive	zero	negative	.I	.M	._	other
y	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 example shows the results of using the DIVIDE function.

```
data _null_;
   a=divide(1, 0);
   put +3 a='(infinity)';
   b=divide(2, .I);
   put +3 b=;
   c=divide(.I, -1);
   put +3 c='(minus infinity)';
   d=divide(constant('big'), constant('small'));
   put +3 d='(infinity because of overflow)';
run;
```

SAS writes the following output to the log:

```
a=I (infinity)
b=0
c=M (minus infinity)
d=I (infinity because of overflow)
```

DLGCDIR Function

Sets the working directory.

Note: DLGCDIR is supported on Linux.

Syntax

DLGCDIR(<*working_directory*>)

Optional Argument

working_directory
specifies the working directory.

Details

You can select a new working directory and information about the directory with the DLGCDIR function:

- You can change the current working directory by specifying a directory with the DLGCDIR function: `rc=dlgcdir("c:\");`
- If no parameters are specified, DLGCDIR reports the current working directory to the log: `rc=dlgcdir();`

Example

Here are DLGCDIR examples. RC=0 specifies a return code with no errors.

```
data _null_;
  rc=dlgcdir("c:\");
  put rc=;
run;
```

```
NOTE: The current working directory is now "c:\".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.40 seconds
      cpu time            0.12 seconds
```

```
/* Test for the current directory. */

data _null_;
  rc=dlgcdir();
  put rc=;
run;
```

```
NOTE: The current working directory is "c:\".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time
```

```
data _null_;
  rc=dlgcdir('%userprofile%');
```

```
  put rc=;
run;
```

```
NOTE: The current working directory is now "C:\Users\userid".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

```
/* Test for the current directory. */
data _null_;
  rc=dlgcdir();
  put rc=;
run;
```

```
NOTE: The current working directory is "C:\Users\userid".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

```
data _null_;
  rc=dlgcdir("C:\Users\userid\websas04\Tools");
  put rc=;
run;
```

```
NOTE: The current working directory is now "C:\Users\userid\websas04\Tools".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

```
/* Test for the current directory. */
data _null_;
  rc=dlgcdir();
  put rc=;
run;
```

```
NOTE: The current working directory is now "C:\Users\userid\websas04\Tools".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

```
data _null_;
  rc=dlgcdir("u:\dev94m4");
  put rc=;
run;
```

```
NOTE: The current working directory is now "u:\dev94m4".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

```
/* Test for the current directory. */

data _null_;
  rc=dlgcdir();
  put rc=;
run;
```

```
NOTE: The current working directory is "u:\dev94m4".
rc=0
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

DNUM Function

Returns the number of members in a directory.

Category: External Files

Syntax

DNUM(*directory-id*)

Required Argument

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

Details

You can use DNUM to determine the highest possible member number that can be passed to DREAD.

Examples

Example 1: Using DNUM to Return the Number of Members

This example opens the directory MYDIR, determines the number of members, and closes the directory:

```
%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf, physical-name));
%let did=%sysfunc(dopen(&filrf));
```

```
%let memcount=%sysfunc(dnum(&did));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DNUM within a DATA Step

This example creates a DATA step that returns the number of members in a directory called MYDIR:

```
data _null_;
   rc=filename("mydir", "physical-name");
   did=dopen("mydir");
   memcount=dnum(did);
   rc=dclose(did);
run;
```

See Also

Functions:

- "DOPEN Function" on page 556
- "DREAD Function" on page 569

DOPEN Function

Opens a directory, and returns a directory identifier value.

Category: External Files

Restriction: You must associate a fileref with the directory before calling DOPEN.

See: "FILENAME Function" on page 611

Syntax

DOPEN(*fileref*)

Required Argument

fileref
is a character constant, variable, or expression that specifies the fileref assigned to the directory. In a DATA step, *fileref* can be a character expression, a character string, or a DATA step variable whose value contains the fileref. If the function is used in a DATA Step, then *fileref* must be enclosed in quotation marks. If the function is used in macro code, then *fileref* must not be enclosed in quotation marks. In macro code, *fileref* can be any expression.

Details

DOPEN opens a directory and returns a directory identifier value (a number greater than 0) that is used to identify the open directory in other SAS external file access functions. If the directory cannot be opened, DOPEN returns 0, and you can obtain the error message by calling the SYSMSG function. The directory to be opened must be identified by a fileref. You can assign filerefs using the FILENAME statement or the FILENAME external file access function. Under some operating environments, you can also assign filerefs using system commands.

If you call the DOPEN function from a macro, then the result of the call is valid only when the result is passed to functions in a macro. If you call the DOPEN function from the DATA step, then the result is valid only when the result is passed to functions in the same DATA step.

Operating Environment Information: The term *directory* that is used in the description of this function and related SAS external file access functions refers to an aggregate grouping of files managed by the operating environment. Different operating environments identify such groupings with different names, such as directory, subdirectory, folder, MACLIB, or partitioned data set.

Examples

Example 1: Using DOPEN to Open a Directory

This example assigns the fileref MYDIR to a directory. It uses DOPEN to open the directory. DOPTNUM determines the number of system-dependent directory information items available, and DCLOSE closes the directory:

```
%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf, physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DOPEN within a DATA Step

This example opens a directory for processing within a DATA step.

```
data _null_;
   drop rc did;
   rc=filename("mydir", "physical-name");
   did=dopen("mydir");
   if did > 0 then do;
/*...more SAS statements... */
   end;
   else do;
      msg=sysmsg();
      put msg;
   end;
run;
```

See Also

Functions:

- “DCLOSE Function” on page 522
- “DOPTNUM Function” on page 560
- “FOPEN Function” on page 720
- “MOPEN Function” on page 1121
- “SYSMSG Function” on page 1437

DOPTNAME Function

Returns directory attribute information.

Category: External Files

Operating environment: For the Linux OS, directory pathname, owner, group, permissions, and time last modified information items are available.

Syntax

DOPTNAME(*directory-id*, *nval*)

Required Arguments

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

nval
is a numeric constant, variable, or expression that specifies the sequence number of the information item.

Details

The DOPTNAME function returns the name of the specified information item number for a directory that was previously opened with the DOPEN function. If *directory-id* points to a list of concatenated directories, then Directory is the list of concatenated directory names.

Linux Specifics: The following information items are available: directory pathname, which is the pathname of *directory-id*, owner, group, permissions, and time last modified.

Examples

Example 1: Using DOPTNAME to Retrieve Directory Attribute Information

This example opens the directory with the fileref MYDIR, retrieves all system-dependent directory information items, writes them to the SAS log, and closes the directory:

```
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf, physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%do j=1 %to &infocnt;
   %let opt=%sysfunc(doptname(&did, &j));
   %put Directory information=&opt;
%end;
%let rc=%sysfunc(dclose(&did));
```

```
   %macro
   test;

    %let
   filrf=mydir;

    %let rc=%sysfunc(filename(filrf, physical-
   name));

    %let did=
   %sysfunc(dopen(&filrf));

    %let infocnt=
   %sysfunc(doptnum(&did));

     %do j=1 %to
   &infocnt;

      %let opt=%sysfunc(doptname(&did,
   &j));

      %put Directory
   information=&opt;


   %end;

    %let rc=
   %sysfunc(dclose(&did));

   %mend
   test;



   %test
```

Example 2: Using DOPTNAME within a DATA Step

This example creates a data set that contains the name and value of each directory information item:

```
data diropts;
   length optname $ 12 optval $ 40;
   keep optname optval;
   rc=filename("mydir", "physical-name");
   did=dopen("mydir");
   numopts=doptnum(did);
   do i=1 to numopts;
      optname=doptname(did, i);
      optval=dinfo(did, optname);
      output;
   end;
   run;
```

See Also

Functions:

- “DINFO Function” on page 548
- “DOPEN Function” on page 556
- “DOPTNUM Function” on page 560

DOPTNUM Function

Returns the number of information items that are available for a directory.

Category: External Files

Operating environment: For the Linux OS, directory pathname, owner, group, permissions, and time last modified information items are available.

Syntax

DOPTNUM(*directory-id*)

Required Argument

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

Details

Linux Specifics: The following information items are available for a directory: directory pathname, which is the pathname of *directory-id*, owner, group, permissions, and time last modified. Therefore, this functions returns a value of 5.

Examples

Example 1: Retrieving the Number of Information Items

This example retrieves the number of system-dependent directory information items that are available for the directory MYDIR and closes the directory:

```
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf, physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DOPTNUM within a DATA Step

This example creates a data set that retrieves the number of system-dependent information items that are available for the MYDIR directory:

```
data _null_;
   rc=filename("mydir", "physical-name");
   did=dopen("mydir");
   infocnt=doptnum(did);
   rc=dclose(did);
run;
```

See Also

Functions:

- “DINFO Function” on page 548
- “DOPEN Function” on page 556
- “DOPTNAME Function” on page 558

DOSUBL Function

Enables the immediate execution of SAS code after a text string is passed.

Category: Macro

Syntax

DOSUBL(*x*)

Required Argument

x
specifies a text string.

Details

The DOSUBL function enables the immediate execution of SAS code after a text string is passed. The function imports macro variables from the calling environment, and macro variables that are created or updated during the execution of the submitted code are exported back to the calling environment.

DOSUBL should be used in a DATA step. The function can also be used with %SYSFUNC outside a step boundary.

DOSUBL executes code in a different SAS executive (known as a side session). Here are the results of using DOSUBL with the following language elements:

CALL EXECUTE
When you use a macro to set the value of a global macro variable, and then you invoke the macro with the DOSUBL function, the macro is executed immediately. However, when you use a DATA step to set the value of a global macro variable, and then you use the CALL EXECUTE routine to call the macro, the DATA step code executes after the current DATA step completes. Example 1 demonstrates execution of the DATA step code.

SYSINDEX
During DOSUBL execution, SYSINDEX is reset in the side session and its value remains unchanged in the calling environment.

LIBNAMES and FILENAMES
LIBNAMES and FILENAMES that are set in the side session are not returned to the calling environment. Example 2 demonstrates this action.

Macro Facility
When running in the side session, the catalog for compiled macros changes. Instead of WORK.SASMACR, the catalog name is WORK.SASMACn, where *n* is 2, 3, 4, and so on. When SAS code in the different executive tries to resolve a macro name (for example, %ABC), SAS code searches in this order:

1. SAS code searches the local catalog WORK.SASMACn.
2. If SAMSTORE and MSTORED are set, SAS code searches the SASMSTORE location.
3. If SASAUTOS and MAUTOSOURCE are set, SAS code searches the SASAUTOS location.
4. SAS code searches WORK.SASMACR from the primary executive.

If macros are only in WORK.SASMACR and are not defined in any other executives or in SASMASTORE or SASAUTOS, any macro references resolve as expected. However, if macros are defined in other executives, or if the macros are available in SASMSTORE or SASAUTOS, the resolution order might result in unexpected output. Examples 3–6 demonstrate this behavior.

Examples

Example 1:

```
/* The %TRYIT macro is created. %TRYIT sets the global macro */
/* variable MYMAC from the macro code. */
%macro tryit(x);
%global mymac;
%if &x=1 %then %let mymac=2;
%else %let mymac=3;
%mend;

/* MYMAC is defined as a global variable, and its value is set to 4. */
%global mymac;
%let mymac=4;

/* The %TRYIT macro is invoked by the CALL EXECUTE routine. The macro
code */
/* is executed immediately, and MYMAC has a value of 2. The value of
the */
/* MYMAC variable is retrieved by using the SYMGET function. */

data _null_;
   call execute('%tryit(1)');
   value=symget('mymac');
   put value= '(should be 2)';
run;

/* The value of MYMAC is set to 4. */
%let mymac=4;

/* This DATA step uses the DOSUBL function to achieve the same result.
*/
data _null_;
   rc=dosubl('%tryit(1)');
   value=symget('mymac');
   put value= '(should be 2)';
run;

/* The definition of %TRYIT is changed so that a DATA step is */
/* invoked to set the value of MYMAC. */
%macro tryit(x);
%global mymac;
data _null_;
   if &x=1 then mymac=2;
   else mymac=3;
```

```
   call symputx('mymac', mymac);
run;
%mend;

/* The value of MYMAC is set to 4 by the %LET statement. */
/* CALL EXECUTE calls the %TRYIT macro and the macro executes
immediately. */
/* Execution of SAS statements generated by the execution of the macro
will
be delayed until after a step boundary.*/
/* SAS macro statements, including macro variable references, */
/* execute immediately.*/
/* Because the DATA step code contained within the macro does not */
/* execute until after the current DATA step completes the value of
MYMAC
 remains 4.*/
/* The call to the SYMGET function does not retrieve the proper value
of MYMAC,
which is 2. */

%let mymac=4;
data _null_;
   call execute('%tryit(1)');
   value=symget('mymac');
   put value= '(should be 2)';
run;

/* If you use the DOSUBL function, the code in the %TRYIT macro is */
/* executed immediately, and the SYMGET function returns a value of 2.
*/
%let mymac=4;
data _null_;
   rc=dosubl('%tryit(1)');
   value=symget('mymac');
   put value= '(should be 2)';
run;
```

Example 2:

This example shows the results of issuing LIBNAME statements outside and inside DOSUBL.

```
LIBNAME outside "C:\data"; /* 1 */
data _null_;
   rc = DOSUBL('LIBNAME inside "C:\newData";'); /* 2 */
run;

   %put outside = %sysfunc(pathname(outside));  /* 3 */
   %put inside = %sysfunc(pathname(inside));    /* 4 */
```

1 Issue a LIBNAME statement outside DOSUBL.

2 Issue a LIBNAME statement within DOSUBL to execute in the side session.

3 Write the path from the LIBNAME statement that was issued outside DOSUBL. This action displays C:\data.

4 Write the path from the LIBNAME statement that was issued inside DOSUBL. An empty string is written to the log because "inside" was not exported back to the calling environment.

```
outside = C:\data
inside =
```

Example 3:

```
%macro test1;    /*1*/
   %put this is test1;    /*2*/
%mend;

%test1;    /*3*/
data _null_;
   rc = dosubl('%test1;');    /*4*/
run;
```

1 Create a macro called test1.

2 Write the text *this is test1* to the log.

3 Invoke macro test1 outside DOSUBL.

4 Invoke macro test1 within DOSUBL.

Here is the output from the example. The first output line, *this is test1*, is written from the local macro execution. The second output line, *this is test1*, is written from the side session execution.

```
this is test1
this is test1
```

Example 4:

```
%macro test1;    /*1*/
   %put this is test1;    /*2*/
%mend;

%test1;    /*3*/
data _null_;
   rc=dosubl('%macro test1; %put this is changed test1; %mend;
   %test1;');    /*4*/
run;

%test1;    /*5*/
```

1 Create a macro called test1.

2 Write the text *this is test1* to the log.

3 Invoke macro test1 outside DOSUBL.

4 Create macro test1, and then invoke it within DOSUBL.

5 Invoke macro test1 outside DOSUBL.

```
this is test1
this is changed test1
this is test1
```

Example 5:

This example shows the results when SASMSTORE is not set.

```
%macro test1;   /*1*/
   %put this is test1;   /*2*/
   %mend;

%test1;   /*3*/
data _null_;
   rc = dosubl('%macro test2; %put this is test2; %mend;
%test2;');   /*4*/
run;

%test2;   /*5*/
```

1 Create a macro called test1.

2 Write the text *this is test1* to the log.

3 Invoke macro test1 outside DOSUBL.

4 Create macro test2, and then invoke it within DOSUBL.

5 Invoke macro test2 outside DOSUBL. This action produces a WARNING.

Here is the output from the code.

The first two lines of the output are the expected results. The warning message is written because the main session does not access WORK.SASMAC2 where test2 resides.

```
this is test1
this is test2


WARNING: Apparent invocation of macro TEST2 not resolved.
   8          %test2;
```

Example 6:

```
data _null_;
   rc = dcreate('temp',getoption('work'));  /*1*/
run;

libname temppath "%sysfunc(getoption(work))/temp";   /*2*/

options sasmstore=temppath mstored;   /*3*/

%macro test1/store;   /*4*/
   %put test1 stored/compiled;   /*5*/
%mend;

%test1;   /*6*/
```

```
data _null_;
   rc = dosubl('%test1;');    /* 7 */
run;

%macro test1;    /* 8 */
   %put test1 local;
%mend;

%test1;    /* 9 */

data _null_;
   rc = dosubl('%test1;');    /* 10 */
run;
```

1 Create a directory named **temp**, using the path to the **Work** directory.

2 Assign a libref called temppath to **temp**.

3 Set the option sasmstore to specify temppath, and then set mstored.

4 Create macro test1 and place it in the SASMSTORE= location.

5 Write test1 to the log.

6 Invoke macro test1 from outside DOSUBL. This action first searches in the primary catalog WORK.SASMACR. Because test1 does not exist in WORK.SASMACR, test1 invokes WORK.SASMACR from the SASMSTORE= location.

7 Invoke macro test1 from within DOSUBL. This action first searches in the local catalog WORK.SASMAC2. Because test1 does not exist in WORK.SASMAC2, test1 invokes WORK.SASMAC2 from the SASMSTORE= location.

8 Redefine a local version of macro test1. By default, this action is saved to the WORK.SASMACR catalog in the primary executive.

9 Invoke macro test1 from outside DOSUBL. This action first searches in the primary catalog WORK.SASMACR. Because test1 finds WORK.SASMACR there, that version of test1 is invoked.

10 Invoke macro test1 from within DOSUBL. This action first searches in the local catalog WORK.SASMAC2 (not in WORK.SASMACR). Because test1 does not exist in WORK.SASMAC2, test1 invokes WORK.SASMAC2 from the SASMSTORE= location.

Here are the results from the code:

```
test1 stored/compiled
test1 stored/compiled
test1 local
test1 stored/compiled
```

Example 7:

This example shows the scope of macro variables that are created by using DOSUBL. In this example, there are temporarily two versions of “A” in the side session that are both GLOBAL. Any references to “A” refer to the version that was created within the side session. The side session does not update the values of the

macro variable table until the side session returns to the calling environment. When the side session returns to the calling environment, the side session detects that there is an "A" in global scope, so the side session consolidates the values and maintains a single copy.

```
%macro outer();   /*1*/
   %global A;   /*2*/
   %let A = first;   /*3*/
   %inner()   /*4*/
%mend outer;

%macro inner();   /*5*/
   data _null_;
      rc = dosubl('   /*6*/
      %put --Side Session--;   /*7*/
      %put &=A;   /*8*/
      %let A=A_Side;   /*9*/
      %put &=A;   /*10*/
      %put _user_;   /*11*/
      ');
   run;
   %put --Main Session--;   /*12*/
   %put _user_;   /*13*/
%mend inner;

%outer()
```

1 Create a macro called outer.

2 Define A as a global variable.

3 Set A to first.

4 Invoke a macro inner.

5 Create a macro inner.

6 Invoke the function DOSUBL.

7 Write --Side Session-- to the log.

8 Write A=first to the log.

9 Assign the value A=A_Side.

10 Write the value of the macro that was defined in step 2.

11 Write the value of the user-generated macro variables.

12 Write the value --Main Session-- to the log.

13 Write the value of the user-generated macro variables.

```
--Side Session--
A=first
A=A_Side
GLOBAL A A_Side
GLOBAL A first

--Main Session--
GLOBAL A A_Side
```

See Also

CALL Routines:

- “CALL EXECUTE Routine” on page 235

DREAD Function

Returns the name of a directory member.

Category: External Files

Syntax

DREAD(*directory-id*, *nval*)

Required Arguments

directory-id
is a numeric value that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

nval
is a numeric constant, variable, or expression that specifies the sequence number of the member within the directory.

Details

DREAD returns a blank if an error occurs (such as when *nval* is out-of-range). Use DNUM to determine the highest possible member number that can be passed to DREAD.

Example

This example opens the directory identified by the fileref MYDIR, retrieves the number of members, and places the number in the variable MEMCOUNT. It then retrieves the name of the last member, places the name in the variable LSTNAME , and closes the directory:

```
%macro
test;
```

```
 %let
filrf=mydir;

 %let rc=%sysfunc(filename(filrf, physical-
name));

 %let did=
%sysfunc(dopen(&filrf));

 %let
lstname=;

 %let memcount=
%sysfunc(dnum(&did));

 %if &memcount > 0
%then

  %let lstname=%sysfunc(dread(&did,
&memcount));

 %let rc=
%sysfunc(dclose(&did));

%mend
test;



%test
```

See Also

Functions:

- “DNUM Function” on page 555
- “DOPEN Function” on page 556

DROPNOTE Function

Deletes a note marker from a SAS data set or an external file.

Categories: SAS File I/O
External Files

Syntax

DROPNOTE(*data-set-id | file-id*, *note-id*)

Required Arguments

data-set-id | file-id
is a numeric variable that specifies the identifier that was assigned when the data set or external file was opened, generally by the OPEN function or the FOPEN function.

note-id
is a numeric value that specifies the identifier that was assigned by the NOTE or FNOTE function.

Details

DROPNOTE deletes a marker set by NOTE or FNOTE. It returns a 0 if successful and ≠0 if not successful.

Example

This example opens the SAS data set MYDATA, fetches the first observation, and sets a note ID at the beginning of the data set. It uses POINT to return to the first observation, and then uses DROPNOTE to delete the note ID:

```
data
mydata;

input
id;

datalines;

1

2

3

;



%let dsid=%sysfunc(open(mydata,
i));

%let rc=
%sysfunc(fetch(&dsid));
```

```
%let noteid=
%sysfunc(note(&dsid));

%let rc=%sysfunc(point(&dsid,
&noteid));

%let rc=
%sysfunc(fetch(&dsid));

%let rc=%sysfunc(dropnote(&dsid,
&noteid));

%let rc=%sysfunc(close(&dsid));
```

See Also

Functions:

- “FETCH Function” on page 600
- “FNOTE Function” on page 718
- “FOPEN Function” on page 720
- “FPOINT Function” on page 728
- “NOTE Function” on page 1151
- “OPEN Function” on page 1183
- “POINT Function” on page 1221

DSNAME Function

Returns the SAS data set name that is associated with a data set identifier.

Category: SAS File I/O

Syntax

DSNAME(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that is returned by the OPEN function.

Details

DSNAME returns the data set name that is associated with a data set identifier, or a blank if the data set identifier is not valid.

Example

This example determines the name of the SAS data set that is associated with the variable DSID and displays the name in the SAS log.

```
%let dsid=%sysfunc(open(sasuser.houses,i));
%put The current open data set is
%sysfunc(dsname(&dsid)).;
```

These statements produce this result:

```
The current open data set is filename.
```

See Also

Functions:

- "OPEN Function" on page 1183

DUR Function

Returns the modified duration for an enumerated cash flow.

Category: Financial

Syntax

DUR(*y*, *f*, *c(1), ..., c(k)*)

Required Arguments

y

specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

f

specifies the frequency of cash flows per period.

Range $f > 0$

c(1), ..., c(k)
specifies a list of cash flows.

Details

The DUR function returns the value

$$C = \sum_{k=1}^{K} \frac{k\left(\frac{c(k)}{(1+y)^{\frac{k}{f}}}\right)}{(P(1+y)f)}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example

```
data _null_;
   d=dur(1/20,
1, .33, .44, .55, .49, .50, .22, .4, .8, .01, .36, .2, .4);
   put d;
run;
```

SAS writes the following output to the log:

```
5.284024988
```

DURP Function

Returns the modified duration for a periodic cash flow stream, such as a bond.

Category: Financial

Syntax

DURP(*A*, *c*, *n*, *K*, k_0, *y*)

Required Arguments

A

specifies the par value.

Range $A > 0$

c

specifies the nominal per-period coupon rate, expressed as a fraction.

Range $0 \le c < 1$

n

specifies the number of coupons per period.

Range $n > 0$ and is an integer

K

specifies the number of remaining coupons.

Range $K > 0$ and is an integer

k_o

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_0 \le 1/n$

y

specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Details

The DURP function returns the value

$$D = \frac{1}{n} \frac{\sum_{k=1}^{K} t_k \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)}$$

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:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{tk}}$$

Example

```
data _null_;
d=durp(1000, 1/100, 4, 14, .33/2, .10);
put d;
run;
```

SAS writes the following output to the log:

```
3.2649588109
```

EFFRATE Function

Returns the effective annual interest rate.

Category: Financial

Syntax

EFFRATE(*compounding-interval*, *rate*)

Required Arguments

compounding-interval
is a SAS interval. This value represents how often *rate* compounds.

rate
is numeric. *Rate* is a nominal annual interest rate (expressed as a percentage) that is compounded at each compounding interval.

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 examples show how the effective rate is calculated:

- If a nominal rate is 10%, then the corresponding effective rate when interest is compounded monthly can be expressed as

```
effective_rate1=EFFRATE('MONTH', 10);
```

- If a nominal rate is 10%, then the corresponding effective rate when interest is compounded quarterly can be expressed as

```
effective-rate2=EFFRATE('QUARTER', 10);
```

```
data one;
   effective_rate1=effrate('MONTH',10);
   put effective_rate1=;
run;
```

These statements produce this result:

```
effective_rate1=10.471306744
```

ENVLEN Function

Returns the length of an environment variable.

Category: SAS File I/O

Syntax

ENVLEN(*argument*)

Required Argument

argument
specifies a character variable that is the name of an operating system environment variable. Enclose *argument* in quotation marks.

Details

The ENVLEN function returns the length of the value of an operating system environment variable. If the environment variable does not exist, SAS returns –1.

Operating Environment Information: The value of *argument* is specific to your operating environment.

Example

The following examples are for illustration purposes only. The actual value that is returned depends on where SAS is installed on your computer.

```
data one;
   x=envlen("PATH");
   y=envlen("PATH");
   z=envlen("THIS IS NOT DEFINED");
   put x=;
   put y=;
   put z=;
run;
```

These statements produce these results:

```
x=332
y=332
z=-1
```

ERF Function

Returns the value of the (normal) error function.

Category: Mathematical

Syntax

ERF(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The ERF function returns the integral, given by

$$\mathrm{ERF}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \varepsilon^{-z^2} dz$$

Example

You can use the ERF function to find the probability (p) that a normally distributed random variable with mean 0 and standard deviation will take 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));
```

```
data one;
   x=erf(1.0);
   y=erf(-1.0);
   put x=;
   put y=;
run;
```

These statements produce these results:

```
x=0.8427007929
y=-0.842700793
```

ERFC Function

Returns the value of the complementary (normal) error function.

Category: Mathematical

Syntax

ERFC(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The ERFC function returns the complement to the ERF function (that is, 1 – ERF(*argument*)).

Example

```
data one;
   x=erfc(1.0);
   y=erfc(-1.0);
   put x=;
   put y=;
run;
```

These statements produce these results:

```
x=0.1572992071
y=1.8427007929
```

EUCLID Function

Returns the Euclidean norm of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

EUCLID(*value-1* <, *value-2 ...*>)

Required Argument

value
specifies a numeric constant, variable, or expression.

Details

If all arguments have missing values, then the result is a missing value. Otherwise, the result is the Euclidean norm of the nonmissing values.

In the following example, $x1, x2, ..., xn$ are the values of the nonmissing arguments.

$$EUCLID(x1, x2, ..., xn) = \sqrt{x1^2 + x2^2 + ... + xn^2}$$

Examples

Example 1: Calculating the Euclidean Norm of Nonmissing Arguments

The following example returns the Euclidean norm of the nonmissing arguments.

```
data _null_;
   x=euclid(., 3, 0, .q, -4);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=5
```

Example 2: Calculating the Euclidean Norm When You Use a Variable List

The following example uses a variable list to calculate the Euclidean norm.

```
data _null_;
   x1=1;
   x2=3;
   x3=4;
   x4=3;
   x5=1;
   x=euclid(of x1-x5);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=6
```

See Also

Functions:

- “LPNORM Function” on page 1091

■ “RMS Function” on page 1347

EXIST Function

Verifies the existence of a SAS library member within a currently assigned SAS data library.

Category: SAS File I/O

Syntax

EXIST(*'member-name'* <, *member-type* <, *generation*>>)

Required Argument

'member-name'
is a character constant, variable, or expression that specifies the SAS library member. If *member-name* is blank or a null string, then EXIST uses the value of the _LAST_ system variable as the member name.

Optional Arguments

member-type
is a character constant, variable, or expression that specifies the type of SAS library member. A few common member types include ACCESS, CATALOG, DATA, and VIEW. If you do not specify a *member-type*, then the member type DATA is assumed.

generation
is a numeric constant, variable, or expression that specifies the generation number of the SAS data set whose existence you are checking. If *member-type* is not DATA, *generation* is ignored.

Positive numbers are absolute references to a historical version by its generation number. Negative numbers are relative references to a historical version in relation to the base version, from the youngest predecessor to the oldest. For example, –1 refers to the youngest version or, one version back from the base version. Zero is treated as a relative generation number.

Details

If you use a sequential library, then the results of the EXIST function are undefined. If you do *not* use a sequential library, then EXIST returns 1 if the library member exists, or 0 if *member-name* does not exist or *member-type* is invalid. If the generation reference number is outside the bounds of generations for the member contained within a library the EXIST function returns a missing value.

Use the CEXIST function to verify the existence of an entry in a catalog.

Comparisons

The EXIST function verifies the existence of a data set, view, catalog, and access file. The CEXIST function verifies the existence of a catalog with the added functionality of verifying whether the catalog can be updated.

Examples

Example 1: Verifying the Existence of a Data Set

This example verifies the existence of a data set.

```
data want;
   if exist('have') then
      set have;
   else
      do;
         put "ERROR: dataset 'have' does not exist";
         call execute ('data 123');
      end;
run;
```

Example 2: Determining If a Generation Data Set Exists

This example verifies the existence of a generation data set by using positive generation numbers (absolute reference):

```
data new(genmax=3);
   x=1;
run;
data new;
   x=99;
run;
data new;
   x=100;
run;
data new;
   x=101;
run;
data _null_;
   test=exist('new', 'DATA', 4);
   put test=;
   test=exist('new', 'DATA', 3);
   put test=;
   test=exist('new', 'DATA', 2);
   put test=;
   test=exist('new', 'DATA', 1);
   put test=;
run;
```

SAS writes the following output to the log:

```
test=1
test=1
test=1
test=0
```

You can change this example to verify the existence of the generation data set by using negative numbers (relative reference):

```
data new2(genmax=3);
   x=1;
run;
data new2;
   x=99;
run;
data new2;
   x=100;
run;
data new2;
   x=101;
run;
data _null_;
   test=exist('new2', 'DATA', 0);
   put test=;
   test=exist('new2', 'DATA', -1);
   put test=;
   test=exist('new2', 'DATA', -2);
   put test=;
   test=exist('new2', 'DATA', -3);
   put test=;
   test=exist('new2', 'DATA', -4);
   put test=;
   test=exist('new2', 'DATA', -5);
   put test=;
run;
```

SAS writes the following output to the log:

```
test=1
test=1
test=1
test=0
test=0
NOTE: Argument 3 to function EXIST('new2','DATA',-5) at line 24 column 9 is
invalid.
test=.
```

See Also

Functions:

- “CEXIST Function” on page 427
- “FEXIST Function” on page 605

- “FILEEXIST Function” on page 609

EXP Function

Returns the value of the exponential function.

Category: Mathematical

Syntax

EXP(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression. For more information, see “Expressions” in *SAS Language Programmer’s Guide for SAS Viya Workbench*.

Details

The EXP function raises the constant *e*, which is approximately 2.71828, to the power that is supplied by the argument. The result is limited by the maximum value of a floating-point value on the computer.

Example

```
data one;
   x=exp(1.0);
   y=exp(0);
   put x=;
   put y=;
run;
```

These statements produce these results:

```
x=2.7182818285
y=1
```

See Also

SAS Language Reference: Concepts

- "SAS Operators" in *SAS Language Programmer's Guide for SAS Viya Workbench*

EXPM1 Function

Returns 1 less than the exponential function of the argument.

Category: Mathematical

Syntax

EXPM1(*x*)

Required Argument

x
specifies a numeric variable, constant, or expression.

Details

The EXPM1 function computes 1 less than the exponential function of the argument. The EXPM1 function is mathematically defined by the following equation:

$$EXPM1(x) = \exp(x) - 1$$

When x is close to 0, EXPM1(x) can be more accurate than $EXP(x) - 1$.

The EXPM1 function is the inverse function to the LOG1PX function. That is, in exact arithmetic, LOG1PX(EXPM1(x)) equals x.

When x is close to 0, the EXPM1 function uses an evaluation that is analyzed by Nicholas Higham in *Accuracy and Stability of Numerical Algorithms* (2d ed).

Examples

Example 1: Computing the Exponential by Using the EXPM1 Function

This example computes the exponential of the value 1E-15 and subtracts 1.

```
data _null_;
   y=expm1 (1E-15);
   put y= E16.;
run;
```

SAS writes the following output to the log:

```
x=1.000000000E-15
```

Example 2: Comparing the EXPM1 Function with the EXP Function

In this example, the value of Y is computed using the EXPM1 function. The value of Z is computed using the EXP function.

```
data _null_;
   y=expm1 (1E-15);
   put y= E16.;
   z=exp(1E-15) - 1;
   put z= E16.;
run;
```

SAS writes the following output to the log:

```
y=1.000000000E-15
z=1.110223025E-15
```

FACT Function

Computes a factorial.

Category: Mathematical

Syntax

FACT(*n*)

Required Argument

n
is a numeric constant, variable, or expression.

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

```
data
one;


x=fact(5);

   put
x=;

run;
```

These statements produce this result:

```
x=120
```

See Also

Functions:

- “COMB Function” on page 445
- “LFACT Function” on page 1068
- “PERM Function” on page 1217

FAPPEND Function

Appends the current record to the end of an external file.

Category: External Files

Syntax

FAPPEND(*file-id* <, *cc*>)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Optional Argument

cc
is a character constant, variable, or expression that specifies a carriage-control character.

Here are the valid values:

blank	indicates that the record starts a new line.
0	skips one blank line before this new line.
-	skips two blank lines before this new line.
1	specifies that the line starts a new page.
+	specifies that the line overstrikes a previous line.
P	specifies that the line is a computer prompt.
=	specifies that the line contains carriage control information.
all else	specifies that the line record starts a new line.

Details

FAPPEND adds the record that is currently contained in the File Data Buffer (FDB) to the end of an external file. FAPPEND returns a 0 if the operation was successful and ≠0 if it was not successful.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it moves data into the File Data Buffer, appends a record, and then closes the file. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));
```

```
 %let fid=%sysfunc(fopen(&filrf,
a));

  %if &fid > 0 %then
%do;

   %let rc=%sysfunc(fput(&fid, Data for the new
record));

   %let rc=
%sysfunc(fappend(&fid));

   %let rc=
%sysfunc(fclose(&fid));


%end;

  %else
%do;

   %put
%sysfunc(sysmsg());


%end;

%mend
test;



%test
```

See Also

Functions:

- “DOPEN Function” on page 556
- “FCLOSE Function” on page 591
- “FGET Function” on page 607
- “FOPEN Function” on page 720
- “FPUT Function” on page 733
- “FWRITE Function” on page 744
- “MOPEN Function” on page 1121

FCLOSE Function

Closes an external file, directory, or directory member.

Category: External Files

Syntax

FCLOSE(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FCLOSE returns a 0 if the operation was successful and ≠0 if it was not successful. If you open a file within a DATA step, it is closed automatically when the DATA step ends.

Operating Environment Information: For the Linux OS, FCLOSE is not required at the end of the DATA step.

Example

This example assigns the fileref MYFILE to an external file, and attempts to open the file. If the file is opened successfully, indicated by a positive value in the variable FID, the program reads the first record, closes the file, and deassigns the fileref:

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=
%sysfunc(fopen(&filrf));
```

```
  %if &fid > 0 %then
%do;

   %let rc=
%sysfunc(fread(&fid));

   %let rc=
%sysfunc(fclose(&fid));


%end;

  %else
%do;

   %put
%sysfunc(sysmsg());


%end;

 %let rc=
%sysfunc(filename(filrf));

%mend
test;



%test
```

See Also

Functions:

- “DCLOSE Function” on page 522
- “DOPEN Function” on page 556
- “FOPEN Function” on page 720
- “FREAD Function” on page 735
- “MOPEN Function” on page 1121

FCOL Function

Returns the current column position in the File Data Buffer (FDB).

Category: External Files

Syntax

FCOL(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

Use FCOL combined with FPOS to manipulate data in the File Data Buffer (FDB).

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is successfully opened, indicated by a positive value in the variable FID, it puts more data into the FDB relative to position POS, writes the record, and closes the file:

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=%sysfunc(fopen(,
o));

  %if ( > 0) %then
%do;

   %let record=This is data for the
record.;

   %let rc=
%sysfunc(fput(, ));

   %let pos=
%sysfunc(fcol());

   %let rc=%sysfunc(fpos(,
%eval(+1)));
```

```
   %let rc=%sysfunc(fput(, more
data));

   %let rc=
%sysfunc(fwrite());

   %let rc=
%sysfunc(fclose());


%end;

 %let rc=
%sysfunc(filename(filrf));

%mend
test;



   %test
```

SAS writes the following new record to the external file:

```
This is data for the record. more data
```

See Also

Functions:

- “FCLOSE Function” on page 591
- “FOPEN Function” on page 720
- “FPOS Function” on page 731
- “FPUT Function” on page 733
- “FWRITE Function” on page 744
- “MOPEN Function” on page 1121

FCOPY Function

Copies records from one fileref to another fileref, and returns a value that indicates whether the records were successfully copied.

Categories: External Files
SAS File I/O

Syntax

FCOPY('*fileref-1*', '*fileref-2*')

Required Arguments

'fileref-1'
specifies an existing fileref from which records are to be copied.

'fileref-2'
specifies an existing fileref to which records are to be copied.

Details

Values That Are Returned by the FCOPY Function

FCOPY returns these values:

- a value of 0 if records were copied without errors or warnings
- a positive value if an error occurred
- a negative value if a warning was issued

You can use the SYSMSG function to retrieve error or warning messages, and you can use the SYSRC function to retrieve the return code.

Using Macro Variables with the FCOPY Function

The following macro variables provide information for the FCOPY function:

- The &SYSCC and &SYSERR macro variables are set if FCOPY writes an error or warning message to the log.
- The &SYSCC and &SYSERR macro variables are not set if FCOPY returns a warning return code and there is no log output from FCOPY.
- The &SYSERRORTEXT macro variable is set if FCOPY writes an error message to the log.
- The &SYSWARNINGTEXT macro variable is set if FCOPY writes a warning message to the log.

Setting a Logical Record Length for a Text File

For Linux OS, the default logical record length for reading from or writing to external files is 32,767 bytes. The maximum logical record length is 1 gigabyte. Text files have a data stream that consists of an unstructured sequence of bytes. A delimiter, such as a carriage-control character, controls the length of the data stream, and divides the information in the data stream into records. If the length of a record is greater than 32,767 bytes, you must define the logical record length of your records so that your data is not truncated when FCOPY copies the text file. To set the logical record length to a larger value, use the LRECL= system option in an OPTIONS statement, or the LRECL= option in a FILENAME statement.

For an example that shows how to set an LRECL value, see "Example 3: Diagnostic Messages" on page 598.

> **TIP** Selecting an arbitrarily large value for the LRECL= option can result in excessive use of memory, which can degrade performance.

Examples

Example 1: Copying a Text File

Setting the MSGLEVEL= system option to I causes informational messages from FCOPY to be written to the log.

```
   /* Set MSGLEVEL to I to write messages from FCOPY to the log. */
options msglevel=i;

filename src 'source.txt';
filename dest 'destination.txt';

   /* Create an example file to copy. */
data _null_;
   file src;
   do i=1, 2105, 300312, 400501;
      put i:words256.;
   end;
run;

   /* Copy the records of SRC to DEST. */
data _null_;
   length msg $ 384;
   rc=fcopy('src', 'dest');
   if rc=0 then
      put 'Copied SRC to DEST.';
   else do;
      msg=sysmsg();
      put rc= msg=;
   end;
run;
```

SAS writes the following output to the log:

```
INFO: The source fileref SRC for the FCOPY function is:
      Filename=your-source-file,
      RECFM=V,LRECL=32767,File Size (bytes)=121,
      Last Modified=15Aug2012:11:21:39,
      Create Time=15Aug2012:09:13:38

INFO: The destination fileref DEST for the FCOPY function is:
      Filename=your-destination-file,
      RECFM=V,LRECL=32767,File Size (bytes)=0,
      Last Modified=15Aug2012:11:21:39,
      Create Time=15Aug2012:09:13:39

Copied SRC to DEST.
```

Example 2: Copying a Binary File

This example copies a binary file from one directory to another. Setting the MSGLEVEL= system option to I causes informational messages from FCOPY to be written to the log.

```
   /* Set MSGLEVEL to I to write messages from FCOPY to the log. */
options msglevel=i;

filename src 'raises.xlsx' recfm=n;
filename dest 'raises-2012.xlsx' recfm=n;

   /* Create an example file to copy. */
data _null_;
   file src;
   do i=1, 2105, 300312, 400501;
     put i:words256.;
   end;
run;

data _null_;
   length msg $ 384;
   rc=fcopy('src', 'dest');
   if rc=0 then
      put 'Copied SRC to DEST.';
   else do;
      msg=sysmsg();
      put rc= msg=;
   end;
run;
```

SAS writes the following output to the log:

```
INFO: The source fileref SRC for the FCOPY function is:
      Filename=your-source-file,
      RECFM=N,LRECL=256,File Size (bytes)=117,
      Last Modified=15Aug2012:12:49:18,
      Create Time=15Aug2012:12:42:00

INFO: The destination fileref DEST for the FCOPY function is:
      Filename=your-destination-file,
      RECFM=N,LRECL=256,File Size (bytes)=0,
      Last Modified=15Aug2012:12:49:18,
      Create Time=15Aug2012:12:42:01

Copied SRC to DEST.
```

Example 3: Diagnostic Messages

This example shows diagnostic messages that result from the FCOPY function when the MSGLEVEL= system option is set to I. The file to be copied from has a record length of 256 bytes, and the file to be copied to has a record length of 5 bytes. Warning messages identify that the file was truncated.

```
filename src 'source.txt' lrecl=256;

   /* Create example file to copy. */
data _null_;
   file src;
   do i=1, 2105, 300312, 400501;
      put i:words256.;
   end;
run;

   /* Make LRECL for DEST short, to force output truncation. */
filename dest 'destination.txt' lrecl=5;

   /* Set MSGLEVEL to I to write messages from FCOPY to the log. */
  options msglevel=i;
data _null_;
   rc=fcopy('src', 'dest');
run;
```

SAS writes the following output to the log:

```
INFO: The source fileref SRC for the FCOPY function is:
      Filename=your-source-file,
      RECFM=V,LRECL=256,File Size (bytes)=121,
      Last Modified=15Aug2012:15:14:38,
      Create Time=15Aug2012:09:13:38

INFO: The destination fileref DEST for the FCOPY function is:
      Filename=your-destination-file,
      RECFM=V,LRECL=5,File Size (bytes)=0,
      Last Modified=15Aug2012:15:14:38,
      Create Time=15Aug2012:09:13:39

WARNING: 3 records were truncated when the FCOPY function wrote to fileref DEST.
WARNING: To prevent the truncation of records in future operations, you can
         increase amount of space needed to accomodate the records by using
         the LRECL= system option or the LRECL= option in the FILENAME statement.
```

FDELETE Function

Deletes an external file or an empty directory.

Category: External Files

See: "FILENAME Function" on page 611

Syntax

FDELETE(*fileref* | *directory*)

Required Arguments

fileref
is a character constant, variable, or expression that specifies the *fileref* that you assign to the external file or directory. You can assign *filerefs* by using the FILENAME statement, the FILENAME external file access function, or the FILENAME statement, FTP, Catalog, Hadoop, WebDAV, and ZIP access methods.

Restriction: The *fileref* that you use with FDELETE cannot be a concatenation.

Operating environment: For the Linux OS, if the function is used in a DATA step, the *fileref* or the environment variable that you specify must be enclosed in quotation marks. If the function is used in macro code, then the *fileref* must not be enclosed in quotation marks.

directory
is a character constant, variable, or expression that specifies an empty directory that you want to delete.

Restriction: You must have authorization to delete the directory.

Details

FDELETE returns 0 if the operation was successful or a nonzero number if it was not successful.

Note: If you use the FILENAME statement, WebDAV access method, you can delete a non-empty directory by using the DEL_ALL WebDAV option. For more information, see "FILENAME Statement: WebDAV Access Method" in *SAS Global Statements for SAS Viya Workbench*.

Examples

Example 1: Deleting an External File

This example generates a *fileref* for an external file in the variable FNAME. Then it calls FDELETE to delete the file and calls the FILENAME function again to deassign the *fileref*.

```
data _null_;
    fname="tempfile";
    rc=filename(fname, "physical-filename");
    if rc = 0 and fexist(fname) then
       rc=fdelete(fname);
    rc=filename(fname);
run;
```

Example 2: Deleting a Directory

This example uses FDELETE to delete an empty directory to which you have Write access. If the directory is not empty, the optional SYSMSG function returns an error message stating that SAS is unable to delete the file.

```
filename testdir 'physical-filename';
data _null_;
   rc=fdelete('testdir');
   put rc=;
   msg=sysmsg();
   put msg=;
run;
```

See Also

Functions:

- “FEXIST Function” on page 605

Statements:

- “FILENAME Statement” in *SAS Global Statements for SAS Viya Workbench*

FETCH Function

Reads the next non-deleted observation from a SAS data set into the Data Set Data Vector (DDV).

Category: SAS File I/O

Syntax

FETCH(*data-set-id* <, 'NOSET'>)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that is returned by the OPEN function.

Optional Argument

'NOSET'
prevents the automatic passing of SAS data set variable values to macro or DATA step variables even if the SET routine has been called.

Details

FETCH returns a 0 if the operation is successful, ≠0 if it is not successful, and – 1 if the end of the data set is reached. FETCH skips observations marked for deletion.

If the SET routine has been called previously, the values for any data set variables are automatically passed from the DDV to the corresponding DATA step or macro variables. To override this behavior temporarily so that fetched values are not automatically copied to the DATA step or macro variables, use the NOSET option.

Example

This example fetches the next observation from the SAS data set MYDATA. If the end of the data set is reached or if an error occurs, SYSMSG retrieves the appropriate message and writes it to the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro
test;

 %let dsid=%sysfunc(open(mydata,
i));

 %let rc=
%sysfunc(fetch(&dsid));

 %if &rc ne 0
%then

  %put
%sysfunc(sysmsg());
```

```
 %else
%do;

  %put fetch was
successful;


%end;

 %let rc=
%sysfunc(close(&dsid));

%mend
test;



%test
```

See Also

Functions:

- “FETCHOBS Function” on page 602
- “GETVARC Function” on page 765
- “GETVARN Function” on page 768

CALL Routines:

- “CALL SET Routine” on page 336

FETCHOBS Function

Reads a specified observation from a SAS data set into the Data Set Data Vector (DDV).

Category: SAS File I/O

Syntax

FETCHOBS(*data-set-id*, *observation-number* <, *options*>)

Required Arguments

data-set-id
is a numeric variable that specifies the data set identifier that is returned by the OPEN function.

observation-number
is a numeric constant, variable, or expression that specifies the number of the observation to read. FETCHOBS treats the observation value as a relative observation number unless you specify the ABS option. The relative observation number might not coincide with the physical observation number on disk, because the function skips observations marked for deletion. When a WHERE clause is active, the function counts only observations that meet the WHERE condition.

Default FETCHOBS skips deleted observations.

Optional Argument

options
is a character constant, variable, or expression that names one or more options, separated by blanks.

Here are valid values:

ABS specifies that the value of *observation-number* is absolute. That is, deleted observations are counted.

NOSET prevents the automatic passing of SAS data set variable values to DATA step or macro variables even if the SET routine has been called.

Details

FETCHOBS returns 0 if the operation was successful, ≠0 if it was not successful, and −1 if the end of the data set is reached. To retrieve the error message that is associated with a nonzero return code, use the SYSMSG function. If the SET routine has been called previously, the values for any data set variables are automatically passed from the DDV to the corresponding DATA step or macro variables. To override this behavior temporarily, use the NOSET option.

If *observation-number* is less than 1, the function returns an error condition. If *observation-number* is greater than the number of observations in the SAS data set, the function returns an end-of-file condition.

Example

This example fetches the 10th observation from the SAS data set MYDATA. If an error occurs, the SYSMSG function retrieves the error message and writes it to the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
data
mydata;
```

```
 do i=1 to
20;


output;


end;

run;



%macro
test;

 %let mydataid=
%sysfunc(open(mydata));

 %let rc =
%sysfunc(fetchobs(&mydataid,10));



  %if &rc = -1
%then

   %put End of data set has been
reached.;

 %if &rc > 0 %then %put
%sysfunc(sysmsg());

 %let rc=
%sysfunc(close(&mydataid));

%mend
test;



%test
```

See Also

Functions:

- “FETCH Function” on page 600
- “GETVARC Function” on page 765
- “GETVARN Function” on page 768

CALL Routines:

- "CALL SET Routine" on page 336

FEXIST Function

Verifies the existence of an external file that is associated with a *fileref*.

Category: External Files

See: "FILENAME Function" on page 611

Syntax

FEXIST(*fileref*)

Required Argument

fileref
is a character constant, variable, or expression that specifies the *fileref* that is assigned to an external file.

Restriction: *Fileref* must have been previously assigned.

Operating environment: Under Linux, if the function is used in a DATA step, *fileref* or the environment variable that you specify must be enclosed in quotation marks. If the function is used in macro code, ,*fileref* must not be enclosed in quotation marks.

Details

FEXIST returns 1 if the external file that is associated with *fileref* exists, and 0 if the file does not exist. You can assign *filerefs* by using the FILENAME statement or the FILENAME external file access function. In some operating environments, you can also assign *filerefs* by using system commands.

Note: If you are using the FILENAME statement, FTP access method to reference the file used in the FEXIST function and the function reports a data set not found but the data set is cataloged, use the FTP access method's WAIT_MILLISECONDS option to increase the response time.

Comparisons

FILEEXIST verifies the existence of a file based on its physical name.

Example

This example verifies the existence of an external file and writes the result to the SAS log:

```
filename mytest
'c:\test.txt';

%let
fref=mytest;



%macro
test;

 %if %sysfunc(fexist(&fref))
%then

  %put The file identified by the fileref &fref
exists.;


%else

 %put
%sysfunc(sysmsg());

%mend
test;



%test
```

See Also

Functions:

- “EXIST Function” on page 582
- “FILEEXIST Function” on page 609
- “FILEREF Function” on page 616

Statements:

- “FILENAME Statement” in *SAS Global Statements for SAS Viya Workbench*

FGET Function

Copies data from the File Data Buffer (FDB) into a variable.

Category: External Files

Syntax

FGET(*file-id*, *variable* <, *length*>)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

variable
in a DATA step, specifies a character variable to hold the data. In a macro, specifies a macro variable to hold the data. If *variable* is a macro variable and it does not exist, it is created.

Optional Argument

length
specifies the number of characters to retrieve from the FDB. If *length* is specified, only the specified number of characters is retrieved (or the number of characters remaining in the buffer if that number is less than length). If *length* is omitted, all characters in the FDB from the current column position to the next delimiter are returned. The default delimiter is a blank. The delimiter is not retrieved.

See See "FSEP Function" on page 741 for more information about delimiters.

Details

FGET returns 0 if the operation was successful, or −1 if the end of the FDB was reached or no more tokens were available.

After FGET is executed, the column pointer moves to the next read position in the FDB.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it reads the first record into the File Data Buffer, retrieves the first token of the record and stores i t in the variable MYSTRING, and then closes the file. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=
%sysfunc(fopen(&filrf));

  %if &fid > 0 %then
%do;

   %let rc=
%sysfunc(fread(&fid));

   %let rc=%sysfunc(fget(&fid,
mystring));

   %put
&mystring;

   %let rc=
%sysfunc(fclose(&fid));


%end;

 %let rc=
%sysfunc(filename(filrf));

%mend
test;



%test
```

See Also

Functions:

- “FCLOSE Function” on page 591

- "FILENAME Function" on page 611
- "FOPEN Function" on page 720
- "FPOS Function" on page 731
- "FREAD Function" on page 735
- "FSEP Function" on page 741
- "MOPEN Function" on page 1121

FILEEXIST Function

Verifies the existence of an external file by its physical name.

Category: External Files

See: "FILENAME Function" on page 611

Syntax

FILEEXIST(*file-name*)

Required Argument

filename
is a character constant, variable, or expression that specifies a fully qualified physical filename of the external file in the operating environment.

Details

FILEEXIST returns 1 if the external file exists and 0 if the external file does not exist. The specification of the physical name for *filename* varies according to the operating environment.

Although your operating environment utilities might recognize partial physical filenames, you must always use fully qualified physical filenames with FILEEXIST.

FILEEXIST can also verify a directory's existence.

Example

This example verifies the existence of an external file. If the file exists, FILEEXIST opens the file. If the file does not exist, FILEEXIST displays a message in the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let myfilepath=%sysfunc(pathname(work))/
abcd.txt;

%macro
test;

  %if %sysfunc(fileexist(&myfilepath)) %then %do;
     %put "The file at &myfilepath exists.";

     /*
      *You can use FOPEN with this code.*/
    filename myfilerf "&myfilepath";
    %let fid=%sysfunc(fopen(myfilerf));

    data null;
        if &fid eq 0 then put "ERROR: CORRECT WAY: FILE FAILED TO
OPEN";
        else put "NOTE: CORRECT WAY: FILE OPENED";
    run;

    %let found_file_size = %sysfunc(finfo(&fid, File Size
(bytes)));
    %let rc = %sysfunc(fclose(&fid));
    %put FileRef myfilerf to path &myfilepath has a size of
&found_file_size;

  %end;

%else

  %put The external file &myfilepath does not
exist.;

%mend
test;



options
mprint;

%test ;
filename myfilerf clear;
```

See Also

Functions:

- “EXIST Function” on page 582
- “FEXIST Function” on page 605
- “FILEREF Function” on page 616
- “FOPEN Function” on page 720

FILENAME Function

Assigns or deassigns a fileref to an external file, directory, or output device.

Category: External Files

Important: If the FILENAME function contains only one argument, *fileref*, then *fileref* is deassigned.

If the first argument in the FILENAME function is a variable, then the variable value is set to blank.

If the FILENAME function contains two or more arguments, and the first argument, *fileref*, is a variable with a blank value, FILENAME creates a non-blank fileref that is 8 bytes long, and the 8-byte value is assigned to the variable in the first argument. If the variable has a length less than 8, then the value of the fileref is truncated and is not usable.

Syntax

FILENAME(*fileref*, <*filename*> <,*device-type*> <, '*host-options*'> <, *directory-reference*>); [Linux]

Required Argument

fileref

specifies the fileref to assign to the external file. In a DATA step, *fileref* can be a character expression, a string enclosed in single quotation marks that specifies the fileref, or a DATA step variable whose value contains the fileref. In a macro (for example, in the %SYSFUNC function), *fileref* is the name of a macro variable (without an ampersand) whose value contains the fileref to assign to the external file. If the function is used within a DATA step, the fileref must be enclosed in single quotation marks. If the function is used in macro code, the fileref must not be enclosed in quotation marks.

Requirement If *fileref* is a DATA step variable, its length must be no longer than eight characters.

Tip If a fileref is a DATA step character variable with a blank value and a maximum length of eight characters, or if a macro variable named in *fileref* has a null value, then a fileref is generated and assigned to the character variable or macro variable, respectively.

Optional Arguments

filename

is a character constant, variable, or expression that specifies the external file.

Linux Specifics: The filename differs according to the device type. For more information that is appropriate for each device, see "FILENAME Statement" in *SAS Global Statements for SAS Viya Workbench*. . Remember that Linux filenames are case sensitive.

Tip You can deassign a fileref that was previously assigned by passing a blank value for the *filename* argument.

device-type
is a character constant, variable, or expression that specifies the type of device or the access method that is used if the fileref points to an input or output device or location that is not a physical file.

Here are valid values:

Linux Specifics:

DISK is the default device type. The device type can be any one of the devices that are listed in "FILENAME Statement" in *SAS Global Statements for SAS Viya Workbench*.

DISK
specifies that the device is a disk drive.

Alias BASE

Tip When you assign a fileref to a file on disk, you are not required to specify DISK.

DUMMY
specifies that the output to the file is discarded.

Tip Specifying DUMMY can be useful for testing.

FILESRVC
specifies an access method that enables you to store and retrieve user content using the SAS Viya service.

Note: For more information see, "FILENAME Statement: FILESRVC Access Method" in *SAS Global Statements for SAS Viya Workbench*.

Here are two examples using the FILESRVC argument.

This example specifies FILESRVC with a file.

```
data _NULL_;
  rc = filename('filrf', '','FILESRVC', "folderpath= '/Users/UserId/My Folder' filename=
  put rc=;
  fid = fopen("filrf");
  put fid=;
run;
```

This example specifies FILESRVC with a folder.

```
data _NULL_;
  rc = filename('dirrf', '','FILESRVC', "folderpath= '/Users/UserId/My Folder' ");
  put rc=;
```

```
  did = dopen("dirrf");
  put did=;
run;
```

GTERM
indicates that the output device type is a graphics device that is receiving graphics data.

PIPE
specifies an unnamed pipe.

Note Some operating environments do not support pipes.

PLOTTER
specifies an unbuffered graphics output device.

PRINTER
specifies a printer or printer spool file.

TAPE
specifies a tape drive.

TEMP
creates a temporary file that exists only as long as the filename is assigned. The temporary file can be accessed only through the logical name and is available only while the logical name exists.

Restriction Do not specify a physical pathname. If you do, SAS returns an error.

Tip Files that are manipulated by the TEMP device can have the same attributes and behave identically to DISK files.

TERMINAL
specifies the user's personal computer.

UPRINTER
specifies a Universal Printing printer definition name.

Operating Environment Information: The FILENAME function also supports operating-environment specific devices. For more information, see the SAS documentation for your operating environment.

'host-options'
specifies host-specific details such as file attributes and processing attributes.

Linux Specifics:

You can use any of the options that are available in the FILENAME statement.

directory-reference
specifies the fileref that was assigned to the directory or partitioned data set in which the external file resides.

Details

You can create a *fileref* with the FILENAME function and FILENAME statement.

FILENAME returns 0 if the operation was successful; ≠0 if it was not successful. The name that is associated with the file or device is called a *fileref* (file reference name). Other system functions that manipulate external files and directories require that the files be identified by fileref rather than by physical filename. The association between a fileref and a physical file lasts only for the duration of the current SAS session or until you change or discontinue the association by using FILENAME.

You can deassign filerefs by specifying one argument in the FILENAME function, or by passing a blank value for the *filename* argument.

Operating Environment Information: The term directory in this description refers to an aggregate grouping of files that are managed by the operating environment. Different operating environments identify these groupings with different names, such as directory, subdirectory, folder, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment.

Under some operating environments, you can also assign filerefs by using system commands. Depending on the operating environment, FILENAME might be unable to change or deassign filerefs that are assigned outside of SAS.

Examples

Example 1: Accessing Directory Information

This example uses the FILENAME function and other functions to access directory information.

```
data a;
  rc=filename("tmpdir", "c:");
  put "rc = 0 if the directory exists: " rc=;
  did=dopen("tmpdir");
  put did=;
  numopts=doptnum(did);
  put numopts=;
  do i = 1 to numopts;
    optname = doptname(did,i);
    put i= optname=;
    optval=dinfo(did,optname);
    put optval=;
  end;
run;

proc printto; run;
```

These code statements produce this log output. Note that the filename function outputs a 0.

```
446  data a;
447    rc=filename("tmpdir", "c:");
448    put "rc = 0 if the directory exists: " rc=;
449    did=dopen("tmpdir");
450    put did=;
451    numopts=doptnum(did);
452    put numopts=;
453    do i = 1 to numopts;
454      optname = doptname(did,i);
455      put i= optname=;
456      optval=dinfo(did,optname);
457      put optval=;
458    end;
459  run;
rc = 0 if the directory exists: rc=0
did=1
numopts=1
i=1 optname=Directory
optval=C:\TEMP\dir
NOTE: The data set WORK.A has 1 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time           0.08 seconds
      cpu time            0.04 seconds
```

Example 2: Assigning a Fileref to an External File

This example assigns the fileref MYFILE to an external file. Next, it deassigns the fileref. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %if &rc ne 0 %then %put
%sysfunc(sysmsg());

 %let rc=
%sysfunc(filename(filrf));

%mend
test;



%test
```

Example 3: Assigning a System-Generated Fileref

This example assigns a system-generated fileref to an external file. The fileref is stored in the variable FNAME. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro
test;

 %let rc=%sysfunc(filename(fname, physical-
filename));

 %if &rc %then %put
%sysfunc(sysmsg());

 %else
%do;

  %put
testing;


%end;

%mend
test;



%test
```

Example 4: Assigning a Fileref to a Pipe File

This example assigns the fileref MYPIPE to a pipe file with the output from the Linux command LS, which lists the files in the directory /u/myid. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=mypipe;
%let rc=%sysfunc(filename(filrf, %str(ls /u/myid), pipe));
```

See Also

Functions:

- “FEXIST Function” on page 605
- “FILEEXIST Function” on page 609
- “FILEREF Function” on page 616
- “SYSMSG Function” on page 1437

FILEREF Function

Verifies whether a *fileref* has been assigned for the current SAS session.

Category: External Files

Syntax

FILEREF(*fileref*)

Required Argument

fileref
is a character constant, variable, or expression that specifies the *fileref* to be validated.

Linux Specifics: In a DATA step, the *fileref* that you specify must be enclosed in double quotation marks. In macro code, *fileref* must not be enclosed in double quotation marks. You can assign *filerefs* by using the FILENAME statement or the FILENAME external file access function. For the Linux OS, *fileref* can also be a Linux environment variable.

Range 1 to 8 characters

Details

The FILEREF function specifies whether a fileref is assigned and if the file exists. There are three return values: LT 0 (fileref is assigned and the file does not exist) , EQ 0 (fileref is assigned and the file exists), and GT 0 (fileref is not assigned). Numeric values are not defined. Here are the possible conditions:

- A negative return code indicates that the *fileref* exists but the physical file associated with the *fileref* does not exist.
- A positive value indicates that the *fileref* is not assigned.
- A value of zero indicates that the *fileref* and external file both exist.

A *fileref* can be assigned to an external file by using the FILENAME statement or the FILENAME function.

Examples

Example 1: Verifying That a Fileref Is Assigned

This example tests whether the *fileref* MYFILE is currently assigned to an external file. A system error message is issued if the *fileref* is not currently assigned:

```
%macro
test;

 %if %sysfunc(fileref(myfile))>0
%then

 %put MYFILE is not
assigned;
```

```
%mend
test;



%test
```

Example 2: Verifying That Both a Fileref and a File Exist

This example tests for a zero value to determine whether both the *fileref* and the file exist:

```
%macro
test;

 %if %sysfunc(fileref(myfile)) ne 0
%then

%put
%sysfunc(sysmsg());

%mend
test;



%test
```

See Also

Functions:

- “FEXIST Function” on page 605
- “FILEEXIST Function” on page 609
- “FILENAME Function” on page 611
- “SYSMSG Function” on page 1437

Statements:

- “FILENAME Statement” in *SAS Global Statements for SAS Viya Workbench*

FINANCE Function

Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.

Category: Financial

Syntax

FINANCE(*string-identifier*, *parameter–1*, *parameter–2*, ..., *parameter-n*)

Required Arguments

string-identifier

specifies a character constant, variable, or expression. Valid values for *string-identifier* are listed in the following table.

string-identifier	Description
"FINANCE ACCRINT Function" on page 623	computes the accrued interest for a security that pays periodic interest.
"FINANCE ACCRINTM Function" on page 624	computes the accrued interest for a security that pays interest at maturity.
"FINANCE AMORDEGRC Function" on page 625	computes the depreciation for each accounting period by using a depreciation coefficient.
"FINANCE AMORLINC Function" on page 627	computes the depreciation for each accounting period.
"FINANCE COUPDAYBS Function" on page 628	computes the number of days from the beginning of the coupon period to the settlement date.
"FINANCE COUPDAYS Function" on page 629	computes the number of days in the coupon period that contains the settlement date.
"FINANCE COUPDAYSNC Function" on page 630	computes the number of days from the settlement date to the next coupon date.
"FINANCE COUPNCD Function" on page 631	computes the next coupon date after the settlement date.
"FINANCE COUPNUM Function" on page 632	computes the number of coupons that are payable between the settlement date and the maturity date.
"FINANCE COUPPCD Function" on page 633	computes the previous coupon date before the settlement date.
"FINANCE CUMIPMT Function" on page 634	computes the cumulative interest that is paid between two periods.

string-identifier	Description
"FINANCE CUMPRINC Function" on page 635	computes the cumulative principal that is paid on a loan between two periods.
"FINANCE DDB Function" on page 637	computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
"FINANCE DDB Function" on page 637	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" on page 639	computes the discount rate for a security.
"FINANCE DOLLARDE Function" on page 640	converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
"FINANCE DOLLARFR Function" on page 640	converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.
"FINANCE DURATION Function" on page 641	computes the annual duration of a security with periodic interest payments.
"FINANCE EFFECT Function" on page 642	computes the effective annual interest rate.
"FINANCE FV Function" on page 643	computes the future value of an investment.
"FINANCE FVSCHEDULE Function" on page 644	computes the future value of an initial principal after applying a series of compound interest rates.
"FINANCE INTRATE Function" on page 645	computes the interest rate for a fully invested security.
"FINANCE IPMT Function" on page 646	computes the interest payment for an investment for a given period.
"FINANCE IRR Function" on page 647	computes the internal rate of return for a series of cash flows.
"FINANCE ISPMT Function" on page 648	calculates the interest paid during a specific period of an investment.

string-identifier	Description
"FINANCE MDURATION Function" on page 649	computes the Macaulay modified duration for a security with an assumed face value of $100.
"FINANCE MIRR Function" on page 650	computes the internal rate of return where positive and negative cash flows are financed at different rates.
"FINANCE NOMINAL Function" on page 651	computes the annual nominal interest rate.
"FINANCE NPER Function" on page 652	computes the number of periods for an investment.
"FINANCE NPV Function" on page 653	computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
"FINANCE ODDFPRICE Function" on page 654	computes the price per $100 face value of a security with an odd first period.
"FINANCE ODDFYIELD Function" on page 655	computes the yield of a security with an odd first period.
"FINANCE ODDLPRICE Function" on page 657	computes the price per $100 face value of a security with an odd last period.
"FINANCE ODDLYIELD Function" on page 658	computes the yield of a security with an odd last period.
"FINANCE PMT Function" on page 659	computes the periodic payment for an annuity.
"FINANCE PPMT Function" on page 661	computes the payment on the principal for an investment for a given period.
"FINANCE PRICE Function" on page 662	computes the price per $100 face value of a security that pays periodic interest.
"FINANCE PRICEDISC Function" on page 663	computes the price per $100 face value of a discounted security.
"FINANCE PRICEMAT Function" on page 664	computes the price per $100 face value of a security that pays interest at maturity.
"FINANCE PV Function" on page 665	computes the present value of an investment.

string-identifier	Description
“FINANCE RATE Function” on page 666	computes the interest rate per period of an annuity.
“FINANCE RECEIVED Function” on page 667	computes the amount received at maturity for a fully invested security.
“FINANCE SLN Function” on page 668	computes the straight-line depreciation of an asset for one period.
“FINANCE SYD Function” on page 669	computes the sum-of-years digits depreciation of an asset for a specified period.
“FINANCE TBILLEQ Function” on page 670	computes the bond-equivalent yield for a treasury bill.
“FINANCE TBILLPRICE Function” on page 671	computes the price per $100 face value for a treasury bill.
“FINANCE TBILLYIELD Function” on page 672	computes the yield for a treasury bill.
“FINANCE VDB Function” on page 673	computes the depreciation of an asset for a specified or partial period by using a declining balance method.
“FINANCE XIRR Function” on page 674	computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
“FINANCE XNPV Function” on page 675	computes the net present value for a schedule of cash flows that is not necessarily periodic.
“FINANCE YIELD Function” on page 676	computes the yield on a security that pays periodic interest.
“FINANCE YIELDDISC Function” on page 677	computes the annual yield for a discounted security (for example, a treasury bill).
“FINANCE YIELDMAT Function” on page 678	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
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.

Category: Financial

Syntax

FINANCE('ACCRINT', *issue*, *first-interest*, *settlement*, *rate*, *par-value*, *frequency*, <*basis*>);

Arguments

issue
specifies the issue date of the security.

first-interest
specifies the first interest date of the security.

settlement
specifies the settlement date.

rate
specifies the interest rate.

par-value
specifies the par value of the security. If you omit *par-value*, SAS uses the value `$1000`.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Accrued Interest: ACCRINT

The following example computes the accrued interest for a security that pays periodic interest.

```
data _null_;
   issue=mdy(2, 27, 1996);
   firstinterest=mdy(8, 31, 1998);
   settlement=mdy(5, 1, 1998);
   rate=0.1;
   par=1000;
   frequency=2;
   basis=1;
   r=finance('accrint', issue, firstinterest,
   settlement, rate, par, frequency, basis);
   put r=;
run;
```

These statements produce this result:

```
r=217.39728
```

FINANCE ACCRINTM Function

Computes the accrued interest for a security that pays interest at maturity.

Category: Financial

Syntax

FINANCE('ACCRINTM', *issue*, *settlement*, *rate*, *par-value*, <*basis*>);

Arguments

issue
specifies the issue date of the security.

settlement
specifies the settlement date.

rate
specifies the interest rate.

par-value
specifies the par value of the security. If you omit *par-value*, SAS uses the value `$1000`.

basis
specifies the optional day count value.

Example: Computing Accrued Interest: ACCRINTM

The following example computes the accrued interest for a security that pays interest at maturity.

```
data _null_;
   issue=mdy(2, 28, 1998);
   maturity=mdy(8, 31, 1998);
   rate=0.1;
   par=1000;
   basis=0;
   r=finance('accrintm', issue, maturity, rate, par, basis);
   put r=;
run;
```

These statements produce this result:

```
r=50.55556
```

FINANCE AMORDEGRC Function

Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.

Category: Financial

Syntax

FINANCE('AMORDEGRC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, <*basis*>);

Arguments

cost
specifies the initial cost of the asset.

date-purchased
specifies the date of the purchase of the asset.

first-period
specifies the date of the end of the first period.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

period
specifies the depreciation period.

rate
specifies the rate of depreciation.

basis
specifies the optional day count value.

> **TIP** When the first argument of the FINANCE function is AMORDEGRC and the value of *basis* is 2, the function returns a missing value.

Example: Computing Depreciation: AMORDEGRC

The following example computes the depreciation for each accounting period by using a depreciation coefficient.

```
data _null_;
   cost=2400;
   datepurchased=mdy(8, 19, 2008);
   firstperiod=mdy(12, 31, 2008);
   salvage=300;
   period=1;
   rate=0.15;
   basis=1;
   r=finance('amordegrc', cost, datepurchased,
   firstperiod, salvage, period, rate, basis);
   put r=;
run;
```

These statements produce this result:

```
r=776
```

FINANCE AMORLINC Function

Computes the depreciation for each accounting period.

Category: Financial

Syntax

FINANCE('AMORLINC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, <*basis*>);

Arguments

cost
specifies the initial cost of the asset.

date-purchased
specifies the date of the purchase of the asset.

first-period
specifies the date of the end of the first period.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

period
specifies the depreciation period.

rate
specifies the rate of depreciation.

basis
specifies the optional day count value.

> **TIP** When the first argument of the FINANCE function is AMORLINC and the value of *basis* is 2, the function returns a missing value.

Example: Computing Description: AMORLINC

The following example computes the depreciation for each accounting period.

```
data _null_;
```

```
   cost=2400;
   dp=mdy(9, 30, 1998);
   fp=mdy(12, 31, 1998);
   salvage=245;
   period=0;
   rate=0.115;
   basis=0;
   r=finance('amorlinc', cost, dp, fp, salvage, period, rate, basis);
   put r=;
run;
```

These statements produce this result:

```
r=69
```

FINANCE COUPDAYBS Function

Computes the number of days from the beginning of the coupon period to the settlement date.

Category: Financial

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.

maturity
specifies the maturity date of the security. The maturity date is the date on which the security expires.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the type of day count basis to use.

Example: Computing Description: COUPDAYBS

The following example computes the number of days from the beginning of the coupon period to the settlement date.

```
data _null_;
   settlement=mdy(12,30,1994);
   maturity=mdy(11,29,1997);
   frequency=4;
   basis=2;
   r=finance('coupdaybs', settlement, maturity, frequency, basis);
   put r=;
run;
```

These statements produce this result:

```
r=31
```

FINANCE COUPDAYS Function

Computes the number of days in the coupon period that contains the settlement date.

Category: Financial

Syntax

FINANCE('COUPDAYS', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: COUPDAYS

The following example computes the number of days in the coupon period that contains the settlement date.

```
data _null_;
   settlement=mdy(1,25,2007);
   maturity=mdy(11,15,2008);
   frequency=2;
   basis=1;
   r=finance('coupdays', settlement, maturity, frequency, basis);
   put r=;
run;
```

These statements produce this result:

```
r=181
```

FINANCE COUPDAYSNC Function

Category: Financial

Syntax

FINANCE('COUPDAYSNC', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: COUPDAYSNC

The following example computes the number of days from the settlement date to the next coupon date.

```
data _null_;
   settlement=mdy(1,25,2007);
   maturity=mdy(11,15,2008);
   frequency=2;
   basis=1;
   r=finance('coupdaysnc', settlement, maturity, frequency, basis);
   put r=;
run;
```

These statements produce this result:

```
r=110
```

FINANCE COUPNCD Function

Category: Financial

Syntax

FINANCE('COUPNCD', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: COUPNCD

The following example computes the next coupon date after the settlement date.

```
data _null_;
   settlement=mdy(1, 25, 2007);
   maturity=mdy(11, 15, 2008);
   frequency=2;
   basis=1;
   r=finance('coupncd', settlement, maturity, frequency, basis);
   put r=date7.;
run;
```

These statements produce this result:

```
r=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.

Category: Financial

Syntax

FINANCE('COUPNUM', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: COUPNUM

The following example computes the number of coupons that are payable between the settlement date and the maturity date.

```
data _null_;
   settlement=mdy(1,25,2007);
   maturity=mdy(11,15,2008);
   frequency=2;
   basis=1;
   r=finance('coupnum', settlement, maturity, frequency, basis);
   put r=;
run;
```

These statements produce this result:

```
r=4
```

FINANCE COUPPCD Function

Computes the previous coupon date before the settlement date.

Category: Financial

Syntax

FINANCE('COUPPCD', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: COUPPCD

The following example computes the previous coupon date before the settlement date.

```
data _null_;
   settlement=mdy(1, 25, 2007);
   maturity=mdy(11, 15, 2008);
   frequency=2;
   basis=1;
   r=finance('couppcd', settlement, maturity, frequency, basis);
   put  settlement;
   put  maturity;
   put r date7.;
run;
```

These statements produce this result:

```
r=11/15/2006
```

FINANCE CUMIPMT Function

Computes the cumulative interest paid between two periods.

Category: Financial

Syntax

FINANCE('CUMIPMT', *rate*, *nper*, *pv*, *start-period*, *end-period*, <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

end-period
specifies the last period in the calculation.

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.

Example: Computing Description: CUMIPMT

The following example computes the cumulative interest that is paid between two periods.

```
data _null_;
   rate=0.09;
   nper=30;
   pv=125000;
   startperiod=13;
   endperiod=24;
   type=0;
   r=finance('cumipmt', rate, nper, pv, startperiod, endperiod, type);
   put r=;
run;
```

These statements produce this result:

```
r=-94054.82033
```

FINANCE CUMPRINC Function

Computes the cumulative principal that is paid on a loan between two periods.

Category: Financial

Syntax

FINANCE('CUMPRINC', *rate*, *nper*, *pv*, *start-period*, *end-period*, <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

end-period
specifies the last period in the calculation.

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.

Example: Computing Description: CUMPRINC

The following example computes the cumulative principal that is paid on a loan between two periods.

```
data _null_;
   rate=0.09;
   nper=30;
   pv=125000;
   startperiod=13;
   endperiod=24;
   type=0;
   r=finance('cumprinc', rate, nper, pv, startperiod, endperiod, type);
   put r=;
run;
```

These statements produce this result:

```
r=-51949.70676
```

FINANCE DB Function

Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

Category: Financial

Syntax

FINANCE('DB', *cost*, *salvage*, *life*, *period*, <*month*>);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

month
specifies the number of months (month is an optional numeric argument). If month is omitted, it defaults to a value of 12.

Example: Computing Description: DB

The following example computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

```
data _null_;
   cost=1000000;
   salvage=100000;
   life=6;
   period=2;
   month=7;
   r=finance('db', cost, salvage, life, period, month);
   put r=;
run;
```

These statements produce this result:

```
r=259639.41667
```

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.

Category: Financial

Syntax

FINANCE('DDB', *cost*, *salvage*, *life*, *period*, <*factor*>);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

Example: Computing Description: DDB

The following example 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 _null_;
   cost=2400;
   salvage=300;
   life=10*365;
   period=1;
   factor=.;
   r=finance('ddb', cost, salvage, life, period, factor);
   put r=;
run;
```

These statements produce this result:

```
r=1.3150684932
```

FINANCE DISC Function

Computes the discount rate for a security.

Category: Financial

Syntax

FINANCE('DISC', *settlement*, *maturity*, *price*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

price
specifies the price of security per $100 face value.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Example: Computing Description: DISC

The following example computes the discount rate for a security.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=0.052420213
```

FINANCE DOLLARDE Function

Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

Category: Financial

Syntax

FINANCE('DOLLARDE', *fractional-dollar*, *fraction*);

Arguments

fractional-dollar
specifies the number expressed as a fraction.

fraction
specifies the integer to use in the denominator of a fraction.

Example: Computing Description: DOLLARDE

The following example converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

```
data _null_;
   fractionaldollar=1.125;
   fraction=16;
   r=finance('dollarde', fractionaldollar, fraction);
   put r=;
run;
```

These statements produce this result:

```
r=1.78125
```

FINANCE DOLLARFR Function

Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

Category: Financial

Syntax

FINANCE('DOLLARFR', *decimal-dollar*, *fraction*);

Arguments

decimal-dollar
specifies a decimal number.

fraction
specifies the integer to use in the denominator of a fraction.

Example: Computing Description: DOLLARFR

The following example converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

```
data _null_;
   decimaldollar=1.125;
   fraction=16;
   r=finance('dollarfr', decimaldollar, fraction);
   put r=;
run;
```

These statements produce this result:

```
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.

Category: Financial

Syntax

FINANCE('DURATION', *settlement*, *maturity*, *coupon*, *yield*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

coupon
specifies the annual coupon rate of the security.

yield
specifies the annual yield of the security.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: DURATION

The following example computes the annual duration of a security with periodic interest payments.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=5.993775
```

FINANCE EFFECT Function

Computes the effective annual interest rate.

Category: Financial

Syntax

FINANCE('EFFECT', *nominal-rate*, *npery*);

Arguments

nominal-rate
specifies a decimal number.

npery
specifies the number of compounding periods per year.

Example: Computing Description: EFFECT

The following example computes the effective annual interest rate.

```
data _null_;
   nominalrate=0.0525;
   npery=4;
   r=finance('effect', nominalrate, npery);
   put r=;
run;
```

These statements produce this result:

```
r=0.053543
```

FINANCE FV Function

Computes the future value of an investment.

Category: Financial

Syntax

FINANCE('FV', *rate*, *nper*, <*payment*>, <*present-value*>, <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

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.

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.

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.

Example: Computing Description: FV

The following example computes the future value of an investment.

```
data _null_;
   rate=0.06/12;
   nper=10;
   payment=-200;
   present-value=-500;
   type=1;
   r=finance('fv', rate, nper, payment, present-value, type);
   put r=;
run;
```

These statements produce this result:

```
r=2581.4033741
```

FINANCE FVSCHEDULE Function

Computes the future value of the initial principal after applying a series of compound interest rates.

Category: Financial

Syntax

FINANCE('FVSCHEDULE', *principal*, *schedule-1*, *schedule-2* ..., *schedule-n*);

Arguments

principal
specifies the present value.

schedule
specifies the sequence of interest rates to apply.

Example: Computing Description: FVSCHEDULE

The following example computes the future value of the initial principal after applying a series of compound interest rates.

```
data _null_;
   principal=1;
   r1=0.09;
   r2=0.11;
   r3=0.1;
   r=finance('fvschedule', principal, r1, r2, r3);
   put r=;
run;
```

These statements produce this result:

```
r=1.33089
```

FINANCE INTRATE Function

Computes the interest rate for a fully invested security.

Category: Financial

Syntax

FINANCE('INTRATE', *settlement*, *maturity*, *investment*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

investment
specifies the amount that is invested in the security.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Example: Computing Description: DISC

The following example computes the discount rate for a security.

```
data _null_;
   settlement=mdy(1, 25, 2007);
   maturity=mdy(6, 15, 2007);
   price=97.975;
   redemption=100;
   basis=1;
   r=finance('intrate', settlement, maturity, investment, redemption,
basis);
   put r=;
run;
```

These statements produce this result:

```
r=0.0535036619
```

FINANCE IPMT Function

Computes the interest payment for an investment for a specified period.

Category: Financial

Syntax

FINANCE('IPMT', *rate*, *period*, *nper*, *pv*, <*fv*>, <*type*>);

Arguments

rate
specifies the interest rate.

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same units as *life*.

nper
specifies the total number of payment periods.

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.

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).

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.

Example: Computing Description: IPMT

The following example computes the interest payment for an investment for a specified period.

```
data _null_;
   rate=0.1/12;
   per=2;
   nper=3;
   pv=100;
   fv=.;
   type=.;
   r=finance('ipmt', rate, per, nper, pv, fv, type);
   put r=;
run;
```

These statements produce this result:

```
r=-0.557857564
```

FINANCE IRR Function

Computes the internal rate of return for a series of cash flows.

Category: Financial

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.

Example: Computing Description: IRR

The following example computes the internal rate of return for a series of cash flows.

```
data _null_;
   v1=-70000;
   v2=12000;
   v3=15000;
   v4=18000;
   v5=21000;
   v6=26000;
   r=finance('irr', v1, v2, v3, v4, v5, v6);
   put r=;
run;
```

These statements produce this result:

```
r=0.086630948
```

FINANCE ISPMT Function

Calculates the interest paid during a specific period of an investment.

Category: Financial

Syntax

FINANCE('ISPMT', *interest-rate*, *period*, *number-payments*, *pv*);

Arguments

interest-rate
specifies the interest rate for the investment.

period
specifies the period for which you want to calculate the interest rate. *Period* must be a value between 1 and *number-payments*.

number-payments
specifies the number of payments for the annuity.

pv
specifies the loan amount or present value of the payments.

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).

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.

Example: Computing Description: ISPMT

The following example 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 ispmt;
   interest=finance('ispmt', 0.075/12, 8, 2*12, 5000);
   put interest=;
run;
```

These statements produce this result:

```
interest= -20.83333333
```

FINANCE MDURATION Function

Computes the Macaulay modified duration for a security with an assumed face value of $100.

Category: Financial

Syntax

FINANCE('MDURATION', *settlement*, *maturity*, *coupon*, *yield*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

coupon
specifies the annual coupon rate of the security.

yield
specifies the annual yield of the security.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: MDURATION

The following example computes the Macaulay modified duration for a security with an assumed face value of $100.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=5.7356698139
```

FINANCE MIRR Function

Computes the internal rate of return where positive and negative cash flows are financed at different rates.

Category: Financial

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.

finance-rate
specifies the interest rate that you pay on the money that is used in the cash flows.

reinvest-rate
specifies the interest rate that you receive on the cash flows as you reinvest them.

Example: Computing Description: MIRR

The following example computes the internal rate of return where positive and negative cash flows are financed at different rates.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=1.3531420172
```

FINANCE NOMINAL Function

Computes the annual nominal interest rates.

Category: Financial

Syntax

FINANCE('NOMINAL', *effective-rate*, *npery*);

Arguments

effective-rate
specifies the effective interest rate.

npery
specifies the number of compounding periods per year.

Example: Computing Description: NOMINAL

The following example computes the annual nominal interest rate.

```
data _null_;
   effectrate=0.08;
   npery=4;
   r= finance('nominal', effectrate, npery);
   put r=;
run;
```

These statements produce this result:

```
r=0.0777061876
```

FINANCE NPER Function

Computes the number of periods for an investment.

Category: Financial

Syntax

FINANCE('NPER', *rate*, *payment*, *pv*, <*fv*>, <*type*>);

Arguments

rate
specifies the interest rate.

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.

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.

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).

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.

Example: Computing Description: NPER

The following example computes the number of periods for an investment.

```
data _null_;
   rate=0.08;
   payment=200;
   pv=1000;
   fv=0;
   type=0;
   r=finance('nper', rate, payment, pv, fv, type);
   put r=;
run;
```

These statements produce this result:

```
r=-4.371981351
```

FINANCE NPV Function

Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

Category: Financial

Syntax

FINANCE('NPV', *rate*, *value-1*, <, ..., value-n>);

Arguments

rate
specifies the interest rate.

value
represents the sequence of the cash flows.

Example: Computing Description: NPV

The following example computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

```
data _null_;
   rate=0.08;
   v1=200;
   v2=1000;
   v3=0.;
   r=finance('npv', rate, v1, v2, v3);
   put r=;
run;
```

These statements produce this result:

```
r=1042.5240055
```

FINANCE ODDFPRICE Function

Computes the price of a security per $100 face value with an odd first period.

Category: Financial

Syntax

FINANCE('ODDFPRICE', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *yield*, *redemption*,*frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

first-coupon
specifies the first coupon date of the security.

rate
specifies the interest rate.

yield
specifies the annual yield of the security.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: ODDFPRICE

The following example computes the price of a security per $100 face value with an odd first period.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=103.94103984
```

FINANCE ODDFYIELD Function

Computes the yield of a security with an odd first period.

Category: Financial

Syntax

FINANCE('ODDFYIELD', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *price*, *redemption*,*frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

first-coupon
specifies the first coupon date of the security.

rate
specifies the interest rate.

price
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: ODDFYIELD

The following example computes the interest of a yield with an odd first period.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=0.06
```

FINANCE ODDLPRICE Function

Computes the price of a security per $100 face value with an odd last period.

Category: Financial

Syntax

FINANCE('ODDLPRICE', *settlement*, *maturity*, *last-interest*, *rate*, *yield*, *redemption*,*frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

last-interest
specifies the last coupon date of the security.

rate
specifies the interest rate.

yield
specifies the annual yield of the security.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: ODDLPRICE

The following example computes the price of a security per $100 face value with an odd last period.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=99.878286015
```

FINANCE ODDLYIELD Function

Computes the yield of a security with an odd last period.

Category: Financial

Syntax

FINANCE('ODDLYIELD', *settlement*, *maturity*, *last-interest*, *rate*, *price*, *redemption*,*frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

last-interest
specifies the last coupon date of the security.

rate
specifies the interest rate.

price
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Details

There is an explicit check in the code for the *LastInterest* date to be on or before the *settlement* date. The *settlement* date must be on or before the *maturity* date. Otherwise, the result is missing.

Example: Computing Description: ODDLYIELD

The following example computes the yield of a security with an odd last period.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=0.0405
```

FINANCE PMT Function

Computes the periodic payment of an annuity.

Category: Financial

Syntax

FINANCE('PMT', *rate*, *nper*, *pv*, <*fv*>, <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

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.

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).

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.

Example: Computing Description: PMT

The following example computes the periodic payment for an annuity.

```
data _null_;
   rate=0.08;
   nper=5;
   pv=91;
   fv=3;
   type=0;
   r=finance('pmt', rate, nper, pv, fv, type);
   put r=;
run;
```

These statements produce this result:

```
r=-23.30290673
```

FINANCE PPMT Function

Computes the payment on the principal for an investment for a specified period.

Category: Financial

Syntax

FINANCE('PPMT', *rate*, *period*, *nper*, *pv*, <*fv*>, <*type*>);

Arguments

rate
specifies the interest rate.

period
specifies the period.

Range: 1–*nper*

nper
specifies the number of payment periods.

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.

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).

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.

Example: Computing Description: PPMT

The following example computes the payment on the principal for an investment for a specified period.

```
data _null_;
   rate=0.08;
```

```
   period=10;
   nper=10;
   pv=200000;
   fv=0;
   type=0;
   r=finance('ppmt', rate, period, nper, pv, fv, type);
   put r=;
run;
```

These statements produce this result:

```
r=-27598.05346
```

FINANCE PRICE Function

Computes the price of a security per $100 face value that pays periodic interest.

Category: Financial

Syntax

FINANCE('PRICE', *settlement*, *maturity*, *rate*, *yield*, *redemption*,*frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

rate
specifies the interest rate.

yield
specifies the annual yield of the security.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: PRICE

The following example computes the price of a security per $100 face value that pays periodic interest.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=94.634361621
```

FINANCE PRICEDISC Function

Computes the price of a discounted security per $100 face value.

Category: Financial

Syntax

FINANCE('PRICEDISC', *settlement*, *maturity*, *discount*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

discount
specifies the discount rate of the security.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Example: Computing Description: PRICEDISC

The following example computes the price of a discounted security per $100 face value.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=48.8125
```

FINANCE PRICEMAT Function

Computes the price of a security per $100 face value that pays interest at maturity.

Category: Financial

Syntax

FINANCE('PRICEMAT', *settlement*, *maturity*, *issue*, *rate*, *yield*,<*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

rate
specifies the interest rate.

yield
specifies the annual yield of the security.

basis
specifies the optional day count value.

Example: Computing Description: PRICEMAT

The following example computes the price of a security per $100 face value that pays interest at maturity.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=99.98449888
```

FINANCE PV Function

Computes the present value of an investment.

Category: Financial

Syntax

FINANCE('PV', *rate*, *nper*, *payment*, <*fv*>, <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

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.

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).

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.

Example: Computing Description: PV

The following example computes the present value of an investment.

```
data _null_;
   rate=0.05;
   nper=10;
   payment=1000;
   fv=200;
   type=0;
   r=finance('pv', rate, nper, payment, fv, type);
   put r=;
run;
```

These statements produce this result:

```
r=-7844.51758
```

FINANCE RATE Function

Computes the interest rate per period of an annuity.

Category: Financial

Syntax

FINANCE('RATE', *nper*, *payment*, *pv*, <*fv*>, <*type*>);

Arguments

nper
specifies the total number of payment periods.

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.

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.

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).

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.

Example: Computing Description: RATE

The following example computes the interest rate per period of an annuity.

```
data _null_;
   nper=4;
   payment=-2481;
   pv=8000;
   r=finance('rate', nper, payment, pv);
   put r=;
run;
```

These statements produce this result:

```
r=0.0921476841
```

FINANCE RECEIVED Function

Computes the amount that is received at maturity for a fully invested security.

Category: Financial

Syntax

FINANCE('RECEIVED', *settlement*, *maturity*, *investment*, *discount*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

investment
specifies the amount that is invested in the security.

discount
specifies the discount rate of the security.

basis
specifies the optional day count value.

Example: Computing Description: RECEIVED

The following example computes the amount that is received at maturity for a fully invested security.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=1014584.6544
```

FINANCE SLN Function

Computes the straight-line depreciation of an asset for one period.

Category: Financial

Syntax

FINANCE('SLN', *cost*, *salvage*, *life*);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Example: Computing Description: SLN

The following example computes the straight-line depreciation of an asset for one period.

```
data _null_;
   cost=2000;
   salvage=200;
   life=11;
   r=finance('sln', cost, salvage, life);
   put r=;
run;
```

These statements produce this result:

```
r=163.63636364
```

FINANCE SYD Function

Computes the sum-of-years digits depreciation of an asset for a specified period.

Category: Financial

Syntax

FINANCE('SYD', *cost*, *salvage*, *life*, *period*);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

period
specifies a period in the same time units that are used for the argument *life*.

Example: Computing Description: SYD

The following example computes the sum-of-years digits depreciation of an asset for a specified period.

```
data _null_;
   cost=2000;
   salvage=200;
   life=11;
   period=1;
   r=finance('syd', cost, salvage, life, period);
   put r=;
run;
```

These statements produce this result:

```
r=300
```

FINANCE TBILLEQ Function

Computes the bond-equivalent yield for a treasury bill.

Category: Financial

Syntax

FINANCE('TBILLEQ', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

discount
specifies the discount rate of the security.

Example: Computing Description: TBILLEQ

The following example computes the bond-equivalent yield for a treasury bill.

```
data _null_;
   settlement=mdy(3, 31, 2008);
   maturity=mdy(6, 1, 2008);
   discount=0.0914;
   r=finance('tbilleq', settlement, maturity, discount);
   put r=;
run;
```

These statements produce this result:

```
r=0.0941514936
```

FINANCE TBILLPRICE Function

Computes the price of a treasury bill per $100 face value.

Category: Financial

Syntax

FINANCE('TBILLPRICE', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

discount
specifies the discount rate of the security.

Example: Computing Description: TBILLPRICE

The following example computes the price of a treasury bill per $100 face value.

```
data _null_;
   settlement=mdy(3, 31, 2008);
   maturity=mdy(6, 1, 2008);
   discount=0.09;
   r=finance('tbillprice', settlement, maturity, discount);
   put r=;
run;
```

These statements produce this result:

```
r=98.45
```

FINANCE TBILLYIELD Function

Computes the yield for a treasury bill.

Category: Financial

Syntax

FINANCE('TBILLYIELD', *settlement*, *maturity*, *price*);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

price
specifies the price of the security per $100 face value.

Example: Computing Description: TBILLYIELD

The following example computes the yield for a treasury bill.

```
data _null_;
   settlement=mdy(3, 31, 2008);
   maturity=mdy(6, 1, 2008);
   price=98;
   r=finance('tbillyield', settlement, maturity, price);
   put r=;
```

```
run;
```

These statements produce this result:

```
r=0.1184990125
```

FINANCE VDB Function

Computes the depreciation of an asset for a specified or partial period by using a declining balance method.

Category: Financial

Syntax

FINANCE('VDB', *cost*, *salvage*, *life*, *start-period*, *end-period*, <*factor*>, <*noswitch*>);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

end-period
specifies the last period in the calculation.

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

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.

Example: Computing Description: VDB

The following example computes the depreciation of an asset for a specified or partial period by using a declining balance method.

```
data _null_;
   cost=2400;
   salvage=300;
   life=10;
   startperiod=0;
   endperiod=1;
   factor=1.5;
   r=finance('vdb', cost, salvage, life, startperiod, endperiod,
factor);
   put r=;
run;
```

These statements produce this result:

```
r=360
```

FINANCE XIRR Function

Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

Category: Financial

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.

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.

guess
specifies an optional number that you guess is close to the result of XIRR.

Example: Computing Description: XIRR

The following example computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=0.3733625335
```

FINANCE XNPV Function

Computes the net present value for a schedule of cash flows that is not necessarily periodic.

Category: Financial

Syntax

FINANCE('XNPV', *rate*, *values*, *dates*);

Arguments

rate
specifies the interest rate.

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.

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.

Example: Computing Description: XNPV

The following example computes the net present value for a schedule of cash flows that is not necessarily periodic.

```
data _null_;
   r=.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', r, v1, v2, v3, v4, v5, d1, d2, d3, d4, d5);
   put r=;
run;
```

These statements produce this result:

```
r=2086.647602
```

FINANCE YIELD Function

Computes the yield on a security that pays periodic interest.

Category: Financial

Syntax

FINANCE('YIELD', *settlement*, *maturity*, *rate*, *price*, *redemption*,*frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

rate
specifies the interest rate.

price
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency*=1; for semiannual payments, *frequency*=2; for quarterly payments, *frequency*=4.

basis
specifies the optional day count value.

Example: Computing Description: YIELD

The following example computes the yield on a security that pays periodic interest.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=0.0650000069
```

FINANCE YIELDDISC Function

Computes the annual yield for a discounted security (for example, a treasury bill).

Category: Financial

Syntax

FINANCE('YIELDDISC', *settlement*, *maturity*, *price*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

price
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Example: Computing Description: YIELDDISC

The following example computes the annual yield for a discounted security (for example, a treasury bill).

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
r=0.0059607748
```

FINANCE YIELDMAT Function

Computes the annual yield of a security that pays interest at maturity.

Category: Financial

Syntax

FINANCE('YIELDMAT', *settlement*, *maturity*, *issue*, *rate*, *price*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

rate
specifies the interest rate.

price
specifies the price of the security per $100 face value.

basis
specifies the optional day count value.

Example: Computing Description: YIELDMAT

The following example computes the annual yield of a security that pays interest at maturity.

```
data _null_;
   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=;
run;
```

These statements produce this result:

```
0.0609543337
```

FIND Function

Searches for a specific substring of characters within a character string.

Category: Character

Restriction: This function is assigned an I18N Level 0 status and is designed for SBCS data. However, if the first argument, *string*, is specified as varchar and there are multi-byte characters, then the FIND function processes the multi-byte data. For more information, see "Internationalization Compatibility for SAS String Functions" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Tip: Use the "KINDEX Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench* instead to write encoding independent code.

Syntax

FIND(*string*, *substring* <, *modifier(s)*> <, *start-position*>)

FIND(*string*, *substring* <, *start-position*> <, *modifier(s)*>)

Required Arguments

string
specifies a character constant, variable, or expression that is searched for substrings.

Tip Enclose a literal string of characters in quotation marks.

substring
is a character constant, variable, or expression that specifies the substring of characters to search for in *string*.

Tip Enclose a literal string of characters in quotation marks.

Optional Arguments

modifier
is a character constant, variable, or expression that specifies one or more modifiers. The following modifiers are valid:

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 one character argument instead of both (or all) character arguments, use the TRIM function instead of the FIND function with the T modifier.

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.

start-position
is a numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction of the search.

Details

Basics

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 *start-position* is not specified, FIND starts the search at the beginning of the *string* and searches the *string* from left to right. If *start-position* is specified, the absolute value of *start-position* determines the position at which to start the search. The sign of *start-position* determines the direction of the search.

Value of *startpos*	**Action**
greater than 0	starts the search at position *start-position* and the direction of the search is to the right. If *start-position* is greater than the length of *string*, FIND returns a value of 0.
less than 0	starts the search at position –*start-position* and the direction of the search is to the left. If –*start-position* is greater than the length of *string*, the search starts at the end of *string*.
equal to 0	returns a value of 0.

Processing SBCS and DBCS Data

The FIND function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria:

- If *string* is not declared as varchar and you are processing single-byte data, then FIND processes SBCS.
- If *string* is declared as varchar and you are processing multi-byte data, then FIND processes DBCS.

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 *modifier* nor the *start-position* arguments.

Example

```
data
one;

   whereisshe=find('She sells seashells? Yes, she does.','she
');

   put
whereisshe=;



   variable1='She sells seashells? Yes, she
does.';

   variable2='she
';


variable3='i';


whereisshe_i=find(variable1,variable2,variable3);

   put
whereisshe_i=;



   expression1='She sells seashells? '||'Yes, she
does.';

   expression2=kscan('he or she',3)||'
';

   expression3=trim('t
');


whereisshe_t=find(expression1,expression2,expression3);

   put
whereisshe_t=;



   xyz='She sells seashells? Yes, she
does.';


startposvar=22;


whereisshe_22=find(xyz,'she',startposvar);
```

```
   put
whereisshe_22=;


   xyz='She sells seashells? Yes, she
does.';


startposexp=1-23;


whereisShe_ineg22=find(xyz,'She','i',startposexp);

   put
whereisShe_ineg22=;

run;
```

These statements produce these results:

```
whereisshe=27
whereisshe_i=1
whereisshe_t=14
whereisshe_22=27
whereisShe_ineg22=14
```

See Also

Functions:

- “COUNT Function” on page 488
- “FINDC Function” on page 683
- “FINDW Function” on page 692
- “INDEX Function” on page 947

FINDC Function

Searches a string for any character in a list of characters.

Category: Character

Restriction: This function is assigned an I18N Level 0 status and is designed for SBCS data. However, if the first argument, *string*, is specified as varchar and there are multi-byte characters, then the FINDC function processes the multi-byte data. For more information, see “Internationalization Compatibility for SAS String Functions” in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Tip: Use the "KINDEXC Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench* instead to write encoding independent code.

Syntax

FINDC(*string* <, *character-list*>)

FINDC(*string*, *character-list* <, *modifier(s)*>)

FINDC(*string*, *character-list*, *modifier(s)* <, *start-position*>)

FINDC(*string*, *character-list*, <*start-position*>, <*modifier(s)*>)

Required Arguments

string
is a character constant, variable, or expression that specifies the character string to be searched.

Tip Enclose a literal string of characters in quotation marks.

character-list
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.

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, of the *start-position* argument.

Note: If the modifier is specified as b or B, then the *start-position* argument needs to be negative or do not specify a *start-position*.

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. This modifier has the same functionality as the *v or V* modifier.

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 the DATA step (excluding WHERE clauses), or in the SQL procedure can make FINDC run faster when you call it in a loop where the *character-list* 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 *character-list* arguments.

Note: 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
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. This modifier has the same functionality as the *k or K* modifier.

w or W
adds printable characters to the list of characters.

x or X
adds hexadecimal characters to the list of characters.

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.

Optional Argument

start-position
is an optional numeric constant, variable, or expression having an integer value that specifies the position at which the search should start and the direction in which to search.

Details

Basics

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 *start-position* 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 *start-position* is specified, the absolute value of *start-position* 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 *start-position* 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 *start-position* and proceeds to the right. If *start-position* is greater than the length of the string, FINDC returns a value of 0.
less than 0	search begins at position –*start-position* and proceeds to the left. If *start-position* is less than the negative of the length of the string, the search begins at the end of the string.

Value of *startpos*	Action
equal to 0	returns a value of 0.

Processing SBCS and DBCS Data

The FINDC function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria:

- If *string* is not declared as varchar and you are processing single-byte data, then FINDC processes SBCS.
- If *string* is declared as varchar and you are processing multi-byte data, then FINDC processes DBCS.

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 *start-position* 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 *start-position* arguments.

Examples

Example 1: Searching for Characters in a String

This example searches a character string and returns the characters that are found.

```
data _null_;
   string='Hi, ho!';
   charlist='hi';
   j=0;
   do until (j=0);
      j=findc(string, charlist, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=2 c=i
j=5 c=h
That's all
```

Example 2: Searching for Characters in a String and Ignoring Case

This example searches a character string and returns the characters that are found. The I modifier is used to ignore the case of the characters.

```
data _null_;
   string='Hi, ho!';
   charlist='ho';
   j=0;
   do until (j=0);
      j=findc(string, charlist, j+1, "i");
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=1 c=H
j=5 c=h
j=6 c=o
That's all
```

Example 3: Searching for Characters and Using the K Modifier

This example searches a character string and returns the characters that do not appear in the character list.

```
data _null_;
   string='Hi, ho!';
   charlist='hi';
   j=0;
   do until (j=0);
      j=findc(string, charlist, "k", j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=!
That's all
```

Example 4: Searching for the Characters h, i, and Blank

This example 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 _null_;
   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;
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

This example searches for the four characters h, i, H, and I. FINDC with the i modifier ignores character case during the search.

```
data _null_;
   whereishi_i=0;
   do until(whereishi_i=0);
      variable1='Hi there, Ian!';
      variable2='hi';
      variable3='i';
      whereishi_i=findc(variable1, variable2, variable3, whereishi_i
+1);
      if whereishi_i=0 then put "The End";
      else do;
         whatfound=substr(variable1, whereishi_i, 1);
         put whereishi_i= whatfound=;
      end;
   end;
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

This example 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 _null_;
   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;
run;
```

SAS writes the following 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

This example 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 _null_;
   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;
run;
```

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 147
- “ANYALPHA Function” on page 150
- “ANYCNTRL Function” on page 152
- “ANYDIGIT Function” on page 154
- “ANYGRAPH Function” on page 158
- “ANYLOWER Function” on page 161
- “ANYPRINT Function” on page 165
- “ANYPUNCT Function” on page 168
- “ANYSPACE Function” on page 170
- “ANYUPPER Function” on page 173
- “ANYXDIGIT Function” on page 175
- “COUNTC Function” on page 491
- “INDEXC Function” on page 949
- “NOTALNUM Function” on page 1142
- “NOTALPHA Function” on page 1144
- “NOTCNTRL Function” on page 1147
- “NOTDIGIT Function” on page 1149
- “NOTGRAPH Function” on page 1156
- “NOTLOWER Function” on page 1159
- “NOTPRINT Function” on page 1163
- “NOTPUNCT Function” on page 1167
- “NOTSPACE Function” on page 1169
- “NOTUPPER Function” on page 1172
- “NOTXDIGIT Function” on page 1174
- “VERIFY Function” on page 1512

FINDW Function

Returns the character position of a word in a string, or returns the number of the word in a string.

Category: Character

Restriction: This function is assigned an I18N Level 1 status and is designed for SBCS data. However, if the first argument, *string*, has multi-byte characters, then the FINDW function processes the DBCS data. For more information, see "Internationalization Compatibility for SAS String Functions" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Syntax

FINDW(*string*, *word* <, *character(s)*>)

FINDW(*string*, *word*, *character(s)*, *modifier(s)* <, *start-position*>)

FINDW(*string*, *word*, *character(s)*, *start-position* <, *modifier(s)*>)

FINDW(*string*, *word*, *start-position* <, *character(s)* <, *modifier(s)*>>)

Required Arguments

string
is a character constant, variable, or expression that specifies the character string to be searched.

word
is a character constant, variable, or expression that specifies the word to search for in *string*.

character
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.

start-position
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should begin and the direction in which to search.

modifier
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the FINDW function.

> **TIP** If you use the *modifier* argument, then it must be positioned after the *character* argument.

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
scans from right to left instead of from left to right, regardless of the sign of the *start-position* 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 a word 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 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.

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 *character* and *modifier* arguments only once, rather than every time the FINDW function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make FINDW run faster when you call it in a loop where the *character* 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 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 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 *character* 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.

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 *character* 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 *start-position* 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 *start-position* argument, then the absolute value of *start-position* specifies the position at which to begin the search. The sign of *start-position* specifies the direction in which to search:

Value of *startpos*	Action
greater than 0	search begins at position *start-position* and proceeds to the right. If *start-position* is greater than the length of the string, then FINDW returns a value of 0.
less than 0	search begins at position –*start-position* and proceeds to the left. If *start-position* 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 *start-position* 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 *start-position* argument, then FINDW searches from right to left starting at the end of the string.

Using the FINDW Function in an ASCII Environment

If you use the FINDW function with only two arguments and your computer uses ASCII, these are the default delimiters:

blank ! $ % & ( ) * + , - . / ; < ^ |

In ASCII environments that do not contain the ^ character, the FINDW function uses the ~ character instead.

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.

Processing SBCS and DBCS Data

The FINDW function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria:

- If *string* is not declared as varchar and you are processing single-byte data, then FINDW processes SBCS.
- If *string* is declared as varchar and you are processing multi-byte data, then FINDW processes DBCS.

Using FINDW With %SYSFUNC

When using FINDW with the macro %SYSFUNC, use the fourth argument to specify the delimiter as a blank.

```
%LET V = %UPCASE(ISO);
%LET K = %UPCASE(SAS ISO EXCEL RAW);

%LET X = %SYSFUNC(FINDw(%str(&k), %str(&v), ,ES));
%put &=x;
```

```
X=2
```

Examples

Example 1: Searching a Character String for a Word

The following example searches a character string for the word "she", and returns the position of the beginning of the word.

```
data _null_;
   whereisshe=findw('She sells sea shells? Yes, she does.', 'she');
   put whereisshe=;
run;
```

SAS writes the following output to the log:

```
whereisshe=28
```

Example 2: Searching a Character String and Using the Character and Start-position Arguments

The following example contains two occurrences of the word "rain." Only the second occurrence is found by FINDW because the search begins in position 25. The *character* argument specifies a space as the delimiter.

```
data _null_;
   result=findw('At least 2.5 meters of rain falls in a rain forest.',
                  'rain', ' ', 25);
   put result=;
run;
```

SAS writes the following output to the log:

```
result=40
```

Example 3: Searching a Character String and Using the I Modifier and the Start-position Argument

The following example uses the I modifier and returns the position of the beginning of the word. The I modifier disregards case, and the *start-position* argument identifies the starting position from which to search.

```
data _null_;
   string='Artists from around the country display their art at
           an art festival.';
   result=findw(string, 'Art', ' ', 'i', 10);
   put result=;
run;
```

SAS writes the following output to the log:

```
result=47
```

Example 4: Searching a Character String and Using the E Modifier

The following example uses the E modifier and returns the number of complete words that are scanned while searching for the word "art."

```
data _null_;
   string='Artists from around the country display their art at
           an art festival.';
   result=findw(string, 'art', ' ', 'E');
   put result=;
run;
```

SAS writes the following output to the log:

```
result=8
```

Example 5: Searching a Character String and Using the E Modifier and the Start-position Argument

The following example 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_;
   string='Artists from around the country display their art at
           an art festival.';
```

```
   result=findw(string, 'art', ' ', 'E', 50);
   put result=;
run;
```

SAS writes the following output to the log:

```
result=3
```

Example 6: Searching a Character String and Using Two Modifiers

The following example uses the I and the E modifiers to find a word in a string.

```
data _null_;
   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=;
run;
```

SAS writes the following output to the log:

```
result=5
```

Example 7: Searching a Character String and Using the R Modifier

The following example uses the R modifier to remove leading and trailing delimiters from a word.

```
data _null_;
    string='Artists from around the country display their art at
                an art festival.';
    word='  art  ';
    result=findw(string, word, ' ', 'R');
    put result=;
run;
```

SAS writes the following output to the log:

```
result=47
```

See Also

Functions:

- “COUNTW Function” on page 495
- “FIND Function” on page 679
- “FINDC Function” on page 683

- "INDEXW Function" on page 951
- "SCAN Function" on page 1366

CALL Routines:

- "CALL SCAN Routine" on page 326

FINFO Function

Returns the value of a file information item.

Category: External Files

Note: FINFO returns the value of a system-dependent information item for an external file. FINFO returns a blank if the value given for *information-item* is invalid.

Syntax

FINFO(*file-id*, *information-item*)

Required Arguments

file-id
is a numeric constant, variable, or expression that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

information-item
is a character constant, variable, or expression that specifies the name of the file information item to be retrieved.

- Create Time: *ddmmmyyyy:hh:mm:ss*

  Note: The Create Time date/time information item is localized to the site's locale. The date/time format might appear slightly different in the locale.

- Last Modified: *ddmmmyyyy:hh:mm:ss*
- Filename
- File size (bytes)
- RECFM
- LRECL

Information-item for pipe files can be one of these file information items:

- Unnamed pipe access device
- PROCESS
- RECFM

- LRECL

Linux Specifics: *Information-item* for disk files can have one of the following values:

- Filename
- Owner Name
- Group Name
- Access Permission
- Last Modified
- File Size (bytes)

If you concatenate filenames, then an additional information-item, File List, is available.

If you are using pipe files, then the only valid value for information-item is PIPE Command.

Comparisons

- The FOPTNAME function determines the names of the available file information items.
- The FOPTNUM function determines the number of system-dependent information items that are available.

Examples

Example 1

This example stores information items about an external file in a SAS data set:

```
data info;
   length infoname infoval $60;
   drop rc fid infonum i close;
   rc=filename('abc', 'physical-filename');
   fid=fopen('abc');
   infonum=foptnum(fid);
   do i=1 to infonum;
      infoname=foptname(fid, i);
      infoval=finfo(fid, infoname);
      output;
   end;
   close=fclose(fid);
run;
```

Example 2: Linux

This example shows the information items available for Linux System Services files:

```
data _null_;
   length opt $100 optval $100;
   /* Allocate file  */
   rc=FILENAME('myfile',
      '/u/userid/one');
   /* Open file */
   fid=FOPEN('myfile');
   /* Get number of information
      items */
   infocnt=FOPTNUM(fid);
   /* Retrieve information items
      and print to log */
   put @1 'Information for a UNIX System Services File:';
   do j=1 to infocnt;
      opt=FOPTNAME(fid,j);
      optval=FINFO(fid,upcase(opt));
      put @1 opt @20 optval;
   end;
   /* Close the file */
   rc=FCLOSE(fid);
   /* Deallocate the file */
   rc=FILENAME('myfile');
run;
```

```
Information for a LINUX
   System Services File:
File Name          /u/userid/one
Access Permission  -rw-rw-rw-
Number of Links    1
Owner Name         USERID
Group Name         GRP
File Size          4
Last Modified      13Jun2017:09:37:02
Created            Mar 16 09:55
NOTE: The DATA statement used
0.07 CPU seconds and 5227K.
```

See Also

Functions:

- “FCLOSE Function” on page 591
- “FOPTNUM Function” on page 726
- “MOPEN Function” on page 1121

FINV Function

Returns a quantile from the *F* distribution.

Category: Quantile

Syntax

FINV (*p*, *ndf*, *ddf* <, *nc*>)

Required Arguments

p
is a numeric probability.

Range $0 \leq p < 1$

ndf
is a numeric numerator degrees of freedom parameter.

Range $ndf > 0$

ddf
is a numeric denominator degrees of freedom parameter.

Range $ddf > 0$

Optional Argument

nc
is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Details

The FINV function returns the pth quantile from the *F* distribution with numerator degrees of freedom *ndf*, denominator degrees of freedom *ddf*, and noncentrality parameter *nc*. The probability that an observation from the *F* distribution is less than the quantile is *p*. This function accepts noninteger degrees of freedom parameters *ndf* and *ddf*.

If the optional parameter *nc* is not specified or has the value 0, the quantile from the central *F* distribution is returned. The noncentrality parameter *nc* is defined

such that if X and Y are normal random variables with means μ and 0, respectively, and variance 1, then X^2/Y^2 has a noncentral *F* distribution with *nc* = μ^2.

CAUTION
For large values of *nc,* the algorithm could fail. In that case, a missing value is returned.

Note: FINV is the inverse of the PROBF function.

Example

These statements compute the 95th quantile value of a central *F* distribution with 2 and 10 degrees of freedom and a noncentral *F* distribution with 2 and 10.3 degrees of freedom and a noncentrality parameter equal to 2:

```
data
one;

   q1=finv(.95, 2,
10);

   q2=finv(.95, 2, 10.3,
2);

   put q1=
q2=;

run;
```

These statements produce these results:

```
q1=4.1028210151 q2=7.583766024
```

See Also

Functions:

- "QUANTILE Function" on page 1293

FIPNAME Function

Converts two-digit FIPS codes to uppercase state names.

Category: State and ZIP code

Syntax

FIPNAME(*expression*)

Required Argument

expression
specifies a numeric constant, variable, or expression that represents a U.S. FIPS code.

Details

If the FIPNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPNAME function converts a U.S. Federal Information Processing Standards (FIPS) code to the corresponding state or U.S. territory name in uppercase, returning a value of up to 20 characters.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or worldwide GSA geographic code for U.S. territories) in uppercase.

Example

```
data
one;


x=fipname(37);

   put
x=;




x=fipnamel(37);

   put
x=;
```

```
x=fipstate(37);

   put
x=;

run;
```

These statements produce these results:

```
x=NORTH CAROLINA
x=North Carolina
x=NC
```

See Also

Functions:

- “FIPNAMEL Function” on page 705
- “FIPSTATE Function” on page 707
- “STFIPS Function” on page 1410
- “STNAME Function” on page 1412
- “STNAMEL Function” on page 1414

FIPNAMEL Function

Converts two-digit FIPS codes to mixed case state names.

Category: State and ZIP code

Syntax

FIPNAMEL(*expression*)

Required Argument

expression
specifies a numeric constant, variable, or expression that represents a U.S. FIPS code.

Details

If the FIPNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPNAMEL function converts a U.S. Federal Information Processing Standards (FIPS) code to the corresponding state or U.S. territory name in mixed case, returning a value of up to 20 characters.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or worldwide GSA geographic code for U.S. territories) in uppercase.

Example

The examples show the differences when using FIPNAME, FIPNAMEL, and FIPSTATE:

```
data
one;


x=fipname(37);

   put
x=;




x=fipnamel(37);

   put
x=;




x=fipstate(37);

   put
x=;

run;
```

These statements produce these results:

```
x=NORTH CAROLINA
x=North Carolina
x=NC
```

See Also

Functions:

- “FIPNAME Function” on page 703
- “FIPSTATE Function” on page 707
- “STFIPS Function” on page 1410
- “STNAME Function” on page 1412
- “STNAMEL Function” on page 1414

FIPSTATE Function

Converts two-digit FIPS codes to two-character state postal codes.

Category: State and ZIP code

Syntax

FIPSTATE(*expression*)

Required Argument

expression
specifies a numeric constant, variable, or expression that represents a U.S. FIPS code.

Details

If the FIPSTATE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPSTATE function converts a U.S. Federal Information Processing Standards (FIPS) code to a two-character state postal code (or worldwide GSA geographic code for U.S. territories) in uppercase.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or worldwide GSA geographic code for U.S. territories) in uppercase.

Example

The examples show the differences when using FIPNAME, FIPNAMEL, and FIPSTATE:

```
data
one;


x=fipname(37);

   put
x;




x=fipnamel(37);

   put
x;




x=fipstate(37);

   put
x;

run;
```

These statements produce these results:

```
x=NORTH CAROLINA
x=North Carolina
x=NC
```

See Also

Functions:

- “FIPNAME Function” on page 703

- “FIPNAMEL Function” on page 705
- “STFIPS Function” on page 1410
- “STNAME Function” on page 1412
- “STNAMEL Function” on page 1414

FIRST Function

Returns the first character in a character string.

Category: Character

Syntax

FIRST(*string*)

Required Argument

string
specifies a character string.

Details

In a DATA step, the default length of the target variable for the FIRST function is 1 character.

The FIRST function returns a string with a length of 1. If *string* has a length of 0, then the FIRST function returns a single blank.

Comparisons

The FIRST function returns the same result as CHAR(*string*, 1) and SUBPAD(*string*, 1, 1). Although the results are the same, the default length of the target variable is different.

Example

The following example shows the results of using the FIRST function.

```
data test;
   string1="abc";
```

```
   result1=first(string1);
   string2="";
   result2=first(string2);
run;
proc print noobs data=test;
run;
```

Output 3.34 *Output from the FIRST Function*

The SAS System

string1	result1	string2	result2
abc	a		

See Also

Functions:

- "CHAR Function" on page 429

FLOOR Function

Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results.

Category: Rounding and Truncation

Syntax

FLOOR(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

If the argument is within 1E-12 of an integer, the function returns that integer.

Comparisons

Unlike the FLOORZ function, the FLOOR function fuzzes the result. 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

```
data
one;


var1=2.1;


a=floor(var1);

   put
a=;




var2=-2.4;


b=floor(var2);

   put
b=;




c=floor(-1.6);

   put
c=;




var6=(1.-1.e-13);


d=floor(1-1.e-13);

   put
d=;
```

```
e=floor(763);

   put
e=;



f=floor(-223.456);

   put
f=;

run;
```

These statements produce these results:

```
a=2
b=-3
c=-2
d=1
e=763
f=-224
```

See Also

Functions:

- “FLOORZ Function” on page 712

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Category: Rounding and Truncation

Syntax

FLOORZ(*argument*)

Required Argument

argument
is a numeric constant, variable, or expression.

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

```
data
one;


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=;

run;
```

These statements produce these results:

```
a=2
b=-3
c=-2
d=0
e=763
f=-224
```

See Also

Functions:

- “FLOOR Function” on page 710

FMTINFO Function

Retrieves information about a format or informat.

Category: 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(*fmtname*, *info*)

Required Arguments

fmtname
specifies the format or informat name.

info
specifies the category, type, or description of the format or informat.

Note: You can specify only one *info* argument in your code.

CAT
specifies the category:

- BIDI Text Handling
- Character
- Currency Conversion
- Date and Time
- DBCS
- Hebrew Text Handling
- ISO 8601
- Numeric

TYPE
specifies the type of language element:

- informat
- format
- both

DESC
specifies a short description of the format or informat.

MIND
specifies the minimum decimal value of the format or informat.

MAXD
specifies the maximum decimal value of the format or informat.

DEFD
specifies the default decimal value of the format or informat.

MINW
specifies the minimum width value of the format or informat.

MAXW
specifies the maximum width value of the format or informat.

DEFW
specifies the default width value of the format or informat.

Details

The FMTINFO function retrieves information about a format or informat. You can retrieve information about a format or informat's category, the type of language

element (format, informat, or both), 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. You can specify only one of these arguments: CAT, TYPE, DESC, MIND, MAXD, DEFD, MINW, MAXW, and DEFW. For example,

```
a=fmtinfo('best',cat);
```

is valid because you specify only one argument, the category argument `cat`.

You must enclose the arguments with apostrophes unless you are passing a character variable name.

The FMTINFO function returns a character string for all data values, including these numeric value arguments: MIND, MAXD, DEFD, MINW, MAXW, and DEFW.

Do not include a length value when specifying a format or informat. For example, the **`Date`** format should include only the name of the format: `a=fmtinfo('date', 'type');`

Example

The following examples retrieve information about the BEST format.

```
data
one;

   a=fmtinfo('best','cat');
   put a=;
   b=fmtinfo('best','type');
   put b=;
   c=fmtinfo('best','desc');
   put c=;
run;
```

These statements produce these results:

```
a=num
b=BOTH
c=SAS System chooses best notation
```

FNONCT Function

Returns the value of the noncentrality parameter of an F distribution.

Category: Mathematical

Syntax

FNONCT(*x*, *ndf*, *ddf*, *probability*)

Required Arguments

x
is a numeric random variable.

Range $x \geq 0$

ndf
is a numeric numerator degree of freedom parameter.

Range $ndf > 0$

ddf
is a numeric denominator degree of freedom parameter.

Range $ddf > 0$

probability
is a probability.

Range $0 < probability < 1$

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:

$$I_x(a,b) = \frac{1}{\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

```
data work;
   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;
run;
proc print;
run;
```

Output 3.35 *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

FNOTE Function

Identifies the last record that was read, and returns a value that the FPOINT function can use.

Category: External Files

Syntax

FNOTE(*file-id*)

Required Argument

file-id
is a numeric constant, variable, or expression that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

You can use FNOTE like a bookmark, marking the position in the file so that your application can later return to that position using FPOINT. The value that is returned by FNOTE is required by the FPOINT function to reposition the file pointer on a specific record.

To free the memory associated with each FNOTE identifier, use DROPNOTE.

Note: You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, indicated by a positive value in the variable FID, then it reads the records, stores in the variable NOTE 3 the position of the third record read, and then later uses FPOINT to point back to NOTE3 to update the file. After updating the record, it closes the file:

```
%macro test;
%let fref=MYFILE;
%let rc=%sysfunc(filename(fref, physical-filename));
%let fid=%sysfunc(fopen(&fref, u));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fread(&fid));
         /* Read second record. */
      %let rc=%sysfunc(fread(&fid));
         /* Read third record. */
      %let rc=%sysfunc(fread(&fid));
         /* Note position of third record. */
      %let note3=%sysfunc(fnote(&fid));
         /* Read fourth record. */
      %let rc=%sysfunc(fread(&fid));
         /* Read fifth record. */
      %let rc=%sysfunc(fread(&fid));
         /* Point to third record. */
      %let rc=%sysfunc(fpoint(&fid,&note3));
         /* Read third record. */
      %let rc=%sysfunc(fread(&fid));
         /* Copy new text to FDB. */
      %let rc=%sysfunc(fput(&fid, New text));
         /* Update third record  */
         /* with data in FDB. */
      %let rc=%sysfunc(fwrite(&fid));
         /* Close file. */
      %let rc=%sysfunc(fclose(&fid));
   %end;
%let rc=%sysfunc(filename(fref));
%mend test;
%test
```

See Also

Functions:

- “DROPNOTE Function” on page 570
- “FCLOSE Function” on page 591
- “FILENAME Function” on page 611
- “FOPEN Function” on page 720
- “FPOINT Function” on page 728
- “FPUT Function” on page 733
- “FREAD Function” on page 735
- “FREWIND Function” on page 737
- “FWRITE Function” on page 744
- “MOPEN Function” on page 1121

FOPEN Function

Opens an external file and returns a file identifier value.

Category: External Files

Syntax

FOPEN(*fileref* <, *open-mode* <, *record-length* <, *record-format*>>>)

Required Argument

fileref
is a character constant, variable, or expression that specifies the *fileref* assigned to the external file.

Note You cannot create a *fileref* that is greater than eight characters in SAS, although an environment variable that is used as a *fileref* can be greater than eight characters.

Optional Arguments

open-mode
is a character constant, variable, or expression that specifies the type of access to the file:

Here are valid values:

A APPEND mode allows writing new records after the current end of the file.

I INPUT mode allows reading only (default).

O OUTPUT mode defaults to the OPEN mode specified in the operating environment option in the FILENAME statement or function. If no operating environment option is specified, it allows writing new records at the beginning of the file.

S Sequential input mode is used for pipes and other sequential devices such as hardware ports.

U UPDATE mode allows both reading and writing. The file is created if it does not exist.

Default I

record-length
is a numeric constant, variable, or expression that specifies the logical record length of the file. To use the existing record length for the file, specify a length of 0, or do not provide a value here.

record-format
is a character constant, variable, or expression that specifies the record format of the file. To use the existing record format, do not specify a value here. Valid values are:

B data are to be interpreted as binary data.

D use default record format.

E use editable record format.

F file contains fixed length records.

P file contains printer carriage control in operating environment-dependent record format.

V file contains variable length records.

Note: If an argument is invalid, FOPEN returns 0, and you can obtain the text of the corresponding error message from the SYSMSG function. Invalid arguments do not produce a message in the SAS log and do not set the _ERROR_ automatic variable.

Details

General Information

CAUTION
Use OUTPUT mode with care. Opening an existing file for output overwrites the current contents of the file without warning.

The FOPEN function opens an external file for reading or updating and returns a file identifier value that is used to identify the open file to other functions. You must associate a *fileref* with the external file before calling the FOPEN function. FOPEN returns a 0 if the file could not be opened. You can assign *filerefs* by using the FILENAME statement or the FILENAME external file access function. Under some operating environments, you can also assign *filerefs* by using system commands.

If you call the FOPEN function from a macro, the result of the call is valid only when it is passed to functions in a macro. If you call the FOPEN function from the DATA step, the result is valid only when it is passed to functions in the same DATA step.

Operating Environment Information: It is good practice to use the FCLOSE function at the end of a DATA step if you used FOPEN to open the file, even though using FCLOSE might not be required in your operating environment. For more information about FOPEN, see the SAS documentation for your operating environment.

Examples

Example 1: Opening a File Using Defaults

This example assigns the *fileref* MYFILE to an external file and attempts to open the file for input using all defaults. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf, physical-filename));
%let fid=%sysfunc(fopen(&filrf));
```

Example 2: Opening a File without Using Defaults

This example attempts to open a file for input without using defaults. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let fid=%sysfunc(fopen(file2, o, 132, e));
```

See Also

Functions:

- “DOPEN Function” on page 556
- “FCLOSE Function” on page 591
- “FILENAME Function” on page 611
- “FILEREF Function” on page 616
- “MOPEN Function” on page 1121
- “SYSMSG Function” on page 1437

Statements:

- "FILENAME Statement" in *SAS Global Statements for SAS Viya Workbench*

FOPTNAME Function

Returns the name of an item of information about an external file.

Category: External Files

Syntax

FOPTNAME(*file-id*, *nval*)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

nval
is a numeric constant, variable, or expression that specifies the number of the information item.

Details

General Information

FOPTNAME returns a missing or null value if an invalid argument to FOPTNAME is used.

For all available platforms, the number, value, and type of information items that are available vary.

Linux: Available Information Items

The following table shows the information items that correspond to the *nval* values for single and concatenated files under Linux operating environments.

Table 3.4 *Information Items for Files under Linux*

nval	Single File	Pipe Files	Concatenated Files
1	Filename	PIPE command	Filename

2	Owner name	File list
3	Group name	Owner name
4	Access permission	Group name
5	Last Modified	Access permissions
6	File size (bytes)	File size (bytes)

Examples

Example 1: Retrieving File Information Items and Writing Them to the Log

This example retrieves the system-dependent file information items that are available and writes them to the log:

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=
%sysfunc(fopen(&filrf));

 %let infonum=
%sysfunc(foptnum(&fid));

  %do j=1 %to
&infonum;

   %let name=%sysfunc(foptname(&fid,
&j));

   %let value=%sysfunc(finfo(&fid,
&name));

   %put File attribute &name equals
&value;


%end;

 %let rc=
%sysfunc(fclose(&fid));
```

```
    %let rc=
   %sysfunc(filename(filrf));

   %mend
   test;



   %test
```

Example 2: Creating a Data Set with Names and Values of File Attributes

This example creates a data set that contains the name and value of the available file attributes:

```
data fileatt;
   length name $ 20 value $ 40;
   drop rc fid j infonum;
   rc=filename("myfile", "physical-filename");
   fid=fopen("myfile");
   infonum=foptnum(fid);
   do j=1 to infonum;
      name=foptname(fid, j);
      value=finfo(fid, name);
      put 'File attribute ' name
       'has a value of ' value;
      output;
   end;
   rc=filename("myfile");
run;
```

Example 3: File Attributes When Using the Pipe Device Type under Linux

The following example creates a data set that contains the NAME and VALUE attributes returned by the FOPTNAME function when you are using pipes under Linux:

```
data fileatt;
   length name $ 30 value $ 40;
   drop fid j infonum;
   filename mypipe pipe 'LINUX-command';
   fid=fopen("mypipe", "s");
   infonum=foptnum(fid);
   do j=1 to infonum;
      name=foptname(fid, j);
      value=finfo(fid, name);
      put 'File attribute' name 'has a value of ' value;
      output;
   end;
run;
```

SAS writes the following output to the log:

Example Code 3.3 *SAS Log*

```
File attribute Pipe Command has a value of UNIX-command
```

Linux-command is the Linux command or program where you are piping your output or where you are reading your input. This command or program must be either fully qualified or defined in your PATH environment variable.

See Also

Functions:

- “DINFO Function” on page 548
- “DOPTNAME Function” on page 558
- “DOPTNUM Function” on page 560
- “FCLOSE Function” on page 591
- “FILENAME Function” on page 611
- “FINFO Function” on page 699
- “FOPEN Function” on page 720
- “FOPTNUM Function” on page 726
- “MOPEN Function” on page 1121

FOPTNUM Function

Returns the number of information items, such as filename or record length, that are available for an external file.

Category: External Files

Syntax

FOPTNUM(*file-id*)

Required Argument

file-id
is the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

General Information

The number of information items that are available for a file depend on the operating environment and the type of file.

FOPTNUM Specifics under Linux

Under Linux, five information items are available for all types of files:

- Filename
- Owner Name
- Group Name
- Access Permission
- File Size (bytes)

If you concatenate filenames, then an additional information item–File List–is available. If you are using piped files, then the only information item that is available is the PIPE Command.

The *open-mode* specified in the FOPEN function determines the value that FOPTNUM returns.

Table 3.5 *Open Mode and FOPTNUM Values*

Open Mode	FOPTNUM Value	Information Items Available
Append Input Update	6 for concatenated files 5 for single files	All information items available.
Output	5 for concatenated files 4 for single files	Because the file is open for output, the File Size information type is unavailable.
Sequential (using Pipe Device Type)	1	The only information item available is PIPE Command.

For an example of how to use the FOPTNUM function, see "Example 3: File Attributes When Using the Pipe Device Type under Linux" on page 725.

Comparisons

- Use FOPTNAME to determine the names of the items that are available for a particular operating environment.

- Use FINFO to retrieve the value of a particular information item.

Example

This example opens the external file with the fileref MYFILE and determines the number of system-dependent file information items available:

```
%let fid=%sysfunc(fopen(myfile));
%let infonum=%sysfunc(foptnum(&fid));
%let rc=%sysfunc(fclose(&fid));
```

See Also

Functions:

- “DINFO Function” on page 548
- “DOPTNAME Function” on page 558
- “DOPTNUM Function” on page 560
- “FINFO Function” on page 699
- “FOPEN Function” on page 720
- “FOPTNAME Function” on page 723
- “MOPEN Function” on page 1121

FPOINT Function

Positions the read pointer on the next record to be read.

Category: External Files

Syntax

FPOINT(*file-id*, *note-id*)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

note-id
specifies the identifier that was assigned by the FNOTE function.

Details

FPOINT returns 0 if the operation was successful, or ≠0 if it was not successful. FPOINT determines only the record to read next. It has no impact on which record is written next. When you open the file for update, FWRITE writes to the most recently read record.

Note: You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it reads the records and uses NOTE3 to store the position of the third record read. Later, it points back to NOTE3 to update the file, and closes the file afterward:

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=%sysfunc(fopen(&filrf,
u));

  %if &fid > 0 %then
%do;

   /* Read first record.
*/

   %let rc=
%sysfunc(fread(&fid));

   /* Read second record.
*/

   %let rc=
%sysfunc(fread(&fid));

   /* Read third record.
*/

   %let rc=
%sysfunc(fread(&fid));
```

```
   /* Note position of third record.
*/

   %let note3=
%sysfunc(fnote(&fid));

   /* Read fourth record.
*/

   %let rc=
%sysfunc(fread(&fid));

   /* Read fifth record.
*/

   %let rc=
%sysfunc(fread(&fid));

   /* Point to third record.
*/

   %let rc=%sysfunc(fpoint(&fid,
&note3));

   /* Read third record.
*/

   %let rc=
%sysfunc(fread(&fid));

   /* Copy new text to FDB.
*/

   %let rc=%sysfunc(fput(&fid, New
text));

   /* Update third record
*/

   /* with data in FDB.
*/

   %let rc=
%sysfunc(fwrite(&fid));

   /* Close file.
*/

   %let rc=
%sysfunc(fclose(&fid));


%end;
```

```
 %let rc=
%sysfunc(filename(filrf));

%mend
test;



%test
```

See Also

Functions:

- “DROPNOTE Function” on page 570
- “FCLOSE Function” on page 591
- “FILENAME Function” on page 611
- “FNOTE Function” on page 718
- “FOPEN Function” on page 720
- “FPUT Function” on page 733
- “FREAD Function” on page 735
- “FREWIND Function” on page 737
- “FWRITE Function” on page 744
- “MOPEN Function” on page 1121

FPOS Function

Sets the position of the column pointer in the File Data Buffer (FDB).

Category: External Files

Syntax

FPOS(*file-id*, *nval*)

Required Arguments

file-id

is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

nval
is a numeric constant, variable, or expression that specifies the column at which to set the pointer.

Details

FPOS returns 0 if the operation was successful, ≠0 if it was not successful. If you open a file in output mode and the specified position is past the end of the current record, the size of the record is increased appropriately. However, in a fixed block or VBA file, if you specify a column position beyond the end of the record, the record size does not change and the text string is not written to the file.

If you open a file in Update mode and the specified position is not past the end of the current record, then SAS writes the record to the file. If the specified position is past the end of the current record, then SAS returns an error message and does not write the new record:

```
ERROR: Cannot increase record length in update mode.
```

Note: If you use the Update mode with the FOPEN function, then you must execute FREAD before you execute FWRITE functions.

Example

This example assigns the fileref MYFILE to an external file and opens the file in Update mode. If the file is opened successfully, indicated by a positive value in the variable FID, SAS reads a record and places data into the file's buffer at column 12. If the resulting record length is less than or equal to the original record length, then SAS writes the record and closes the file. If the resulting record length is greater than the original record length, then SAS writes an error message to the log.

```
%macro ptest;
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf, test.txt));
%let fid=%sysfunc(fopen(&filrf, o));
%let rc=%sysfunc(fread(&fid));
%put &fid;
%if (&fid > 0) %then
   %do;
      %let dataline=This is some data.;
         /* Position at column 12 in the FDB. */
      %let rc=%sysfunc(fpos(&fid, 12));
      %put &rc one;
         /* Put the data in the FDB. */
      %let rc=%sysfunc(fput(&fid, &dataline));
      %put &rc two;
      %if (&rc ne 0) %then
         %do;
            %put %sysfunc(sysmsg());
         %end;
```

```
      %else %do;
         /* Write the record. */
      %let rc=%sysfunc(fwrite(&fid));
      %if (&rc ne 0) %then
         %do;
            %put write fails &rc;
         %end;
      %end;
         /* Close the file. */
      %let rc=%sysfunc(fclose(&fid));
   %end;
%let rc=%sysfunc(filename(filrf));
%mend;
%ptest;
```

SAS writes the following output to the log:

```
1
0 one
0 two
```

See Also

Functions:

- “FCLOSE Function” on page 591
- “FCOL Function” on page 592
- “FILENAME Function” on page 611
- “FOPEN Function” on page 720
- “FPUT Function” on page 733
- “FWRITE Function” on page 744
- “MOPEN Function” on page 1121

FPUT Function

Moves data to the File Data Buffer (FDB) of an external file, starting at the FDB's current column position.

Category: External Files

Syntax

FPUT(*file-id*, *cval*)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

cval
is a character constant, variable, or expression that specifies the file data.

Details

FPUT returns 0 if the operation was successful, ≠0 if it was not successful. The number of bytes moved to the FDB is determined by the length of the variable. The value of the column pointer is then increased to one position past the end of the new text.

Note: You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file in APPEND mode. If the file is opened successfully, indicated by a positive value in the variable FID, it moves data to the FDB using FPUT, appends a record using FWRITE, and then closes the file. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro ptest;
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf, test.txt));
%let fid=%sysfunc(fopen(&filrf, a));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fread(&fid));
      %let mystring=This is some data.;
      %let rc=%sysfunc(fput(&fid, &mystring));
      %let rc=%sysfunc(fwrite(&fid));
      %let rc=%sysfunc(fclose(&fid));
   %end;
%else
   %put %sysfunc(sysmsg());
%let rc=%sysfunc(filename(filrf));
%put return code=&rc;
%mend;
%ptest;
```

SAS writes the following output to the log:

```
return code = 0
```

See Also

Functions:

- "FCLOSE Function" on page 591
- "FILENAME Function" on page 611
- "FNOTE Function" on page 718
- "FOPEN Function" on page 720
- "FPOINT Function" on page 728
- "FPOS Function" on page 731
- "FWRITE Function" on page 744
- "MOPEN Function" on page 1121
- "SYSMSG Function" on page 1437

FREAD Function

Reads a record from an external file into the File Data Buffer (FDB).

Category: External Files

Syntax

FREAD(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FREAD returns 0 if the operation was successful, ≠0 if it was not successful. The position of the file pointer is updated automatically after the Read operation so that successive FREAD functions read successive file records.

To position the file pointer explicitly, use FNOTE, FPOINT, and FREWIND.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file opens successfully, it lists all of the file's records in the log:

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=
%sysfunc(fopen(&filrf));

 %if &fid > 0
%then

  %do
%while(%sysfunc(fread(&fid))=0);

   %let rc=%sysfunc(fget(&fid, c,
200));

   %put
&c;


%end;

 %let rc=
%sysfunc(fclose(&fid));

 %let rc=
%sysfunc(filename(filrf));

%mend
test;



%test
```

See Also

Functions:

- "FCLOSE Function" on page 591
- "FGET Function" on page 607
- "FILENAME Function" on page 611

- “FNOTE Function” on page 718
- “FOPEN Function” on page 720
- “FPOINT Function” on page 728
- “FREWIND Function” on page 737
- “MOPEN Function” on page 1121

FREWIND Function

Positions the file pointer to the start of the file.

Category: External Files

Syntax

FREWIND(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FREWIND returns 0 if the operation was successful, ≠0 if it was not successful. FREWIND has no effect on a file opened with sequential access.

Example

This example assigns the fileref MYFILE to an external file. Then it opens the file and reads the records until the end of the file is reached. The FREWIND function then repositions the pointer to the beginning of the file. The first record is read again and stored in the File Data Buffer (FDB). The first token is retrieved and stored in the macro variable VAL:

```
%macro
test;

 %let
filrf=myfile;
```

```
 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=
%sysfunc(fopen(&filrf));

 %let
rc=0;

 %do %while (&rc ne
-1);

  /* Read a record.
*/

  %let rc=
%sysfunc(fread(&fid));


%end;

 /* Reposition pointer to beginning of file.
*/

 %if &rc = -1 %then
%do;

  %let rc=
%sysfunc(frewind(&fid));

  /* Read first record.
*/

  %let rc=
%sysfunc(fread(&fid));

  /* Read first token
*/

  /* into macro variable VAL.
*/

  %let rc=%sysfunc(fget(&fid,
val));

  %put
val=&val;


%end;


%else
```

```
  %put Error on fread=
%sysfunc(sysmsg());

 %let rc=
%sysfunc(fclose(&fid));

 %let rc=
%sysfunc(filename(filrf));

%mend
test;



%test
```

See Also

Functions:

- “FCLOSE Function” on page 591
- “FGET Function” on page 607
- “FILENAME Function” on page 611
- “FOPEN Function” on page 720
- “FREAD Function” on page 735
- “MOPEN Function” on page 1121
- “SYSMSG Function” on page 1437

FRLEN Function

Returns the size of the last record that was read, or, if the file is opened for output, returns the current record size.

Category: External Files

Syntax

FRLEN(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Example

This example opens the file that is identified by the fileref MYFILE. It determines the minimum and maximum length of records in the external file and writes the results to the log:

```
%macro
test;

 %let fid=
%sysfunc(fopen(myfile));

 %let
min=0;

 %let
max=0;

  %if (%sysfunc(fread(&fid))=0) %then
%do;

   %let min=
%sysfunc(frlen(&fid));

   %let
max=&min;

    %do
%while(%sysfunc(fread(&fid))=0);

     %let reclen=
%sysfunc(frlen(&fid));

      %if (&reclen > &max) %then %let
max=&reclen;

      %if (&reclen < &min) %then %let
min=&reclen;


%end;


%end;

 %let rc=
%sysfunc(fclose(&fid));
```

```
 %put max=&max
min=&min;

%mend
test;



%test
```

See Also

Functions:

- “FCLOSE Function” on page 591
- “FOPEN Function” on page 720
- “FREAD Function” on page 735
- “MOPEN Function” on page 1121

FSEP Function

Sets the token delimiters for the FGET function.

Category: External Files

Syntax

FSEP(*file-id*, *character(s)* <, 'x' | 'X'>)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

character
is a character constant, variable, or expression that specifies one or more delimiters that separate items in the File Data Buffer (FDB). Each character listed is a delimiter. That is, if *character* is #@, either # or @ can separate items. Multiple consecutive delimiters, such as @#@, are treated as a single delimiter.

Default blank

Optional Argument

'x' | 'X'
specifies that the character delimiter is a hexadecimal value.

Restrictions 'x' and 'X' are the only valid values for this argument. All other values will cause an error to occur.

If you pass 'x' or 'X' as the third argument, a valid hexadecimal string must be passed as the second argument, *character*. Otherwise, the function will fail. A valid hexadecimal string is an even number of 0–9 and A–F characters.

Tip If you use a macro statement, then quotation marks enclosing x or X are not required.

Details

FSEP returns 0 if the operation was successful, ≠0 if it was not successful.

Example

An external file has data in this form:

```
John J. Doe,Male,25,Weight Lifter
Pat O'Neal,Female,22,Gymnast
```

Note that each field is separated by a comma.

This example reads the file that is identified by the fileref MYFILE, using the comma as a separator, and writes the values for NAME, GENDER, AGE, and WORK to the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks, but a literal comma in a function argument must be enclosed in a macro quoting function such as %STR.

```
%macro
test;

 %let fid=
%sysfunc(fopen(myfile));

 %let rc=
%sysfunc(fsep(&fid,%str(,)));

  %do
%while(%sysfunc(fread(&fid))=0);

   %let rc=%sysfunc(fget(&fid,
name));
```

```
   %let rc=%sysfunc(fget(&fid,
gender));

   %let rc=%sysfunc(fget(&fid,
age));

   %let rc=%sysfunc(fget(&fid,
work));

   %put name=%bquote(&name) gender=&gender age=&age
work=&work;


%end;

%let rc=
%sysfunc(fclose(&fid));

%mend
test;



%test
```

See Also

Functions:

- “FCLOSE Function” on page 591
- “FGET Function” on page 607
- “FOPEN Function” on page 720
- “FREAD Function” on page 735
- “MOPEN Function” on page 1121

FUZZ Function

Returns the nearest integer if the argument is within 1E−12 of that integer.

Category: Rounding and Truncation

Syntax

FUZZ(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The FUZZ function returns the nearest integer value if the argument is within 1E−12 of the integer (that is, if the absolute difference between the integer and argument is less than 1E−12). Otherwise, the argument is returned.

Example

```
data
one;

var1=5.9999999999999;


x=fuzz(var1);

  put x=
16.14;


y=fuzz(5.99999999);

   put y=
16.14;

run;
```

These statements produce these results:

```
x=6.00000000000000
y=5.99999999000000
```

FWRITE Function

Writes a record to an external file.

Category: External Files

Syntax

FWRITE(*file-id* <, *cc*>)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Optional Argument

cc
is a character constant, variable, or expression that specifies a carriage-control character

Here are valid values:

blank	starts the record on a new line.
0	skips one blank line before a new line.
-	skips two blank lines before a new line.
1	starts the line on a new page.
+	overstrikes the line on a previous line.
P	interprets the line as a computer prompt.
=	interprets the line as carriage control information.
all else	starts the line record on a new line.

Details

FWRITE returns 0 if the operation was successful, ≠0 if it was not successful. FWRITE moves text from the File Data Buffer (FDB) to the external file. In order to use the carriage-control characters, you must open the file with a record format of `P` (print format) in FOPEN.

Note: When you use the Update mode, you must execute FREAD before you execute FWRITE. You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it writes the numbers 1 to 50 to the

external file, skipping two blank lines. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro
test;

 %let
filrf=myfile;

 %let rc=%sysfunc(filename(filrf, physical-
filename));

 %let fid=%sysfunc(fopen(&filrf, o, 0,
P));



  %do i=1 %to
50;

   %let rc=%sysfunc(fput(&fid,
%sysfunc(putn(&i,2.))));

   %if (%sysfunc(fwrite(&fid,-)) ne 0) %then %put
%sysfunc(sysmsg());


%end;

 %let rc=
%sysfunc(fclose(&fid));

%mend
test;



%test
```

See Also

Functions:

- “FAPPEND Function” on page 588
- “FCLOSE Function” on page 591
- “FGET Function” on page 607
- “FILENAME Function” on page 611
- “FOPEN Function” on page 720
- “FPUT Function” on page 733
- “SYSMSG Function” on page 1437

GAMINV Function

Returns a quantile from the gamma distribution.

Category: Quantile

Syntax

GAMINV(*p*, *a*)

Required Arguments

p
is a numeric probability.

Range $0 \leq p < 1$

a
is a numeric shape parameter.

Range $a > 0$

Details

The GAMINV function returns the pth quantile from the gamma distribution, with shape parameter *a*. The probability that an observation 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

```
data
one;

   q1=gaminv(0.5,
9);

   q2=gaminv(0.1,
2.1);
```

```
   put q1=
q2=;

run;
```

These statements produce these results:

```
q1=8.6689511844 q2=0.5841932369
```

See Also

Functions:

- "QUANTILE Function" on page 1293

GAMMA Function

Returns the value of the gamma function.

Category: Mathematical

Syntax

GAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Restriction Nonpositive integers are invalid.

Details

The GAMMA function returns the integral given by

$$\mathrm{GAMMA}(x) = \int_0^{\infty} t^{x-1} e^{-t}\, dt.$$

For positive integers, GAMMA(x) is (x − 1)!. This function is commonly denoted by $\Gamma(x)$.

Example

```
data
one;


x=gamma(6);

   put
x=;

run;
```

These statements produce this result:

```
x=120
```

GARKHCLPRC Function

Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.

Category: Financial

Syntax

GARKHCLPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Required Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t

is a nonmissing value that specifies the time to maturity, in years.

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

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*.

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*.

sigma
is a nonmissing, positive fraction that specifies the volatility of the currency rate.

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, 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:

$$\text{CALL} = \max((S - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" on page 11.

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

```
data
one;

   a=garkhclprc(40, .5,
38, .06, .04, .2);



   b=garkhclprc(19, .25,
20, .05, .03, .09);

   put a=;
   put b=;
run;
```

These statements produce these results:

```
a=1.449425106
b=1.1304209448
```

See Also

Functions:

- “GARKHPTPRC Function” on page 751

GARKHPTPRC Function

Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

Category: Financial

Syntax

GARKHPTPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Required Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t

is a nonmissing value that specifies the time to maturity, in years.

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

***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*.

***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*.

sigma

is a nonmissing, positive fraction that specifies the volatility of the currency rate.

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" on page 11.

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

```
data
one;

   a=garkhptprc(50, .7,
55, .05, .04, .2);

   b=garkhptprc(32, .3,
33, .05, .03, .3);

   put a=
b=;

run;
```

These statements produce these results:

```
a=1.4050880945 b=1.5647320514
```

See Also

Functions:

- "GARKHCLPRC Function" on page 749

GCD Function

Returns the greatest common divisor for one or more integers.

Category: Mathematical

Syntax

GCD(*x1, x2, x3, ..., xn*)

Required Argument

x
specifies a numeric constant, variable, or expression that has an integer value.

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.

If any of the arguments are missing, then the returned value is a missing value.

Example

The following example returns the greatest common divisor of the integers 10 and 15.

```
data _null_;
   x=gcd(10, 15);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=5
```

See Also

Functions:

- "LCM Function" on page 1046

GEODIST Function

Returns the geodetic distance between two latitude and longitude coordinates.

Category: Distance

Syntax

GEODIST(*latitude-1*, *longitude-1*, *latitude-2*, *longitude-2* <, *'options'*>)

Required Arguments

latitude-1* | *latitude-2
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.

longitude-1* | *longitude-2
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, the value must be between 540 and –540. If the value is expressed in radians, it must be between 3*pi and –3*pi .

Optional Argument

'option'
specifies a character constant, variable, or expression.

Here are valid values:

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.

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 example 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_;
   distance=geodist(30.68, -88.25, 35.43, -82.55);
   put 'Distance= ' distance 'kilometers';
run;
```

SAS writes the following output to the log:

```
Distance= 748.6529147 kilometers
```

Example 2: Calculating the Geodetic Distance in Miles

The following example 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_;
   distance=geodist(30.68, -88.25, 35.43, -82.55, 'M');
   put 'Distance = ' distance 'miles';
run;
```

SAS writes the following output to the log:

```
Distance = 465.29081088 miles
```

Example 3: Calculating the Geodetic Distance with Input Measured in Degrees

The following example 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_;
   input lat1 long1 lat2 long2;
   Distance = geodist(lat1, long1, lat2, long2, 'DM');
   put 'Distance = ' Distance 'miles';
   datalines;
35.2 -78.1 37.6 -79.8
;
run;
```

SAS writes the following output to the log:

```
Distance = 190.72474282 miles
```

Example 4: Calculating the Geodetic Distance with Input Measured in Radians

The following example 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_;
   input lat1 long1 lat2 long2;
   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';
   datalines;
35.2 -78.1 37.6 -79.8
;
run;
```

SAS writes the following output to the log:

```
Distance= 190.68397508 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.

Category: Descriptive Statistics

Syntax

GEOMEAN(*argument* <, *argument*, ..., *argument-n*>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If any argument is zero, then the geometric mean is zero. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the geometric mean of the nonmissing values.

Let n be the number of arguments with 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

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

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 nonmissing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Example

```
data
one;

   x1=geomean(1, 2, 2,
4);

   x2=geomean(., 2, 4,
8);

   x3=geomean(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=2 x2=4 x3=2.8284271247
```

See Also

Functions:

- “GEOMEANZ Function” on page 759
- “HARMEAN Function” on page 889
- “HARMEANZ Function” on page 891
- “MEAN Function” on page 1103

GEOMEANZ Function

Returns the geometric mean, using zero fuzzing.

Category: Descriptive Statistics

Syntax

GEOMEANZ(*argument* <, *argument*, ..., *argument-n*>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If any argument is zero, then the geometric mean is zero. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the geometric mean of the nonmissing values.

Let n be the number of arguments with 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

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

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 nonmissing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

```
data
one;

   x1=geomeanz(1, 2, 2,
4);
```

```
   x2=geomeanz(., 2, 4,
8);

   x3=geomeanz(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=2 x2=4 x3=2.8284271247
```

See Also

Functions:

- “GEOMEAN Function” on page 758
- “HARMEAN Function” on page 889
- “HARMEANZ Function” on page 891
- “MEAN Function” on page 1103

GETOPTION Function

Returns the value of a SAS system or graphics option.

Category: Special

Syntax

GETOPTION(*option-name* <, *return-value-option*> <*return-value-formatting-options*>)

Required Argument

option-name
is a character constant, variable, or expression that specifies the name of the system option.

Tips Do not put an equal sign after the name. For example, write PAGESIZE= as PAGESIZE.

SAS options that are passwords, such as EMAILPW, return the value xxxxxxxx, and not the actual password.

Return Value Options

DEFAULTVALUE
returns the default option value.

Restriction DEFAULTVALUE is valid only for SAS system options. SAS issues a warning message when the DEFAULTVALUE option is specified and *option-name* is a graphics option.

HOWSCOPE
returns a character string that specifies the scope of an option.

Restriction HOWSCOPE is valid only for SAS system options. SAS issues a warning message when the HOWSCOPE option is specified and *option-name* is a graphics option.

HOWSET
returns a character string that specifies how an option value was set.

Restriction HOWSET is valid only for SAS system options. SAS issues a warning message when the HOWSET option is specified and *option-name* is a graphics option.

STARTUPVALUE
returns the system option value that was used to start SAS either on the command line or in a configuration file.

Restriction STARTUPVALUE is valid only for SAS system options. SAS issues a warning message when the STARTUPVALUE option is specified and *option-name* is a graphics option.

Return Value Formatting Options

CM
reports graphic units of measure in centimeters.

Restriction CM is valid only for graphics options and the following SAS system options: BOTTOMMARGIN, TOPMARGIN, RIGHTMARGIN, and LEFTMARGIN. SAS writes a note to the log when the CM option is specified and *option-name* is not a graphics option or an option that specifies a margin value.

EXPAND
for options that contain environment variables, returns the option value with the value of the environment variable.

Restrictions Variable expansion is valid only in the Windows and UNIX operating environments.

EXPAND is valid only for character system option values. EXPAND is ignored if *option-name* has an option type of Boolean, such as CENTER or NOCENTER, or if the value of the option is numeric.

Note SAS issues a note when EXPAND is specified for Boolean options and for options that have numeric values. SAS issues a warning when EXPAND is specified and the option is a graphics option.

Tip By default, some option values are displayed with expanded variable values. Other options require the EXPAND option in the PROC OPTIONS statement. Use the DEFINE option in the PROC OPTIONS statement to determine whether an option value expands variables by default or if the EXPAND option is required. If the output from PROC OPTIONS DEFINE shows the following information, you must use the EXPAND option to expand variable values:

```
Expansion: Environment variables, within the option value,
           are not expanded
```

KEYEXPAND

for options that contain environment variables, returns the value in the format **`option-name=value`**.

Restriction KEYEXPAND is valid only for character system option values. SAS issues an error message when the KEYEXPAND option is specified and *option-name* is a graphics option. KEYEXPAND is ignored if *option-name* has an option type of Boolean, such as CENTER or NOCENTER, or if the value of the option is numeric.

KEYWORD

returns option values in a **`option-name=value`** format that would be suitable for direct use in the SAS OPTIONS or GOPTIONS global statements.

Restrictions KEYWORD is not valid when it is used with the HEXVALUE, EXPAND, KEYEXPAND, or LOGNUMBERFORMAT options. SAS writes a note to the log when the GETOPTION function contains conflicting options.

KEYWORD is valid only for character or numeric system option values. KEYWORD is ignored for system options whose option type is Boolean, such as CENTER or NOCENTER. SAS issues an error message when the KEYWORD option is specified and *option-name* is a graphics option.

Note For a system option with a null value, the GETOPTION function returns a value of ' ' (single quotation marks with a blank space between them). An example is EMAILID=' '.

HEXVALUE

returns the option value as a hexadecimal value.

Restriction HEXVALUE is valid only for character or numeric system option values. If HEXVALUE is specified for system options whose option

type is Boolean, such as CENTER or NOCENTER, or if *option-name* is a graphics option, SAS issues an error message.

IN
reports graphic units of measure in inches.

Restriction IN is valid only for graphics options and the following SAS system options: BOTTOMMARGIN, TOPMARGIN, RIGHTMARGIN, and LEFTMARGIN. SAS writes a note to the log when the IN option is specified and *option-name* is not a graphics option or an option that specifies a margin value.

LOGNUMBERFORMAT
formats SAS system option values using locale-specific punctuation.

Restriction Do not use LOGNUMBERFORMAT if the returned value is used to set an option value by using the OPTIONS statement. The OPTIONS statement does not accept commas in numeric values.

Examples

Example 1: Using GETOPTION to Save and Restore the YEARCUTOFF Option

This example saves the value of the YEARCUTOFF option, processes SAS statements based on the value of the YEARCUTOFF option, and then resets the value to 1940 if it is not already 1940.

```
/* Save the value of the YEARCUTOFF system option */
%let cutoff=%sysfunc(getoption(yearcutoff,keyword));

data ages;
  if getoption('yearcutoff') = '1940' then
     do;
        ...more SAS statements...
     end;
     else do;
             ...more SAS statements...
             /* Reset YEARCUTOFF   */
             options &cutoff;
          end;
run;
```

Example 2: Using GETOPTION to Obtain Different Reporting Options

This example defines a macro to illustrate the use of the GETOPTION function to obtain the value of system and graphics options by using different reporting options.

```
%macro showopts;
%put MAPS= %sysfunc(getoption(MAPS));
```

```
%put MAPSEXPANDED=%sysfunc(getoption(MAPS, EXPAND));
%put MAPSKEYEXPANDED=%sysfunc(getoption(MAPS, KEYEXPAND));
%put PAGESIZE= %sysfunc(getoption(PAGESIZE));
%put PAGESIZESETBY= %sysfunc(getoption(PAGESIZE, HOWSET));
%put PAGESIZESCOPE= %sysfunc(getoption(PAGESIZE, HOWSCOPE));
%put PS= %sysfunc(getoption(PS));
%put LS= %sysfunc(getoption(LS));
%put PS(keyword form)= %sysfunc(getoption(PS,keyword));
%put LS(keyword form)= %sysfunc(getoption(LS,keyword));
%put HSIZE= %sysfunc(getoption(HSIZE));
%put VSIZE= %sysfunc(getoption(VSIZE));
%put HSIZE(in/keyword form)= %sysfunc(getoption(HSIZE,in,keyword));
%put HSIZE(cm/keyword form)= %sysfunc(getoption(HSIZE,cm,keyword));
%put VSIZE(in/keyword form)= %sysfunc(getoption(VSIZE,in,keyword));
%put HSIZE(cm/keyword form)= %sysfunc(getoption(VSIZE,cm,keyword));
%mend;

goptions VSIZE=8.5 in HSIZE=11 in;
options PAGESIZE=67;
%showopts
```

The following is the SAS log:

```
MAPS= maps-path/maps
MAPSEXPANDED=maps-path/maps
MAPSKEYEXPANDED=MAPS=maps-path/maps
PAGESIZE= 67
PAGESIZESETBY= Options Statement
PAGESIZESCOPE= Compute Server
PS= 67
LS= 132
PS(keyword form)= PS=67
LS(keyword form)= LS=132
HSIZE= 11.0000 in
VSIZE= 8.5000 in
HSIZE(in/keyword form)= HSIZE=11.0000 in
HSIZE(cm/keyword form)= HSIZE=27.9400 cm
VSIZE(in/keyword form)= VSIZE=8.5000 in
HSIZE(cm/keyword form)= VSIZE=21.5900 cm
```

GETVARC Function

Returns the value of a SAS data set character variable.

Category: SAS File I/O

Syntax

GETVARC(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
is a numeric constant, variable, or expression that specifies the data set identifier that the OPEN function returns.

variable-number
is a numeric constant, variable, or expression that specifies the number of the variable in the Data Set Data Vector (DDV).

Tips You can obtain this value by using the VARNUM function.

This value is listed next to the variable when you use the CONTENTS procedure.

Details

Use VARNUM to obtain the number of a variable in a SAS data set. VARNUM can be nested or it can be assigned to a variable that can then be passed as the second argument, as shown in the following examples. GETVARC reads the value of a character variable from the current observation in the Data Set Data Vector (DDV) into a macro or DATA step variable.

Examples

Example 1: Obtaining the 10th Observation

This example opens the Sasuser.Houses data set and gets the entire 10th observation. The data set identifier value for the open data set is stored in the macro variable MYDATAID. This example nests VARNUM to return the position of the variable in the DDV, and reads in the value of the character variable STYLE.

```
%macro
test;

   %let mydataid=%sysfunc(open(sasuser.houses,
i));

   %let rc=%sysfunc(fetchobs(&mydataid,
10));

   %let style=%sysfunc(getvarc(&mydataid, %sysfunc(varnum (&mydataid,
STYLE))));
   %put
&style;

   %let rc=
%sysfunc(close(&mydataid));
```

```
%mend
test;



%test
```

Example 2: Fetching Data From Observation 10

This example assigns VARNUM to a variable that can then be passed as the second argument. This example fetches data from observation 10.

```
%macro
test;

   %let mydataid=%sysfunc(open(sashelp.class,
i));

   %let namenum=%sysfunc(varnum(&mydataid,
NAME));

   %let rc=%sysfunc(fetchobs(&mydataid,
10));

   %let user=%sysfunc(getvarc(&mydataid,
&namenum));

   %put
&user;

   %let rc=
%sysfunc(close(&mydataid));

%mend
test;



%test
```

See Also

Functions:

- “FETCH Function” on page 600
- “FETCHOBS Function” on page 602
- “GETVARN Function” on page 768
- “VARNUM Function” on page 1504

GETVARN Function

Returns the value of a SAS data set numeric variable.

Category: SAS File I/O

Note: The GERVARN function reads the value of a double variable.

Syntax

GETVARN(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
is a numeric constant, variable, or expression that specifies the data set identifier that the OPEN function returns.

variable-number
is a numeric constant, variable, or expression that specifies the number of the variable in the Data Set Data Vector (DDV).

Tips You can obtain this value by using the VARNUM function.

This value is listed next to the variable when you use the CONTENTS procedure.

Details

Use VARNUM to obtain the number of a variable in a SAS data set. You can nest VARNUM or you can assign it to a variable that can then be passed as the second argument, as shown in the "Examples" section. GETVARN reads the value of a numeric variable from the current observation in the Data Set Data Vector (DDV) into a macro variable or DATA step variable.

Examples

Example 1: Obtaining the 10th Observation

This example obtains the entire 10th observation from a SAS data set. The data set must have been previously opened using OPEN. The data set identifier value for the open data set is stored in the variable MYDATAID. This example nests

VARNUM, and reads in the value of the numeric variable PRICE from the 10th observation of an open SAS data set.

```
%macro
test;

   %let mydataid=
%sysfunc(open(sashelp.class));

   %let rc=%sysfunc(fetchobs(&mydataid,
10));

   %let age=%sysfunc(getvarn(&mydataid, %sysfunc(varnum (&mydataid,
age))));
   %put
&age;

   %let rc=
%sysfunc(close(&mydataid));

%mend
test;



%test
```

Example 2: Fetching data from observation 10

This example assigns VARNUM to a variable that can then be passed as the second argument. This example fetches data from observation 10.

```
%macro
test;

   %let mydataid=
%sysfunc(open(sashelp.class));

   %let num=%sysfunc(varnum (&mydataid,
age));

   %let rc=%sysfunc(fetchobs(&mydataid,
10));

   %let age=%sysfunc(getvarn (&mydataid,
#));

   %put
&age;

   %let rc=
%sysfunc(close(&mydataid));

%mend
test;
```

```
%test
```

See Also

Functions:

- “FETCH Function” on page 600
- “FETCHOBS Function” on page 602
- “GETVARC Function” on page 765
- “VARNUM Function” on page 1504

GIT_BRANCH_CHKOUT Function

Check out a branch in a Git repository.

Category: Git

See: For more information, see “Using Git Functions in SAS” on page 67.
If you previously used the GITFN_ functions, see “Deprecated Git Functions”.

Syntax

GIT_BRANCH_CHKOUT(*'directory'*, *'branch'*)

Required Arguments

'directory'
specifies the path to a local Git repository on the SAS server.

'branch'
specifies the name of a branch in the local Git repository.

Details

Overview

The GIT_BRANCH_CHKOUT function checks out a specified branch in the local repository. Checking out a branch changes the contents of the local repository on the SAS server to match the specified branch’s contents.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.6 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example

```
data _null_;
 rc= git_branch_chkout(
  "C:\User\Local_Git_Repo",        /*1*/
  "your-target-branch");          /*2*/
  put rc=;
run;
```

1 Specify your local repository.

2 Specify your target Git branch to check out.

The preceding statements produce these results:

```
NOTE: Branch "your-target-branch" successfully checked out.
rc=0
```

GIT_BRANCH_DELETE Function

Deletes a Git branch in the repository.

Category: Git

See: For more information, see"Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_BRANCH_DELETE('*directory*', '*branch-name*'<, *remote*>)

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'branch-name'
specifies the name of the branch to be deleted.

Optional Argument

remote
indicates that the branch is a remote branch. The only valid value for *remote* is 1.

Details

Overview

The GIT_BRANCH_DELETE function deletes a branch in the local Git repository. If your target branch is a remote branch, specify 1 for the optional argument *remote*. If the target branch is local, do not supply a value for the optional argument *remote*.

Return Codes

The GIT_BRANCH_DELETE function has return codes to indicate whether the function was successful. If an error occurs, check the log for additional return codes from Git.

Table 3.7 *Return Codes*

Return Code	Description
–1	Indicates that the delete branch function failed because it is the current branch. See the log for additional information from Git.
0	Indicates that the branch was successfully deleted. A note in the log is displayed stating "Branch *branch-name* was successfully deleted."

Example: Delete a Branch in the Local Repository.

```
data _null_;
 rc= git_branch_delete(
  "C:\User\Local_Git_Repo\",       /* 1 */
  "your-branch-name");           /* 2 */
 put rc=;
run;
```

1 Specify your local Git repository.

2 Specify the name of the Git branch that you want to delete.

The preceding statements produce these results:

```
NOTE: Branch "your-branch-name" successfully deleted.
 rc=0
```

GIT_BRANCH_MERGE Function

Merges a Git branch into the currently checked-out branch.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_BRANCH_MERGE(*'directory'*, *'branch-name'*, *'author-name'*, *'author-email'* <, remote>)

Required Arguments

'directory'
specifies the pathname of a local repository on the SAS server.

'branch-name'
specifies the branch that is merged into the currently checked-out branch.

'author-name'
specifies the author's name for the branch merge.

'author-email'
specifies the author's email for the branch merge.

Optional Argument

remote
indicates if the target branch is a remote branch or local branch.

1
indicates that the target branch is a remote branch.

0 or NULL
indicates that the target branch is a local branch.

Details

Overview

The GIT_BRANCH_MERGE function merges a specified Git branch into the currently checked-out Git branch. If a fast-forward merge is acceptable, the GIT_BRANCH_MERGE function automatically performs the fast-forward merge. If a fast-forward merge is not acceptable, the GIT_BRANCH_MERGE function creates a merge commit. The GIT_BRANCH_MERGE function uses the local branch by default. If your target branch is a remote branch, specify the value 1 for the optional argument *remote*. You can explicitly indicate the target branch is a local branch by specifying the value 0 for optional argument*remote*.

Return Codes

The GIT_BRANCH_MERGE function has return codes that indicate whether the function was successful.

Table 3.8 *Return Codes*

Return Code	Description
–2	Indicates that the index has conflicts. Check the log for conflicting files.
–1	Indicates that the merge failed. Check the log for additional details.
0	Indicates that the merge was successful.
1	Indicates that the branch was already up-to-date.

Example

```
data _null_;
 rc = git_branch_merge(
  "C:\User\Local_Git_Repo",          /*1*/
  "your-target-branch",             /*2*/
  "your-author-name",               /*3*/
  "your-author-email");             /*4*/
 put rc=;
run;
```

1 Specify the pathname of your local Git repository.

2 Specify the name of the branch that is merged into your currently checked-out branch. To check out a different branch, use the GIT_BRANCH_CHKOUT function.

3 Specify the name of the author for the merge that appears in the commit history.

4 Specify the email of the author for the merge that appears in the commit history.

The preceding statements produce these results:

```
NOTE: Merge 'your-target-branch' into current branch 'your-current-branch'
successful.
 rc=0
```

GIT_BRANCH_NEW Function

Creates a Git branch.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_BRANCH_NEW(*'directory'*, *'commit-ID'*, *'branch-name'*, *force*)

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'commit-ID'
specifies which commit level the new branch is cloned from.

'branch-name'
specifies a name for the new branch.

force
specifies whether to overwrite existing branches with the same name or not to overwrite existing branches.

Details

Overview

The GIT_BRANCH_NEW function creates a branch in the local repository at a specified commit level. Branch names can contain only letters and numbers. The force argument allows you to specify whether you want to overwrite an existing branch with the same name. Valid values for force are listed below:

0
do not overwrite an existing branch.

1
overwrite an existing branch.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.9 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example

```
data _null_;
 rc = git_branch_new(
  "your-directory",           /* 1 */
  "your-commit-ID",           /* 2 */
  "your-branch-name",         /* 3 */
  your-force-value);          /* 4 */
 put rc=;
run;
```

1 Specify your local Git repository.

2 Specify the commit ID to build the new branch from.

3 Specify a name for the new branch.

4 Specify whether to overwrite an existing branch with the same name.

The preceding statements produce this result:

```
NOTE: Branch "your-branch-name" successfully created.
rc=0
```

GIT_CLONE Function

Clones a Git repository into a directory on the SAS server.

Category: Git

Note: Starting with 2025.09, the GIT_CLONE function supports custom headers for remote functions.

See: For more information, see "Using Git Functions in SAS" on page 67.

If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

Form 1: **GIT_CLONE** (*'uri'*, *'directory '*<, *'user'*, *'password | personal-access-token'*> <, '', '', *'SSL-CA-certificate-path'*, 0, '', *'customHeader'*>)

Form 2: **GIT_CLONE** (*'uri'*, *'directory'*, *'user'*, *'password'*, *'ssh-public-key-path'*, *'ssh-private-key-path'*<, '', '', '', 0, '', *'custom-header'*>)

Form 3: **GIT_CLONE**(*'uri'*, *'directory'*, '', '', '', '', *'SSL-CA-certificate-path'*, *kerberos*<, *'custom-header'*>)

Form 4: **GIT_CLONE**(*'uri'*, *'directory'*, '', '', '', '', '', *2*, *auth-domain-ID*<, *custom-header*>)

Required Arguments

'uri'
specifies the Git repository URI. The URI can be either HTTPS or SSH.

'directory'
specifies a target directory on the SAS server to clone the repository to.

Requirement: The directory must be empty

Tip: If the specified directory does not exist, it is created if you have the required permissions.

Arguments for Secure HTTPS Connections

'user'
specifies the user name for a secure repository.

'password | personal-access-token'
specifies the password for a secure repository

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

''
specifies an empty string.

'SSL-CA-certificate-path'
specifies the pathname for the SSL certificate. If you do not require an SSL certificate, specify empty quotation marks ('' or "") for this argument.

Arguments for SSH Connections

'user'
specifies the user name for a secure repository.

'password'
specifies the password for a secure repository.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

'ssh-public-key-path'
specifies the pathname for the public SSH key file.

'ssh-private-key-path'
specifies the pathname for the private SSH key file.

Arguments for Kerberos Authentication

''
specifies an empty string.

authentication-option
specifies the authentication method. Specify 1 to use Kerberos authentication. Specify 2 to use the credentials service. Specify 0, or leave the argument blank to not use Kerberos or the credentials service authentication.

Arguments for Credentials Service

authentication-option
specifies the authentication method. Specify 1 to use Kerberos authentication. Specify 2 to use the credentials service. Specify 0, or leave the argument blank to not use Kerberos or the credentials service authentication.

'auth-domain-ID'
specifies the domain ID to retrieve the credentials.

Arguments for Custom Headers

'custom-header'
specifies a custom HTTP header.

Details

Overview

The GIT_CLONE function clones the specified Git repository into a directory on the SAS server. Connections to the Git repository can be made using HTTPS or SSH. If you use an SLL certifcate, then you must specify empty strings for the *ssh-public-*

key and *ssh-private-key* arguments. When using Kerberos authentication the *user*, *password*, *ssh-public-key*, and*ssh-private-key* are ignored. You can specify empty strings (" or "") for the ignored arguments.

Return Values

The GIT_CLONE function has return codes that indicate whether the clone was successful.

Table 3.10 *Return Codes*

Return Codes	Description
–3	The clone operation failed to authenticate.
-1	The clone operation failed because the specified local repository could not be opened.
0	The clone operation succeeded.
2	The clone operation failed because the specified local repository path name was invalid.
3	The clone operation failed because the specified directory is not empty.
12	The clone operation failed because the remote URL was invalid or there was an error with the HTTPS authentication.
23	The clone operation failed because the SSH session could not be authenticated.

Examples

Example 1: Cloning a Git Repository using HTTPS

```
data _null_;
    rc = git_clone (                  /* 1 */
     "https-url",                     /* 2 */
     "target-directory");             /* 3 */
    put rc=;
run;
```

1 Assign a variable to store the return code from SAS.

2 Specify the HTTPS URL of the Git repository to be cloned.

3 Specify the directory that you want to clone the repository into. The directory must be on the SAS server and must be empty.

These statements produce these results:

```
rc=0
```

Example 2: Cloning a Git Repository using SSH

```
data _null_;
    rc = git_clone(                          /*1*/
     "ssh-user@ssh-url",                     /*2*/
     "your-target-directory",
     "ssh-user",                             /*3*/
     "your-ssh-key-password",                /*4*/
     "your-public-key-file",
     "your-private-key-file");
    put rc=;
run;
```

1 Assign a variable to store the return code from SAS.

2 Specify the SSH URL of your Git repository.

3 Specify the SSH user name for the connection. For example, if your SSH URL is “git@github.com:myname/myrepo.git” then the SSH user name is“git”.

4 Specify the password for your SSH key. If your SSH keys are not password protected, specify empty quotation marks (“” or ’’).

These statements produce these results:

```
rc=0
```

Example 3: Cloning a Git Repository using Kerberos

```
data _null_;
    rc = git_clone(
     'uri',
     'your-target-directory',
     '',
     '',
     '',
     '',
     '',
     1);
    put rc=;
run;
```

Example 4: Error in Cloning Repository

In this example, a non-empty target directory is specified. The return code from SAS is 1 and the log writes additional details from Git.

```
data _null_;
    rc = git_clone(                  /*1*/
     "https-url",
     "bad-directory");               /*2*/
```

```
    put rc=;
run;
```

1 Assign a variable to store the return code from SAS.

2 In this example, we specify a non-empty directory on the SAS server to clone the repository to. SAS will set the return code equal to 1, indicating there was a non-library related error with the clone. The SAS log writes additional error details from Git.

These statements produce these results:

```
ERROR: Return code from GIT is (3). 'bad-directory' exists and is not an empty
directory 1
 rc=1
```

1 This error code is returned from Git and has more details than the SAS return code.

GIT_COMMIT_FREE Function

Clears the commit record object that was created by GIT_COMMIT_LOG for the specified repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_COMMIT_FREE(*'directory'*)

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Details

Overview

The GIT_COMMIT_FREE function clears the commit record from memory that is associated with a specified repository. A successful run of this function removes all of the commit objects from a specified repository and allows for new commit objects to be created using the GIT_COMMIT_LOG function.

Return Codes

The GIT_COMMIT_FREE function has return codes that indicate whether the function was successful.

Table 3.11 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git operations can be used.
–2	Indicates that the libgit2 library is available, but the function failed. See the log for details.
0	Indicates that the commit record was successfully cleared.
1	Indicates that no commit record was found for the specified repository. Therefore, no commit record was cleared.

Example: Successfully Clearing a Commit Record

```
data _null_;
 rc = git_commit_free("your-repository");        /* 1 */
 put rc=;                                         /* 2 */
run;
```

1 Specify a target local repository on the SAS server to clear the commit record from.

2 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
NOTE: your-repository commit objects freed successfully.
 rc=0
```

See Also

"GIT_COMMIT_GET Function" on page 784

GIT_COMMIT Function

Commits staged files to the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_COMMIT(*'directory'*, *'update-reference'*, *'author-name'*, *'author-email'*, *'commit-message'*<,*'second-parent-commit-ID'*>)

Required Arguments

'directory'
specifies the pathname of the local Git repository on the SAS server.

'update-reference'
specifies the update reference that you want to commit to.

Note: Specify "HEAD" to commit to the head of the currently checked-out branch.

'author-name'
specifies the name of the commit author.

'author-email'
specifies the email of the commit author.

'commit-message'
specifies the commit message.

Optional Argument

'second-parent-commit-ID'
specifies a second parent commit ID to complete merge commits when resolving conflicts.

Details

Overview

The GIT_COMMIT function commits all staged files to the local repository.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.12 Return Codes

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Commit Files to the Local Repository

```
data _null_;
 rc = git_commit(                          /*1*/
    "your-local-repository",              /*2*/
    "HEAD",                                                  /*3*/
    "your-name",                          /*4*/
    "your-email",                         /*5*/
    "your-commit-message");               /*6*/
   put rc=;
run;
```

1 Assign a variable to the GIT_COMMIT function.

2 Specify the local repository that you want to commit changes to.

3 Specify the update reference that you want to commit changes to. If you want to commit changes to the current update reference, specify "HEAD" for the "*update-reference*" argument.

4 Specify a name for the commit.

5 Specify an email for the commit.

6 Specify a message for the commit.

The preceding statements produce this result:

```
rc=.
```

See Also

"GIT_PUSH Function" on page 812

GIT_COMMIT_GET Function

Returns the specified attribute of the *n*th commit object that is associated with the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_COMMIT_GET(*n*, '*directory*', '*attribute*', *attribute-out*)

Required Arguments

n
specifies the *n*th commit object to retrieve attributes from.

'directory'
specifies the pathname of a local Git repository on the SAS server.

'attribute'
specifies the attribute of the commit object to return.

author
returns the author who submitted the commit.

children_ids
returns a list of the children commit IDs

committer
returns the name of the committer.

committer_email
returns the email of the committer.

email
returns the email of the commit author.

id
returns the commit ID of the commit object.

in_current_branch
returns "True" or "False" to indicate if the commit is in the current branch.

message
returns the commit message.

parent_ids
returns a list of the parent commit IDs.

stash
returns "True" or "False" to indicate if the commit is a stash commit.

time
returns the time of the commit.

attribute-out
specifies a character variable that stores the returned attribute.

Details

Overview

The GIT_COMMIT_GET function returns a specified attribute from the *n*th commit object in the local repository.

Return Codes

There are no return codes for the GIT_COMMIT_GET function.

Example: Retrieve an Attribute From a Commit Record.

```
data _null_;
 length attribute-out $ 1024;          /*1*/
 attribute-out = "";                   /*2*/
 rc = git_commit_get(                 /*3*/
  n,                                   /*4*/
  "directory",                         /*5*/
  "attribute",                         /*6*/
  attribute-out);                      /*7*/
 put attribute-out=;                   /*8*/
run;
```

1 Set a length for a storage variable that stores the returned attribute. Ensure that the length is equal to or greater than the attribute that is returned. This variable is used for the *attribute-out* argument.

2 Initialize the storage character variable.

3 Assign a variable to the GIT_COMMIT_GET function.

4 Specify the numeric integer number of the commit object that you want to retrieve an attribute from. The number of commit objects is found using the GIT_COMMIT_LOG function.

5 Specify the location of the local Git repository that you want to retrieve a status record from.

6 Specify the attribute that you want to retrieve from the status object.

7 Specify the variable that will store the returned attribute from the status object. The variable must be initialized prior to calling the GIT_COMMIT_GET function.

8 Use the PUT statement to display the returned attribute in the SAS log.

The preceding statements produce this result:

```
attribute-out= your-returned-attribute
```

See Also

- “GIT_COMMIT_LOG Function” on page 787
- “GIT_COMMIT_FREE Function” on page 781

GIT_COMMIT_LOG Function

Returns the number of commit objects that are associated with the local repository.

Category: Git

See: For more information, see “Using Git Functions in SAS” on page 67.
If you previously used the GITFN_ functions, see “Deprecated Git Functions”.

Syntax

GIT_COMMIT_LOG(*'directory'*)

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Details

Overview

The GIT_COMMIT_LOG function creates a commit record object and returns the number of commit objects that are associated with the local Git repository.

Return Codes

The GIT_COMMIT_LOG function has return codes that indicate whether the operation was successful.

Table 3.13 *Return Codes*

Return Code	Description
–1	The commit log operation failed. Check the log for details.
≥0	The number of commit objects in the local repository.

Example: Return the Number of Commit Objects in a Local Repository

```
data _null_;
 n = git_commit_log("your-directory");      /*1*/
 put n=;                                      /*2*/
run;
```

1 Assign the GIT_COMMIT_LOG function to a variable. Specify the directory of your local repository.

2 Use the PUT statement to display the return code to the SAS log.

The following log shows that there is 1 commit object in the repository, so the variable *n* returns 1.

```
n=1
```

See Also

- "GIT_COMMIT_GET Function" on page 784
- "GIT_COMMIT_FREE Function" on page 781

GIT_DELETE_REPO Function

deletes a local Git repository and all content within the repository.

Category: Git

See: For more information, see"Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_DELETE_REPO('*directory*')

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Details

Overview

The GIT_DELETE_REPO function deletes a specified local Git repository and all of the contents inside that repository.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.14 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example

Example Code 3.1 *Delete a Local Git Repository from the SAS Server*

```
data _null_;
   rc=git_delete_repo("C:\MyLocalGitRepo");
   put rc=;
run;
```

The preceding statements produce this result:

```
rc=0
```

GIT_DIFF_FILE_IDX Function

Returns the diff for a file in the index.

Category: Git

See: For more informations, see “Using Git Functions in SAS”.
If you previously used the GITFN_ functions, see “Deprecated Git Functions”.

Syntax

GIT_DIFF_FILE_IDX('*directory*', '*file-name*', *diff-content*<, *staged*, '*output-file-path*'>)

Required Arguments

'directory'
specifies the pathname of a local repository on the SAS server.

'file-name'
specifies the file name of the file in the index to return changes for.

diff-content
specifies a character variable to return the content of the diff to.

Optional Arguments

staged
specifies if the target file is staged or is not staged. If no value is specified for *staged*, GIT_DIFF_FILE_IDX defaults to not staged.

0 Indicates the target file is not staged.

1 Indicates the target file is staged.

'output-file-path'
specifies the file path to output the diff content to.

Details

Overview

The GIT_DIFF_FILE_IDX function returns diff of a file in the index against the last committed repository version of that file. Diff content can be stored in a SAS variable specified by the *diff-content* argument. Diff content is returned in JSON format. The variable specified for the *diff_content* argument must be initialized prior to calling the GIT_DIFF_FILE_IDX function. Diff content can also be outputted to a text file by using the optional arguments.

The maximum size of SAS variable is 32,767 bytes. If your diff exceeds the maximum size, the variable *diff-content* will be incomplete and contain invalid JSON. To view diff content larger than 32,767, use the "*output-file-path*" to export the diff content to a text file.

Return Codes

The GIT_DIFF_FILE_IDX function has return codes to indicate whether the function was successful.

Table 3.15 Return Codes

Return Code	Description
–1	Indicates the function failed. See the log for additional details.
0	Indicates the function was successful.

Examples

Example 1: Store the Diff Content of a File in a Variable and Return the Results

```
data _null_;
length diff_content $ 1024;          /*1*/
diff_content="";                     /*2*/
 rc=git_diff_file_idx(
  "C:\User\LocalGitRepo",                      /*3*/
  "your-file-name",                          /*4*/
  diff_content);                     /*5*/
put rc=;                                     /*6*/
put diff_content=;                   /*7*/
run;
```

1 Set the length of your *diff_content* variable high enough to store all of the changes to the file.

2 Initialize the *diff_content* variable.

3 Specify the pathname of your local repository.

4 Specify the local pathname of the file that you want to diff.

5 Specify the variable that will store the content of the diff.

6 Use the PUT statement to write the SAS return code to the log.

7 Use the PUT statement to write the diff content to the log.

The content of your diff will be stored inside your diff-content-variable. Each diff is enclosed in braces {} seen in the example output below:

```
rc=0
 diff_content={ your-diff-content }
```

Example 2: Store the Diff Content of a File in a Text File

```
data _null_;
length diff_content $ 32767;
diff_content="";
rc=gitfn_diff_idx_f(
   "C:\User\LocalGitRepo",
   your-file-name,
```

```
      diff_content,
      0,
      "C:\User\Diff_files\diff.txt");
put rc=;
put diff_content=;
run;
```

SAS still attempts to store the diff content in a SAS variable, but if the content exceeds the 32,767 byte size limit the JSON stored in the variable will be incomplete and invalid. The entire diff is outputted in JSON to your specified text file.

```
rc=0
diff_content={your diff content}
```

GIT_DIFF_FREE Function

Clears the diff record object associated with a local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_DIFF_FREE('*directory*', '*older-commit*', '*newer-commit*')

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'older-commit'
specifies the older of the two commit IDs from the diff.

'newer_commit'
specifies the newer of the two commit IDs from the diff.

Details

Overview

The GIT_DIFF_FREE function clears the diff record that is associated with the local repository between two specific commit IDs. Clearing the diff record frees the system memory used to store the diff record.

Return Codes

The GIT_DIFF_FREE function has return codes that indicate whether the function was successful.

Table 3.16 *Return Codes*

Return Code	Description
–2	Indicates that the libgit2 library is available, but the function failed. See the log for details.
–1	Indicates that the libgit2 library is unavailable and no Git operations can be used.
0	Indicates that the diff record was successfully cleared.
1	Indicates that no diff record was found for the specified repository. Therefore, no diff record was cleared.

Example: Successfully Clearing the Diff Record for a Local Repository

```
data _null_;
 rc = git_diff_free(             /*1*/
  "your-repository",               /*2*/
  "your-older-commit-ID",          /*3*/
  "your-newer-commit-ID");         /*4*/
 put rc=;                          /*5*/
run;
```

1 Assign a variable to the GIT_DIFF_FREE function.

2 Specify a local repository on the SAS server.

3 Specify an older commit ID in the local repository to diff.

4 Specify a newer commit ID in the local repository to diff.

5 Use the PUT statement to return the number of changes to the SAS log.

The preceding statements produce these results:

```
NOTE: Diff objects freed successfully.
rc=0
```

See Also

- “GIT_DIFF_GET Function” on page 795

■ "GIT_DIFF Function" on page 794

GIT_DIFF Function

Returns the number of diffs between two commits in the local repository and creates a diff record object for the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS".

If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_DIFF('*directory*', '*older-commit-ID*', '*newer-commit-ID*')

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'older-commit-ID'
specifies the older of the two commit IDs that you want to diff.

Note: To return a file list for a specific commit ID, specify the string "NULL" for the argument "*older-commit-ID*".

'newer-commit-ID'
specifies the newer of the two commit IDs that you want to diff.

Details

Overview

The GIT_DIFF function returns the number of changes between two commits and creates a diff record object in the local repository.

Return Codes

The GIT_DIFF function has return codes to indicate if the function was successful.

Table 3.17 *Return Codes*

Return Code	Description

–1	The libgit2 library is unavailable and no Git operations can be used.
≥0	The operation was successful and there are n diff objects for the specified commit IDs.

Example: Successfully Returning the Number of Changes between Two Commits

```
data _null_;
 n=git_diff(
  "your-repository",          /*1*/
  "older-commit-ID",          /*2*/
  "newer-commit-ID");         /*3*/
 put n=;                      /*4*/
run;
```

1 Specify a local repository on the SAS server.

2 Specify an older commit ID in the local repository to diff.

3 Specify a newer commit ID in the local repository to diff.

4 Use the PUT statement to return the number of changes to the SAS log.

The following log shows that there is 1 change between the two commits, so the variable *n* returns 1.

```
n=1
```

See Also

"GIT_DIFF_GET Function" on page 795

GIT_DIFF_GET Function

Returns the specified attribute of the *n*th diff object in the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_DIFF_GET(*n*, '*directory*', '*older-commit-ID*', '*newer-commit-ID*, '*attribute*', *attribute-out*)

Required Arguments

n
specifies the *n*th diff object to retrieve an attribute from.

'directory'
specifies the pathname of a local Git repository on the SAS server.

'older-commit-ID'
specifies the older of the two commit IDs that you want to diff.

'newer-commit-ID'
specifies the newer of the two commit IDs that you want to diff.

'attribute'
specifies the attribute of the diff object to return. The following attributes can be returned from the diff:

file
returns the name of the file relative to the local repository that had changes between the two commits.

diff_content
returns the content of the diff for the file.

diff_type
returns the type of change.

attribute-out
specifies a variable that stores the returned attribute.

Details

Overview

The GIT_DIFF_GET function returns the specified attribute from the *n*th diff object in a local Git repository.

Return Codes

The GIT_DIFF_GET function does not have return codes. The value of the assignment variable remains missing after the function is run.

Example: Return an Attribute From the Diff Record

```
data _null_;
```

```
 length attribute-out $ 1024;        /*1*/
 attribute-out="";                   /*2*/
 rc=git_diff_get(                    /*3*/
  n,                                 /*4*/
  "directory",                       /*5*/
  "older-commit-ID",
  "newer-commit-ID",
  "attribute",                       /*6*/
  attribute-out);                    /*7*/
 put attribute-out=;                 /*8*/
run;
```

1 Set a length for a storage variable that stores the returned attribute. Ensure that the length is equal to or greater than the attribute that is returned. This variable is used for the *attribute-out* argument.

2 Initialize the storage variable.

3 Assign a variable to the GIT_DIFF_GET function.

4 Specify the numeric integer number of the commit object that you want to retrieve an attribute from. The number of commit objects is found using the GIT_DIFF function.

5 Specify the location of the local Git repository that you want to retrieve a status record from.

6 Specify the attribute that you want to retrieve from the status object.

7 Specify the variable that will store the returned attribute from the status object. The variable must be initialized prior to calling the GIT_DIFF_GET function.

8 Use the PUT statement to display the returned attribute in the SAS log.

The preceding statements produce this result:

```
attribute-out= returned-attribute
```

See Also

"GIT_DIFF Function" on page 794

GIT_DIFF_TO_FILE Function

Writes the diff content to a file reference.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_DIFF_TO_FILE(*n*, '*directory*', '*older-commit-ID*', '*newer-commit-ID*', '*file-reference*')

Required Arguments

n
specifies the *n*th commit object to retrieve attributes from.

'directory'
specifies the pathname of a local Git repository on the SAS server.

'older-commit-ID'
specifies the older of the two commit IDs that you want to diff.

Note: To return a file list for a specific commit ID, specify the string "NULL" for the argument "*older-commit-ID*".

'newer-commit-ID'
specifies the newer of the two commit IDs that you want to diff.

'file-reference'
specifies the pathname of the file to write the diff content.

Details

Overview

The GIT_DIFF_TO_FILE function writes diff content to a file. Writing diff content to a file bypasses the 32,767 size limit of SAS character variables.

Note: If the file that is specified by the argument "*file-reference*" exists, the contents of the file are overwritten with the diff content. If the file does not exist and you have the correct permissions, a text file is created for you and the diff content is written to that text file.

Return Codes

The GIT_DIFF function has return codes to indicate whether the function was successful.

Table 3.18 *Return Codes*

Return Code	Description
0	The function was successful.

Return Code	Description
–1	An error occurred.

Example

Example Code 3.2 *Run a Diff between Two Commits and Write the Diff Content to a File*

```
data _null_;
   n= git_diff("C:\MyLocalGitRepo",
    "ef350cee6d6507a84577a6a9e19683e8fffab78a",
    "bf735c3a546ad8c228a616219c6b6e3808baa764");
   put n=;
   rc= git_diff_to_file(1,
    "C:\MyLocalGitRepo",
    "ef350cee6d6507a84577a6a9e19683e8fffab78a",
    "bf735c3a546ad8c228a616219c6b6e3808baa764",
    "C:\MyLocalGitRepo\diff_content.txt");
   put rc=;
run;
```

The preceding statements produce these results:

```
n=1
rc=0
```

GIT_FETCH Function

Fetches updates from the remote repository.

Syntax

GIT_FETCH(*'directory'*, *'user'*, *'password'*, *'ssh-public-key-path'*, *'ssh-private-key-path'*<, *'target-branch'*, *'SLL-certificate-path'*, *kerberos*><, *'custom-header'*>)

GIT_FETCH(*'directory'*, '', '', '', '', *'target-branch'*, '', *2*, *'auth-domain-ID'*<, *'custom-header'*>)

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'user'
specifies the user name for a secure repository.

'password'
specifies the password for a secure repository.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

'ssh-public-key-path'
specifies the pathname for the public SSH key file.

'ssh-private-key-path'
specifies the pathname for the private SSH key file.

Optional Arguments

'target-branch'
specifies the name of the remote repository target branch to fetch. If the target branch is not specified, all changes are fetched from the remote repository.

Note: The GIT_FETCH function automatically prepends "origin/" to "*target-branch*". If you want to fetch "origin/MyBranch", you need to specify only the string 'MyBranch' for the argument *'target-branch'*.

'SSL-CA-certificate-path'
specifies the pathname for the SSL certificate. If you do not require an SSL certificate, specify empty quotation marks ('' or "") for this argument.

authentication-option
specifies the authentication method. Specify 1 to use Kerberos authentication. Specify 2 to use the credentials service. Specify 0, or leave the argument blank to not use Kerberos or the credentials service authentication.

'auth-domain-ID'
specifies the domain ID to retrieve the credentials.

'custom-header'
specifies a custom HTTP header.

Details

Overview

The GIT_FETCH function retrieves updates from the remote repository that were pushed by other users. Unlike GIT_PULL, GIT_FETCH does not automatically attempt to merge updates with your local repository.

Return Codes

The GIT_FETCH function has return codes that indicate whether the fetch was successful.

Table 3.19 *Return Codes*

Return Code	Description
–3	The fetch operation failed to authenticate.
–1	The fetch operation failed because the specified local repository could not be opened.
0	The fetch operation succeeded.
2	The fetch operation failed because the specified local repository pathname was invalid.
12	The fetch operation failed because the remote URL was invalid or there was an error with the HTTPS authentication.
23	The fetch operation failed because the SSH session could not be authenticated.

Examples

Example 1: Fetch Updates from a Specified Branch on the Remote Repository Using a User Name and Password

Note: The SSH argument's public key and private key must be specified as empty strings ("" or '') when using user name and password authentication.

```
data _null_;
   rc= git_fetch(
    "C:/MyLocalGitRepo",
    "your-name",
    "your-password",
    "",
    "",
    "branch-name");
   put rc=;
run;
```

The preceding statements produce these results:

```
NOTE: Fetch successful!
rc=0
```

Example 2: Fetch All Updates from the Remote Repository Using SSH Keys

Note: You must specify "git" for the user name argument and an empty string ("" or '') for the password argument when using SSH authentication.

```
data _null_;
   rc= git_fetch(
    "C:/MyLocalGitRepo",
    "git",
    "",
    "C:\My_SSH_Dir\id_rsa_pub",
    "C:\My_SSH_Dir\id_rsa");
   put rc=;
run;
```

The preceding statements produce these results:

```
NOTE: Fetch successful!
rc=0
```

Example 3: Fetch Updates from a Specified Branch Using Kerberos

When using Kerberos authentication, specify values for only *directory*, *branch-name*, and *kerberos*. All other arguments must be empty strings ("" or '').

```
data _null_;
   rc= git_fetch(
    'C:/MyLocalGitRepo',
    '',
    '',
    '',
    '',
    'branch-name',
    '',
    1);
   put rc=;
run;
```

```
NOTE: Fetch successful!
rc=0
```

GIT_INDEX_ADD Function

Stages 1 to *n* number of files to commit to the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_INDEX_ADD(*'directory'*, *'file-1'*, *'status-1'* < , ... ,*'file-n'*, *'status-n'* >)

Required Arguments

'directory'
specifies the path of a cloned Git repository on the SAS server.

'file'
specifies the local pathname of a file to stage in the local repository.

Note: The pathname is relative to your local repository.

'status'
specifies the status of a file to stage in the local repository. The status can be "New", "Modified", or "Deleted".

Details

Overview

The GIT_INDEX_ADD function stages files in the local repository to be committed. To find the status of your files, use the "GIT_STATUS Function".

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.20 Return Codes

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Stage a File in the Local Repository

In this example, we first use the GIT_STATUS function to find the local pathname of two files and their status'. Next, we use the GIT_INDEX_ADD function to stage the two files. Clear the previous status from memory using the GIT_STATUS_FREE function and then confirm that the files were staged by checking the value of the status attribute using the GIT_STATUS function again.

```
data _null_;
   n = git_status("your-
repository");
                                          /*1*/
   rc = git_index_add("your-repository", "your-file1", "file1-status",
"your-file2","file2-status");   /*2*/
   rc = git_status_free("your-repository");
   n = git_status("your-
repository");
                                          /*3*/
run;
```

1 Use the GIT_STATUS function to determine the file's local pathname and the file's status.

2 Stage the file using the GIT_INDEX_ADD function by specifying the local repository and the file name and status of the files that you want to stage.

3 Validate that the file was staged using the GIT_STATUS function and confirming that the value of the staged attribute has changed from False to True.

The preceding statements produce these results:

```
NOTE: Entry: your-file1. Staged: False. Status: file1-status.
NOTE: Entry: your-file2. Staged: False. Status: file2-status.
NOTE: Entry: your-file1. Staged: True. Status: file1-status.
NOTE: Entry: your-file2. Staged: True. Status: file2-status.
```

See Also

- "GIT_INDEX_REMOVE Function" on page 805
- "GIT_STATUS Function"

GIT_INDEX_REMOVE Function

Unstages 1 to *n* number of files to commit to the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_INDEX_REMOVE('*directory*', '*file-1*' < , ..., '*file-n*' >)

Required Arguments

'directory'
specifies the path of a cloned Git repository on the SAS server.

'file'
specifies the local pathname of a file to stage in the local repository.

Details

Overview

The GIT_INDEX_REMOVE function unstages files in the local repository to be committed. You must specify at least one file, but multiple files can be unstaged by specifying additional file arguments.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.21 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Unstage a File in the Local Repository

In this example, we first use the GIT_STATUS function to find the local pathname of a file. Next, we use the GIT_INDEX_REMOVE function to unstage the file. Clear the previous status from memory using the GIT_STATUS_FREE function then confirm that the file was unstaged by checking the value of the Status attribute using the GIT_STATUS function again.

```
data _null_;
   n = git_status("your-repository");                        /*1*/
   rc = git_index_remove("your-repository", "your-file");    /*2*/
   rc= git_status_free("your-repository");
   n = git_status("your-repository");                        /*3*/
run;
```

1 Use the GIT_STATUS function to determine the local pathname of the file.

2 Unstage the file using the GIT_INDEX_REMOVE function by specifying the local repository and the file you want to unstage.

3 Validate that the file was unstaged using the GIT_STATUS function and confirming that the value of the staged attribute has changed from True to False.

The preceding statements produce these results:

```
NOTE: Entry: your-file. Staged: True. Status: your-file-status.
NOTE: Entry: your-file. Staged: False. Status: your-file-status.
```

See Also

"GIT_INDEX_ADD Function" on page 803

GIT_INIT_REPO Function

Initializes a new local Git repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_INIT_REPO('*directory*', '*default-branch-name*'<, '*remote-URL*'>)

Required Arguments

'directory'
specifies a target directory on the SAS server to create a local repository.

'default-branch-name'
specifies the default branch name for the repository.

Optional Argument

'remote-URL'
specifies the remote repository URL to be configured for the local repository. If no remote URL is set, the repository is a local-only repository. Local-only repositories return errors if you use the GIT_PUSH, GIT_PULL, or GIT_FETCH function.

Details

Overview

The GIT_INIT_REPO function creates a local Git repository on the SAS server. If no remote repository URL is specified during initialization, you can use the GIT_SET_URL function on page 828 to set the remote repository URL after initialization"GIT_SET_URL Function" on page 828.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.22 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Initialize a New Local Git Repository on the SAS Server

```
data _null_;
  rc= git_init_repo(
    "C:\User\New_Repo",
    "main",
    "your-remote-URL");
  put rc=;
```

```
run;
```

These statements produce these results:

```
NOTE: C:\User\New_Repo repository successfully initialized.
rc=0
```

GIT_PULL Function

Pulls changes from the remote repository into the local repository.

Category: Git

Note: Starting with 2025.09, the GIT_PULL function supports custom headers for remote functions.

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

Form 1: **GIT_PULL**(*'directory'*)

Form 2: **GIT_PULL**(*'directory'*, *'user'*, *'password'*<, '', '', *'SSL-certificate-path'*, 0, '', *'custom-header'*>)

Form 3: **GIT_PULL**(*'directory'*, *'ssh-user'*, *'ssh-password'*, *'ssh-public-key'*, *'ssh-private-key'*)

Form 4: **GIT_PULL**(*'directory'*, '', '', '', '', *'SSL-certificate-path'*, *kerberos*<, *'custom-header'*>)

Form 5: **GIT_PULL**(*'directory'*, '', '', '', '', '',*2*, *auth-domain-ID*<, *'custom-header'*>)

Required Argument

'directory'
specifies the local repository that you want to pull changes to from the remote repository.

Arguments for User Name-Password Authentication

'user'
specifies the Git user name credential for the remote repository.

'password'
specifies the Git password credential for the remote repository. Passwords can be encrypted using PROC PWENCODE.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

'SSL-CA-certificate-path'
specifies the pathname for the SSL certificate. If you do not require an SSL certificate, specify empty quotation marks ('' or "") for this argument.

''
specifies an empty string.

Arguments for SSH Authentication

'ssh-user'
specifies the SSH user name for the remote repository.

'ssh-password'
specifies the password for your SSH keys. Passwords can be encrypted using PROC PWENCODE.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

Note: If your keys are not password-protected, specify empty quotation marks ("" or '').

'ssh-public-key'
specifies the pathname for the public ssh key file.

'ssh-private-key'
specifies the pathname for the private ssh key file.

Arguments for Kerberos Authentication

''
specifies an empty string.

authentication-option
specifies the authentication method. Specify 1 to use Kerberos authentication. Specify 2 to use the credentials service. Specify 0, or leave the argument blank to not use Kerberos or the credentials service authentication.

Arguments for Credentials Service

authentication-option
specifies the authentication method. Specify 1 to use Kerberos authentication. Specify 2 to use the credentials service. Specify 0, or leave the argument blank to not use Kerberos or the credentials service authentication.

'auth-domain-ID'
specifies the domain ID to retrieve the credentials.

Arguments for Custom Headers

'custom-header'
specifies a custom HTTP header.

Details

Overview

The GIT_PULL function pulls changes from the remote repository to the local repository. If authentication is required, you must specify either your Git account credentials or your SSH credentials.

Return Codes

The GIT_PULL function has return codes that indicate whether the pull was successful.

Table 3.23 *Return Codes*

Return Code	Description
–3	The pull operation failed because there were too many authentication attempts..
–2	The pull operation failed because there was a merge conflict with the local repository.
–1	The pull operation failed because the specified local repository could not be opened.
0	The pull operation was successful and the local repository was updated.
1	The local repository is up-to-date and there are no new changes to pull from the remote repository.
2	The pull operation failed because the specified local repository path name was invalid.
12	The pull operation failed because the remote URL was invalid or there was an error with the HTTPS authentication.

Return Code	Description
23	The pull operation failed because the SSH session could not be authenticated.
34	The pull operation failed because GIT could not find the appropriate mechanism for credentials. (Check that you are using the correct authentication method for your repository.)

Examples

Example 1: Pull to Local Git Repository Using User Name and Password Authentication

```
data _null_;
 rc= git_pull(                        /*1*/
  "your-local-repository",            /*2*/
  "your-Git-username",                /*3*/
  "your-Git-password");               /*4*/
 put rc=;                             /*5*/
run;
```

1 Assign the GIT_PUSH function to a variable.

2 Specify the local repository to pull changes to from the remote repository.

3 Specify your Git user name for authentication to the remote repository.

4 Specify your Git password for authentication to the remote repository.

5 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
NOTE: Head has been fastforwarded to match the remote repository.
 rc=0
```

Example 2: Pull to Local Git Repository Using SSH Authentication

```
data _null_;
 rc= git_pull(                      /*1*/
  "your-local-repository",          /*2*/
  "your-ssh-username",              /*3*/
  "your-ssh-password",              /*4*/
  "your-ssh-public-key",            /*5*/
  "your-ssh-private-key");          /*6*/
 put rc=;                           /*7*/
run;
```

1. Assign the GIT_PULL function to a variable.
2. Specify the local repository to pull changes to from the remote repository.
3. Specify your SSH user name for authentication to the remote repository. This is the same SSH user name that you used to clone the repository with GIT_CLONE.
4. Specify the password for your SSH key. If your SSH keys are not password-protected, specify empty quotation marks ("" or '').
5. Specify the pathname of your public SSH key.
6. Specify the pathname of your private SSH key.
7. Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
NOTE: Head has been fastforwarded to match the remote repository.
 rc=0
```

Example 3: Pull to Local Repository Using Kerberos Authentication

```
data _null_;
 rc= git_pull(
  'your-local-repository',
  '',
  '',
  '',
  '',
  '',
  1);
 put rc=;
run;
```

```
NOTE: Head has been fastforwarded to match the remote repository.
 rc=0
```

See Also

- "GIT_COMMIT Function" on page 782
- "GIT_PUSH Function" on page 812

GIT_PUSH Function

Pushes the committed files in the local repository to the remote repository.

Category: Git

Note: Starting with 2025.09, the GIT_PUSH function supports custom headers for remote functions.

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

Form 1: **GIT_PUSH**(*'directory'*)

Form 2: **GIT_PUSH**(*'directory'*, *'user'*, *'password'* <, '', '', *'SSL-certificate-path'*><, 0, '', *'customHeader'*>)

Form 3: **GIT_PUSH**(*'directory'*, *'ssh-user'*, *'ssh-password'*, *'ssh-public-key'*, *'ssh-private-key'*)

Form 4: **GIT_PUSH**(*'directory'*, '', '', '', '', *'SSL-certificate-path'*, *kerberos*<, '', *'customHeader'*>

Form 5: **GIT_PUSH**(*'directory'*, '', '', '', '', '', *2*, *'auth-domain-ID'*<, *'customHeader'*>)

Required Argument

'directory'
specifies the local repository that you want to push to the remote repository.

Arguments for User-Password Authentication

'user'
specifies your Git user name credential for the remote repository.

'password'
specifies your Git password credential for the remote repository.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

'SSL-CA-certificate-path'
specifies the pathname for the SSL certificate. If you do not require an SSL certificate, specify empty quotation marks ('' or "") for this argument.

Arguments for SSH Authentication

'ssh-user'
specifies the SSH user name for the remote repository.

'ssh-password'
specifies the password for your SSH keys. Passwords can be encrypted using PROC PWENCODE.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

Note: If your keys are not password-protected, specify empty quotation marks ("" or '').

'ssh-public-key'
specifies the pathname for the public ssh key file.

'ssh-private-key'
specifies the pathname for the private ssh key file.

Arguments for Kerberos Authentication

''
specifies an empty string.

authentication-option
specifies the authentication method. Specify 1 to use Kerberos authentication. Specify 2 to use the credentials service. Specify 0, or leave the argument blank to not use Kerberos or the credentials service authentication.

Arguments for Credentials Service

authentication-option
specifies the authentication method. Specify 1 to use Kerberos authentication. Specify 2 to use the credentials service. Specify 0, or leave the argument blank to not use Kerberos or the credentials service authentication.

'auth-domain-ID'
specifies the domain ID to retrieve the credentials.

Arguments for Custom Headers

'custom-header'
specifies a custom HTTP header.

Details

Overview

The GIT_PUSH function pushes changes from the local repository to the remote repository.

Return Codes

The GIT_PUSH function has return codes that indicate whether the push was successful.

Table 3.24 *Return Codes*

Return Code	Description
-3	The push operation failed because there were too many authentication attempts.
-1	The push operation failed because the specified local repository could not be opened.
0	The push operation succeeded.
2	The push operation failed because the specified local repository path name was invalid.
4	The push operation failed because a reference that you are trying to update on the remote contains commits.
12	The push operation failed because the remote URL was invalid or there was an error with the HTTPS authentication.
23	The push operation failed because the SSH session could not be authenticated.
34	The push operation failed because GIT could not find the appropriate mechanism for credentials. (Check that you are using the correct authentication method for your repository.)

Examples

Example 1: Push to Remote Git Repository Using User Name and Password Authentication

```
data _null_;
 rc= git_push(                    /*1*/
  "your-local-repository",        /*2*/
  "your-Git-username",            /*3*/
  "your-Git-password");           /*4*/
run;
```

1 Assign the GIT_PUSH function to a variable.

2 Specify the local repository to push to the remote repository.

3 Specify your Git user name for authentication to the remote repository.

4 Specify your Git password for authentication to the remote repository.

The preceding statements produce these results:

```
NOTE: Push successful!
```

Example 2: Push to Remote Git Repository Using SSH Authentication

```
data _null_;
 rc= git_push(                    /*1*/
  "your-local-repository",        /*2*/
  "your-ssh-user-name",           /*3*/
  "your-ssh-key-password",        /*4*/
  "your-public-ssh-key",          /*5*/
  "your-private-ssh-key");        /*6*/
run;
```

1 Assign the GIT_PUSH function to a variable.

2 Specify the local repository to commit to the remote repository.

3 Specify your SSH user name for authentication to the remote repository. This is the same SSH user name that you used to clone the repository with GIT_CLONE.

4 Specify the password for your SSH key. If your SSH keys are not password-protected, specify empty quotation marks ("" or '').

5 Specify the pathname of your public SSH key.

6 Specify the pathname of your private SSH key.

The preceding statements produce these results:

```
NOTE: Push successful!
```

Example 3: Push to Remote Git Repository Using Kerberos Authentication

```
data _null_;
 rc= git_push(
  'your-local-repository',
  '',
  '',
  '',
  '',
  '',
  1);
run;
```

```
NOTE: Push successful!
```

See Also

- “GIT_COMMIT Function” on page 782
- “GIT_PULL Function” on page 808

GIT_REBASE Function

Rebases your current branch to a specified commit ID.

Category: Git

See: For more information, see “Using Git Functions in SAS” on page 67.
If you previously used the GITFN_ functions, see “Deprecated Git Functions”.

Syntax

GIT_REBASE('*directory*', '*current-branch-commit-ID*', '*rebase-commit-ID*', '*name*', '*email*')

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'current-branch-commit-ID'
specifies the commit ID of the current branch.

'rebase-commit-ID'
specifies the commit ID to rebase your current branch on to.

'name'
specifies the name of the person who is performing the rebase.

'email'
specifies the email of the person who is performing the rebase.

Details

Overview

The GIT_REBASE function rebases your current branch on to a different commit ID.

If a conflict occurs, the conflict must be resolved before continuing the rebase operation. After you resolve the conflicts, stage and commit the files and continue the rebase.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.25 Return Codes

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Rebase Your Current Branch on to a New Base

```
data _null_;
   rc = GIT_REBASE(
    "C:\MyLocalGitRepo",
    "current-branch-commit-ID",
    "rebase-commit-ID",
    "your-name",
    "your-email");
   put rc=;
run;
```

```
NOTE: Rebase successful.
rc=0
```

GIT_REBASE_OP Function

Used to execute rebase operations when a conflict occurs.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_REBASE_OP('*directory*', '*operation*', '*branch-name*', '*name*', '*email*')

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'operation'
specifies the rebase operation to perform. The following operations are valid:

ABORT
cancels the rebase and resets the repository to its previous state before the rebase.

SKIP
skips the conflicting commit that stopped the rebase and continues the rebase.

CONTINUE
continues the rebase after the conflicts have been resolved.

FINISH
finishes the rebase at the current location.

'branch-name'
specifies the name of the branch that was checked out when the rebase started.

'name'
specifies the name of the person who is performing the rebase.

'email'
specifies the email of the person who is performing the rebase.

Details

Overview

The GIT_REBASE_OP function is used to resolve conflicts that can occur during a rebase. If a conflict occurs during a rebase, the rebase is paused until you issue one of the four valid rebase operations.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.26 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Continue the Rebase by Using the "CONTINUE" Operation

```
data _null_;
   rc = git_rebase_op(
    "C:\LocalGitRepo",
    "CONTINUE",
    "main",
    "your-name",
    "your-email");
put rc=;
run;
```

The preceding statements produce this result:

```
rc=0
```

GIT_REF_FREE Function

Clears the reference objects that are associated with the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.

Syntax

GIT_REF_FREE (*'directory'*)

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Details

Return Codes

The GIT_REF_FREE function has return codes that indicate whether the function was successful.

Table 3.27 Return Codes

Return Code	Description
0	The free operation was successful.
1	No reference objects were found for the local repository.

Example: Return the Reference List to a Data Structure

```
data work.refs;
    LENGTH PATH $ 1024;
    LENGTH REFERENCE $ 1024;
    LENGTH ID $ 40;
    PATH="";
    REFERENCE="";
    ID="";
    N=1;

    N=GIT_REF_LIST("<localRepoPath>", 1);
    DO I=1 TO N;
        RC=GIT_REF_GET(i,"<localRepoPath>", "REPO_PATH",PATH);
        RC=GIT_REF_GET(i,"<localRepoPath>", "REFERENCE",REFERENCE);
        RC=GIT_REF_GET(i,"<localRepoPath>", "ID", ID);
        PUT PATH=;
        PUT REFERENCE=;
        PUT ID=;
        output;
    END;
    RC=GIT_REF_FREE("<localRepoPath>");
RUN;
```

GIT_REF_GET Function

Returns the specified attribute of a reference object in the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.

Syntax

GIT_REF_GET (*n*, *'directory'*, *'attribute'*, *attribute-out*

Required Arguments

n
specifies the *n*th reference object to retrieve attributes from.

'directory'
specifies the pathname of a local Git repository on the SAS server.

'attribute'
specifies the attribute of the reference object to retrieve.

repo_path
returns the repository path.

reference
returns the relative path to the reference file within the local repository. The return value always beings with "refs" and ends with the name of the reference.

id
returns the commit ID that is associated with the refence.

attribute-out
specifies a character variable that stores the returned attribute.

Details

Return Codes

The GIT_REF_GET function has return codes that indicate whether the function was successful.

Table 3.28 *Return Codes*

Return Code	Description
–2	No reference objects were found in the specified local repository.
–1	The GIT_REF_GET function encountered an error.
0	The specified reference object and attribute were found and returned.
1	The reference object was found, but the specified attribute was invalid.
2	No reference object exists at the specified index.

Example: Return the Reference List to a Data Structure

```
data work.refs;
    LENGTH PATH $ 1024;
    LENGTH REFERENCE $ 1024;
    LENGTH ID $ 40;
    PATH="";
    REFERENCE="";
    ID="";
    N=1;

    N=GIT_REF_LIST("<localRepoPath>", 1);
    DO I=1 TO N;
        RC=GIT_REF_GET(i,"<localRepoPath>", "REPO_PATH",PATH);
        RC=GIT_REF_GET(i,"<localRepoPath>", "REFERENCE",REFERENCE);
        RC=GIT_REF_GET(i,"<localRepoPath>", "ID", ID);
        PUT PATH=;
        PUT REFERENCE=;
        PUT ID=;
        output;
    END;
    RC=GIT_REF_FREE("<localRepoPath>");
RUN;
```

GIT_REF_LIST Function

Returns the list of reference objects that are associated with the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.

Syntax

GIT_REF_LIST (*'directory'*, <*create-data-set*>)

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Optional Argument

create-data-set

1 returns the reference list in a data structure.

specifies to return the reference list in a data structure stored in memory. The data structure is accessed using the GIT_REF_GET function, and the data structure must be cleared using the GIT_REF_FREE function.

Details

Return Codes

The GIT_REF_LIST function has return codes that indicate whether the function was successful.

Table 3.29 *Return Codes*

Return Code	Description
–1	The GIT_REF_LIST function failed.
n >= 0	The number of reference objects that are associated with the local repository. The value can be 0 or greater.

Example: Return the Reference List to a Data Structure

```
data work.refs;
    LENGTH PATH $ 1024;
    LENGTH REFERENCE $ 1024;
    LENGTH ID $ 40;
    PATH="";
    REFERENCE="";
    ID="";
    N=1;

    N=GIT_REF_LIST("<localRepoPath>", 1);
    DO I=1 TO N;
        RC=GIT_REF_GET(i,"<localRepoPath>", "REPO_PATH",PATH);
        RC=GIT_REF_GET(i,"<localRepoPath>", "REFERENCE",REFERENCE);
        RC=GIT_REF_GET(i,"<localRepoPath>", "ID", ID);
        PUT PATH=;
        PUT REFERENCE=;
        PUT ID=;
        output;
    END;
    RC=GIT_REF_FREE("<localRepoPath>");
RUN;
```

GIT_RESET_FILE Function

Resets a file in the index to the local repository version.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_RESET_FILE(*'directory'*, *'file-name'*)

Required Arguments

'directory'
specifies the pathname of a local repository on the SAS server.

'file-name'
specifies the local name of a file in the index to reset.

Details

Overview

The GIT_RESET_FILE function resets a file in the index to the last commit version in the local repository.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.30 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Reset a File in the Local Repository.

```
data _null_;
 rc = git_reset_file(
  "your-local-repository",      /*1*/
  "your-file");                 /*2*/
 put rc=;                       /*3*/
run;
```

1 Specify your local repository on the SAS server.

2 Specify the file to reset in the local repository.

3 Use the PUT statement to output the SAS return code to the log.

Thee preceding statements produce these results:

```
NOTE: your-file reset successfully.
 rc=0
```

GIT_RESET Function

Resets the local repository to a specified commit.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_RESET('*directory*', '*commit-ID*', '*reset_type*')

Required Arguments

'directory'
specifies the pathname of a cloned Git repository on the SAS server.

'commit-ID'
specifies the commit ID to reset the local repository to.

'reset_type'
specifies the type of reset.

Details

Overview

The GIT_RESET function resets the local repository to a specified commit from the repository's commit history. The type of reset specified by the *reset_type* argument determines what happens to the local repository.

Untracked files or new files that have not been staged in your repository can not be reset using the GIT_RESET function. To reset and untracked file you can stage the file and use the GIT_RESET function, or delete the file from your repository.

Soft
: resets the current branch to the prior commit. No changes are made to the staged changes (the index) or your working directory.

Mixed
: resets the current branch to the prior commit and resets your staged changes (the index). No changes are made to your working directory.

Hard
: resets the current branch to the prior commit, resets your staged changes (the index), and discards all changes in your working directory. Untracked files are not reset. You can not undo a hard reset.

Return Codes

The GIT_RESET function does not have return codes. A successful reset returns a note to the log stating that the reset was successful.

Example: Reset the local repository to a specified commit ID

```
data _null_;
rc = git_reset(
  "your-local-repository",        /*1*/
  "your-commit-ID",               /*2*/
  "your-reset-type");             /*3*/
run;
```

1 Specify your local Git repository.

2 Specify a commit ID to reset to.

3 Specify the type of reset to perform.

The preceding statements produce this result:

```
NOTE: your-reset-type reset to commit:your-commit-ID successful.1
```

1 A successful reset returns a SAS note to the log stating the reset-type, the commit-ID, and a message stating the reset was successful.

GIT_SET_URL Function

Sets the remote repository URL for a local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_SET_URL('*directory*', '*URL*')

Required Arguments

'directory'
specifies the pathname of the local Git repository on the SAS server.

'URL'
specifies the URL of the remote repository.

Details

Overview

The GIT_SET_URL function sets the remote repository URL for a local Git repository.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.31 *Return Codes*

Return Code	Description
–1	The Git operation failed.
0	The Git operation was successful.

Example: Set the Remote Repository URL for a Local Repository

```
data _null_;
   rc = git_set_url(
    "C:\User\LocalGitRepo",
    "your-remote-URL");
put rc=;
run;
```

The preceding statements produce these results:

```
NOTE: Remote URL: your-remote-URL successfully set for repository:
      C:\User\LocalGitRepo.
rc=0
```

GIT_STASH_APPLY Function

Applies file changes that are stored in a stash to the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_STASH_APPLY('*directory*'<, *stash-index*>)

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Optional Argument

stash-index
specifies the index of the stash to apply. If no stash index is specified, the GIT_STASH_APPLY function defaults to the top of the stash stack that corresponds to the index position *stash-index*=0.

Details

Overview

The GIT_STASH_APPLY function applies the contents of the stash stack at a specified index position. If no index is specified for the optional argument *stash-index*, the GIT_STASH_APPLY function applies the top of the stash stack by default.

Return Codes

The GIT_STASH_APPLY function uses return codes to indicate whether the Git operation was successful.

Table 3.32 *Return Codes*

Return Code	Description
–22	The stash apply failed because there were uncommitted changes in the index.
–13	The stash apply failed and local changes are overwritten.
–1	The stash apply failed because the specified local repository could not be opened.
0	The stash apply was successful.

Examples

Example 1

```
data _null_;
   rc = git_stash_apply("C:\LocalGitRepo");
   put rc=;
run;
```

```
NOTE: C:\LocalGitRepo stash successfully applied.
rc=0
```

Example 2

```
data _null_;
   rc = git_stash_apply("C:\LocalGitRepo",1);
```

```
   put rc=;
run;
```

```
NOTE: C:\LocalGitRepo stash successfully applied.
rc=0
```

GIT_STASH_DROP Function

Drops the contents of the stash stack at the specified index.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_STASH_DROP('*directory*'<, *stash-index*>)

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Optional Argument

stash-index
specifies the index of the stash to drop. If no stash index is specified, the GIT_STASH_DROP function defaults to the top of the stash stack that corresponds to the index position *stash-index*=0.

Details

Overview

The GIT_STASH_DROP function drops the contents of the stash stack at a specified index position. If no index is specified for the optional argument *stash-index*, the GIT_STASH_DROP function uses the top of the stash stack by default.

Return Codes

The GIT_STASH_DROP function has return codes to indicate whether the function was successful.

Table 3.33 Return Codes

Return Code	Description
0	The stash drop was successful.
–1	The stash drop failed.

Examples

Example 1: Drop the Top Stash from the Stash Stack

```
data _null_;
   rc = git_stash_drop("C:\LocalGitRepo");
   put rc=;
run;
```

The preceding statements produce these results:

```
NOTE: C:\LocalGitRepo stash successfully dropped.
rc=0
```

Example 2: Drop the Stash at Stash Index Position 1

```
data _null_;
   rc = git_stash_drop("C:\LocalGitRepo",1);
   put rc=;
run;
```

The preceding statements produce these results:

```
NOTE: C:\LocalGitRepo stash successfully dropped.
rc=0
```

GIT_STASH Function

Stashes file changes that have not been committed.

Category: Git

See: For more information, see “Using Git Functions in SAS” on page 67.
If you previously used the GITFN_ functions, see “Deprecated Git Functions”.

Syntax

GIT_STASH('*directory*', '*name*', '*email*')

Required Arguments

'directory'
specifies the pathname of a local Git repository on the SAS server.

'name'
specifies the name of the person who is performing the stash.

'email'
specifies the email of the person who is performing the stash.

Details

Overview

The GIT_STASH function stores any uncommitted changes made to your files for later use. The changes that are stored in your stash can be reapplied by using the GIT_STASH_APPLY function. The changes that are stored in your stash can be deleted by using the GIT_STASH_DROP function. Your most recent stash is located at the top of your stash stack. If you execute multiple stashes without dropping the stashes, the stashes are stored sequentially in the stash stack. The stash stack is accessed by using the stash index. The most recent stash is stored at index position 0, and any previous stashes are stored at index positions 1 to N.

Return Codes

The SAS Git functions use return codes to indicate whether the Git operation was successful.

Table 3.34 *Return Codes*

Return Code	Description
-2	The Git operation failed because there was nothing to stash.
-1	The Git operation failed.
0	The Git operation was successful.

Example

```
data _null_;
```

```
    rc = git_stash(
     "C:\LocalGitRepo",
     "your-name",
     "your-email");
    put rc=;
run;
```

The preceding statements produce these results:

```
NOTE: C:\LocalGitRepo stash successful.
rc=0
```

GIT_STASH_POP Function

Applies the changes that are stored in the stash, and then drops the contents of the stash.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_STASH_POP('*directory*')

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Details

Overview

The GIT_STASH_POP function applies the changes that are currently stored in the stash to the local repository. After the changes are applied, the contents of the stash are dropped. If multiple stashes exist in your repository, the most recent stash is applied and then dropped. The GIT_STASH_POP function is functionally similar to combining the GIT_STASH_APPLY function and the GIT_STASH_DROP function.

Return Codes

The GIT_STASH_POP function uses return codes to indicate whether the Git operation was successful.

Table 3.35 Return Codes

Return Code	Description
–22	The stash pop failed because there were uncommitted changes in the index.
–13	The stash pop failed and local changes are overwritten.
–1	The stash pop failed because the specified local repository could not be opened.
0	The stash pop was successful.

Example: Pop the Top Stash in the Stash Stack

```
data _null_;
   rc = git_stash_pop("C:\LocalGitRepo");
   put rc=;
run;
```

The preceding statements produce these results:

```
NOTE: C:\LocalGitRepo stash successfully popped.
rc=0
```

GIT_STATUS_FREE Function

Clears the status record object that was created by GIT_STATUS for the specified repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_STATUS_FREE ('*directory*')

Required Argument

'directory'
specifies the pathname of a cloned Git repository on the SAS server.

Details

Overview

The GIT_STATUS_FREE function clears the status record that is associated with a specified repository.

Return Codes

The GIT_STATUS_FREE function has return codes that indicate whether the function was successful.

Table 3.36 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git operations can be used.
–2	Indicates that the libgit2 library is available, but the function failed. See the log for details.
0	Indicates that the status record was successfully cleared.
1	Indicates that no status record was found for the specified repository. Therefore, no status record could be cleared.

Example: Successfully Clearing a Status Record

```
data _null_;
 rc = git_status_free("your-repository");        /* 1 */
 put rc=;                                        /* 2 */
run;
```

1 Specify a target local repository on the SAS server to clear a status record from.

2 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce this result:

```
rc=0
```

See Also

- “GIT_STATUS Function” on page 837
- “GIT_STATUS_GET Function” on page 838

GIT_STATUS Function

Returns the status objects for files in the local repository and creates a status record.

Category: Git

See: For more information, see “Using Git Functions in SAS” on page 67.
If you previously used the GITFN_ functions, see “Deprecated Git Functions”.

Syntax

GIT_STATUS('*directory*')

Required Argument

'directory'
specifies the pathname of a local Git repository on the SAS server.

Details

Overview

The GIT_STATUS function returns the number of status objects in your local repository and creates a status record. The function also writes a note to the SAS log for each status object in the local repository. The note returns the attributes below:

Entry
name of the file.

Staged
Boolean operator that indicates whether the file is staged or unstaged.

Status
state of the file. The status can be “New”, “Modified”, or “Deleted”.

Return Codes

The GIT_STATUS function has return codes that indicate whether the status operation was successful.

Table 3.37 Return Codes

Return Code	Description
–2	The libgit2 library is available, but the status function failed. See the log for details.
–1	The libgit2 library is unavailable and no Git operations can be used.
n	The operation was successful and there are n status objects in the local repository.

Example: Local Repository Has One Status Object

```
data _null_;
    n = git_status("your-local-repository-directory");
    put n=;
run;
```

The following log shows that the local repository has one status object for the new file (*your-file-name*) that has not been staged. Because there is one object, the *n* variable equals 1.

```
NOTE: Entry: your-file-name. Staged: False. Status: New.
n=1
```

See Also

- "GIT_STATUS_FREE Function" on page 835
- "GIT_STATUS_GET Function" on page 838

GIT_STATUS_GET Function

Returns the specified attribute of the *n*th status object returned from GITFN_STATUS() in the local repository.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_STATUS_GET(*n*, '*directory*', '*attribute*', *attribute-store*)

Required Arguments

n
specifies the *n*th status object to retrieve attributes from.

'directory'
specifies the pathname of a local Git repository on the SAS server.

'attribute'
specifies the attribute of the status object to return. Here are the valid values for attribute:

path
returns the file path of the file relative to the local repository.

status
returns the status, "New", "Modified", or "Deleted" of file.

staged
returns "True" or "False" indicating if the file is staged or not.

attribute-store
specifies a character variable that stores the returned attribute.

Details

Overview

The GIT_STATUS_GET function returns the specified attribute from the *n*th status object in a local Git repository.

Return Codes

The GIT_STATUS_GET function has return codes that indicate whether the function was successful.

Table 3.38 *Return Codes*

Return Code	Description
–2	Indicates that the libgit2 library is available, but no status objects were found in the local repository. The GIT_STATUS function should be run first to create status objects for the local repository.
–1	Indicates that the libgit2 library is unavailable and no Git functions can be used.

Return Code	Description
0	Indicates that the specified *n*th status object was found and the specified attribute was returned.
1	Indicates that the specified *n*th status object was found, but the specified attribute was invalid.
2	Indicates that the specified repository has a status record, but the specified *n*th status object does not exist.

Example: Return the PATH Attribute from a Status Object

In this example, we have previously successfully run the GITFN_STATUS function on the local repository to create a status record. Now we can use the GIT_STATUS_GET function to return an attribute from a status object in our local repository.

```
data _null_;
 length attribute-store $ 1024;      /*1*/
 attribute-store="";                 /*2*/
 rc = git_status_get(                /*3*/
        n,                           /*4*/
       "directory",                  /*5*/
       "attribute",                  /*6*/
        attribute-store);            /*7*/
 put varstore=;                      /*8*/
 put rc=;                            /*9*/
run;
```

1 Set a length for a storage variable that stores the returned attribute. Ensure that the length is equal to or greater than the attribute that is returned. This variable is used for the *attribute-store* argument.

2 Initialize the storage character variable.

3 Assign a variable to store the return code from SAS.

4 Specify the numeric integer number of the status object that you want to retrieve an attribute from. Status objects are listed in numeric order in the SAS log using the GIT_STATUS function.

5 Specify the location of the local Git repository that you want to retrieve a status record from.

6 Specify the attribute that you want to retrieve from the status object. Valid attributes are "PATH", "STAGED", or "STATUS".

7 Specify the variable that will store the returned attribute from the status object. The variable must be initialized prior to calling the GIT_STATUS_GET function.

8 Use the PUT statement to display the returned attribute in the SAS log.

9 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
varstore=your-returned-attribute
 rc=0
```

GIT_VERSION Function

Specifies whether libgit2 is available and if available, specifies the version that is being used.

Category: Git

See: For more information, see "Using Git Functions in SAS" on page 67.
If you previously used the GITFN_ functions, see "Deprecated Git Functions".

Syntax

GIT_VERSION()

Without Arguments

The GIT_VERSION function has no arguments. If you try to pass arguments, an error occurs.

Details

Return Values

The GIT_VERSION function has two possible return values:

Table 3.39 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git functions can be used.
version	Indicates the version of the libgit2 library that is being used.

Example

```
data _null_;
   version = git_version();        /*1*/
   put version=;                   /*2*/
run;
```

1 Assign a variable to store the return code from SAS.

2 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce this result:

```
version=0.27
```

GITFN_CLONE Function

Clones a Git repository into a directory on the SAS server.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

Form 1: **GITFN_CLONE** (*uri*, *directory*)

Form 2: **GITFN_CLONE** (*uri*, *directory*, <*user*, *password*, *ssh-public-key*, *ssh-private-key*>)

Required Arguments

uri
specifies the Git repository URI. The URI can be either HTTPS or SSH.

directory
specifies a target directory on the SAS server to clone the repository to.

Requirement: The directory must be empty

Tip: If the specified directory does not exist, it is created if you have the required permissions.

Arguments for SSH Connections

user
specifies the user name for a secure repository.

password
specifies the password for a secure repository.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

ssh-public-key-path
specifies the pathname for the public SSH key file.

ssh-private-key-path
specifies the pathname for the private SSH key file.

Details

Overview

The GITFN_CLONE function clones the specified Git repository into a directory on the SAS server. Connections to the Git repository can be made using HTTPS or SSH.

Return Values

The GITFN_CLONE function has return codes that indicate whether the clone was successful.

Table 3.40 *Return Codes*

Return Codes	Description
-1	Indicates that the libgit2 library is unavailable and no Git functions can be used.
0	Indicates that the clone operation succeeded.
>0	Indicates that the libgit2 library is available, but the operation failed. See the SAS log for details.

Examples

Example 1: Cloning a Git Repository using HTTPS

```
data _null_;
    rc = gitfn_clone (                  /*1*/
     "https-url",                       /*2*/
     "target-directory");               /*3*/
    put rc=;                            /*4*/
run;
```

1 Assign a variable to store the return code from SAS.

2 Specify the HTTPS URL of the Git repository to be cloned.

3 Specify the directory that you want to clone the repository into. The directory must be on the SAS server and must be empty.

4 Use the PUT statement to display the SAS return code in the SAS log.

These statements produce this result:

```
rc=0
```

Example 2: Cloning a Git Repository using SSH

```
data _null_;
    rc = gitfn_clone(                   /*1*/
     "ssh-user@ssh-url",                /*2*/
     "your-target-directory",
     "ssh-user",                        /*3*/
     "your-ssh-key-password",           /*4*/
     "your-public-key-file",
     "your-private-key-file");
    put rc=;                            /*5*/
run;
```

1 Assign a variable to store the return code from SAS.

2 Specify the SSH URL of your Git repository.

3 Specify the SSH user name for the connection. For example, if your SSH URL is “git@github.com:myname/myrepo.git” then the SSH user name is“git”.

4 Specify the password for your SSH key. If your SSH keys are not password protected, specify empty quotation marks ("" or '').

5 Use the PUT statement to display the SAS return code in the SAS log.

These statements produce this result:

```
rc=0
```

Example 3: Error in Cloning Repository

In this example, a non-empty target directory is specified. The return code from SAS is 1 and the log writes additional details from Git.

```
data _null_;
    rc = gitfn_clone(                  /*1*/
     "https-url",
     "bad-directory");                 /*2*/
    put rc=;                           /*3*/
run;
```

1 Assign a variable to store the return code from SAS.

2 In this example, we specify a non-empty directory on the SAS server to clone the repository to. SAS will set the return code equal to 1, indicating there was a non-library related error with the clone. The SAS log writes additional error details from Git.

3 Use the PUT statement to display the SAS return code in the SAS log.

These statements produce these results:

```
ERROR: Return code from GIT is (3). 'bad-directory' exists and is not an empty
directory 1
 rc=1
```

1 This error code is returned from Git and has more details than the SAS return code.

GITFN_CO_BRANCH Function

Check out a branch in a Git repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_CO_BRANCH(*directory*, *branch*)

Required Arguments

directory
specifies the path to a local Git repository on the SAS server.

branch
specifies the name of a branch in the local Git repository.

Details

Overview

The GITFN_CO_BRANCH function checks out a specified branch in the local repository. Checking out a new branch changes the contents of the local repository on the SAS server to match the specified branch's contents.

Return Codes

The GITFN_CO_BRANCH function has return codes to indicate whether the function was successful.

Table 3.41 *Return Codes*

Return Code	Description
–1	Indicates the specified branch was not found.
0	Indicates branch was successfully checked out.

Example

```
data _null_;
 rc= GITFN_CO_BRANCH(
  "your-local-repository",        /*1*/
  "your-target-branch");          /*2*/
  put rc=;                        /*3*/
run;
```

1 Specify your local repository.

2 Specify your target Git branch to check out.

3 Use the PUT statement to display the SAS return code to the log.

The preceding statements produce these results:

```
NOTE: Branch "your-target-branch" successfully checked out.
rc=0
```

GITFN_COMMIT_GET Function

Returns the specified attribute of the *n*th commit object that is associated with the local repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_COMMIT_GET(*n*, *directory*, *attribute*, *attribute-out*)

Required Arguments

n
specifies the *n*th commit object to retrieve attributes from.

directory
specifies the path of a cloned Git repository on the SAS server.

attribute
specifies the attribute of the status object to return.

attribute-out
specifies a character variable that stores the returned attribute.

Details

Overview

The GITFN_COMMIT_GET function returns a specified attribute from the *n*th commit object in the local repository. Here are the valid attributes:

Table 3.42 *Commit Object Attributes*

Attribute	Description
COMMIT_ID	Specifies the commit ID of the commit object.
AUTHOR	Specifies the author who submitted the commit.
EMAIL	Specifies the email of the commit author.

Attribute	Description
MESSAGE	Specifies the commit message.
PARENT_IDS	Specifies the parent commit IDs of the commit object.
TIME	Specifies the time of the commit.

Return Codes

There are no return codes for the GITFN_COMMIT_GET function.

Example: Retrieve an Attribute From a Commit Record.

```
data _null_;
 length attribute-out $ 1024;          /* 1 */
 attribute-out = "";                   /* 2 */
 rc = GITFN_COMMIT_GET(                /* 3 */
  n,                                   /* 4 */
  "directory",                         /* 5 */
  "attribute",                         /* 6 */
  attribute-out);                      /* 7 */
 put attribute-out=;                   /* 8 */
run;
```

1 Set a length for a storage variable that stores the returned attribute. Ensure that the length is equal to or greater than the attribute that is returned. This variable is used for the *attribute-out* argument.

2 Initialize the storage character variable.

3 Assign a variable to the GITFN_COMMIT_GET function.

4 Specify the numeric integer number of the commit object that you want to retrieve an attribute from. The number of commit objects is found using the GITFN_COMMIT_LOG function.

5 Specify the location of the local Git repository that you want to retrieve a status record from.

6 Specify the attribute that you want to retrieve from the status object.

7 Specify the variable that will store the returned attribute from the status object. The variable must be initialized prior to calling the GITFN_COMMIT_GET function.

8 Use the PUT statement to display the returned attribute in the SAS log.

The preceding statements produce this result:

```
attribute-out= your-returned-attribute
```

See Also

- “GIT_COMMIT_LOG Function” on page 787
- “GIT_COMMIT_FREE Function” on page 781

GITFN_COMMITFREE Function

Clears the commit record object that was created by GITFN_COMMIT_LOG for the specified repository.

Category: Git

See: For more information see “Using Git Functions in SAS”.

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see “Deprecated Git Functions”.

Syntax

GITFN_COMMITFREE(*directory*)

Required Argument

directory
specifies the pathname of a cloned Git repository on the SAS server.

Details

Overview

The GITFN_COMMITFREE function clears the commit record that is associated with a specified repository. A successful run of this function removes all of the commit objects from a specified repository and allows for new commit objects to be created using the GITFN_COMMIT_LOG function.

Return Codes

The GITFN_COMMITFREE function has return codes that indicate whether the function was successful.

Table 3.43 Return Codes

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git operations can be used.
–2	Indicates that the libgit2 library is available, but the function failed. See the log for details.
0	Indicates that the commit record was successfully cleared.
1	Indicates that no commit record was found for the specified repository. Therefore, no commit record was cleared.

Example: Successfully Clearing a Commit Record

```
data _null_;
 rc = gitfn_commitfree("your-repository");        /*1*/
 put rc=;                                         /*2*/
run;
```

1 Specify a target local repository on the SAS server to clear the commit record from.

2 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
NOTE: your-repository commit objects freed successfully.
 rc=0
```

See Also

- “GIT_COMMIT_GET Function” on page 784
- “GIT_COMMIT_GET Function” on page 784

GITFN_COMMIT_LOG Function

Returns the number of commit objects that are associated with the local repository.

Category: Git

See: For more information see “Using Git Functions in SAS”.

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_COMMIT_LOG(*directory*)

Required Argument

directory
specifies the pathname of a cloned Git repository on the SAS server.

Details

Overview

The GITFN_COMMIT_LOG function creates a commit record object and returns the number of commit objects that are associated with the local Git repository.

Return Codes

The GITFN_COMMIT_LOG function has return codes that indicate whether the operation was successful.

Table 3.44 *Return Codes*

Return Code	Description
–1	The libgit2 library is unavailable and no Git operations can be used.

Example: Return the Number of Commit Objects in a Local Repository

```
data _null_;
 n = GITFN_COMMIT_LOG("your-directory");      /*1*/
 put n=;                                      /*2*/
run;
```

1 Assign the GITFN_COMMIT_LOG function to a variable. Specify the directory of your local repository.

2 Use the PUT statement to display the return code to the SAS log.

The following log shows that there is 1 commit object in the repository, so the variable *n* returns 1.

```
n= 1
```

See Also

- "GIT_COMMIT_GET Function" on page 784
- "GIT_COMMIT_FREE Function" on page 781

GITFN_COMMIT Function

Commits staged files to the local repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_COMMIT(*directory*, *update-ref*, *author-name*, *author-email*, *commit-message*)

Required Arguments

directory
specifies the pathname of the local Git repository on the SAS server.

update-ref
specifies the update reference that you want to commit to.

author-name
specifies the name of the commit author.

author-email
specifies the email of the commit author.

commit-message
specifies the commit message.

Details

Overview

The GITFN_COMMIT function commits all staged files to the local repository.

Return Codes

There are no return codes from SAS to indicate a successful commit. However, because GITFN_COMMIT is a function, you still must assign it to a variable. The assignment variable is set to missing. The GITFN_STATUS function must be used to confirm that the commit was successful. A successful commit is indicated by the file or files no longer appearing in the status record when GITFN_STATUS is called on the local repository.

Example: Commit Files to the Local Repository

```
data _null_;
 rc = gitfn_commit(                    /* 1 */
    "your-local-repository",           /* 2 */
    "your-update-reference",           /* 3 */
    "your-name",                       /* 4 */
    "your-email",                      /* 5 */
    "your-commit-message");            /* 6 */
   put rc=;
run;
```

1 Assign a variable to the GITFN_COMMIT function.

2 Specify the local repository that you want to commit changes to.

3 Specify the update reference that you want to commit changes to. If you want to commit changes to the current update reference, specify "HEAD" for the *update-ref* argument.

4 Specify a name for the commit.

5 Specify an emailfor the commit.

6 Specify a message for the commit.

The preceding statements produce this result:

```
rc=.
```

See Also

"GIT_PUSH Function" on page 812

GITFN_DEL_BRANCH Function

Deletes a Git branch in the repository.

Category: Git

See: For more information see"Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_DEL_BRANCH(*directory*, *branch-name*)

Required Arguments

directory
specifies the pathname of a cloned Git repository on the SAS server.

branch-name
specifies the name of the branch to be deleted.

Details

Overview

The GITFN_DEL_BRANCH function deletes a branch in the local Git repository.

Return Codes

The GITFN_DEL_BRANCH function has return codes to indicate whether the function was successful. If an error occurs, check the log for additional return codes from Git.

Table 3.45 *Return Codes*

Return Code	Description
–1	Indicates that the delete branch function failed because it is the current branch. See the log for additional information from Git.
0	Indicates that the branch was successfully deleted. A note in the log is displayed stating "Branch *branch-name* was successfully deleted."

Example: Delete a Branch in the Local Repository.

```
data _null_;
 rc= gitfn_del_branch(
  "your-local-repository",       /*1*/
  "your-branch-name");           /*2*/
 put rc=;                        /*3*/
run;
```

1 Specify your local Git repository.

2 Specify the name of the Git branch that you want to delete.

3 Use the PUT statement to display the SAS return code to the log.

The preceding statements produce these results:

```
NOTE: Branch "your-branch-name" successfully deleted.
 rc=0
```

GITFN_DEL_REPO Function

Deletes a local Git repository and its contents.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_DEL_REPO(*directory*)

Required Argument

directory
specifies the pathname of a cloned Git repository on the SAS server.

Details

Overview

The GITFN_DEL_REPO function deletes the specified local repository and all of the repository's contents from the SAS server.

Return Codes

The GITFN_DEL_REPO function has two return codes that indicate whether the function was successful.

Table 3.46 *Return Codes*

Return Code	Description
0	Indicates that the local repository was successfully deleted.
–1	Indicates that the local repository was not deleted.

Example: Successfully Deleting a Local Repository

```
data _null_;
 rc = GITFN_DEL_REPO("your-local-repository");
 put rc=;
run;
```

The preceding statements produce this result:

```
rc=0
```

GITFN_DIFF_FREE Function

Clears the diff record object associated with a local repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_DIFF_FREE(*directory*, *older-commit*, *newer-commit*)

Required Arguments

directory
specifies the pathname of a cloned Git repository on the SAS server.

older-commit
specifies the older of the two commit IDs from the diff.

newer_commit
specifies the newer of the two commit IDs from the diff.

Details

Overview

The GITFN_DIFF_FREE function clears the diff record that is associated with the local repository between two specific commit IDs.

Return Codes

The GITFN_DIFF_FREE function has return codes that indicate whether the function was successful.

Table 3.47 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git operations can be used.
–2	Indicates that the libgit2 library is available, but the function failed. See the log for details.
0	Indicates that the diff record was successfully cleared.
1	Indicates that no diff record was found for the specified repository. Therefore, no diff record was cleared.

Example: Successfully Clearing the Diff Record for a Local Repository

```
data _null_;
```

```
 rc = GITFN_DIFF_FREE(              /*1*/
  "your-repository",                /*2*/
  "your-older-commit-ID",           /*3*/
  "your-newer-commit-ID");          /*4*/
 put rc=;                           /*5*/
run;
```

1 Assign a variable to the GITFN_DIFF_FREE function.

2 Specify a local repository on the SAS server.

3 Specify an older commit ID in the local repository to diff.

4 Specify a newer commit ID in the local repository to diff.

5 Use the PUT statement to return the number of changes to the SAS log.

The preceding statements produce these results:

```
NOTE: Diff objects freed successfully.
rc=0
```

See Also

- “GIT_DIFF_GET Function” on page 795
- “GIT_DIFF Function” on page 794

GITFN_DIFF_GET Function

Returns the specified attribute of the *n*th diff object in the local repository.

Category: Git

See: For more information see “Using Git Functions in SAS”.

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see “Deprecated Git Functions”.

Syntax

GITFN_DIFF_GET(*n*, *directory*, *attribute*, *attribute-out*)

Required Arguments

n

specifies the *n*th diff object to retrieve an attribute from.

directory
specifies the pathname of a cloned Git repository on the SAS server.

attribute
specifies the attribute of the diff object to return.

attribute-out
specifies a variable that stores the returned attribute.

Details

Overview

The GITFN_DIFF_GET function returns the specified attribute from the *n*th diff object in a local Git repository. Here are the valid attributes:

file
the name of the file that had changes between the two commits.

diff_content
the content of the diff for the file.

diff_type
the type of change.

Return Codes

The GITFN_DIFF_GET function does not have return codes. The value of the assignment variable remains missing after the function is run.

Example: Return an Attribute From the Diff Record

```
data _null_;
 length attribute-out $ 1024;        /* 1 */
 attribute-out="";                   /* 2 */
 rc=gitfn_diff_get(                  /* 3 */
  n,                                 /* 4 */
  "directory",                       /* 5 */
  "older-commit",                    /* 6 */
  "newer-commit",                    /* 7 */
  "attribute",                       /* 8 */
  attribute-out);                    /* 9 */
 put attribute-out=;                 /* 10 */
run;
```

1 Set a length for a storage variable that stores the returned attribute. Ensure that the length is equal to or greater than the attribute that is returned. This variable is used for the *attribute-out* argument.

2 Initialize the storage variable.

3 Assign a variable to the GITFN_DIFF_GET function.

4 Specify the numeric integer number of the commit object that you want to retrieve an attribute from. The number of commit objects is found using the GITFN_DIFF function.

5 Specify the location of the local Git repository that you want to retrieve a status record from.

6 Specify the older of the two commit IDs that you want to diff.

7 Specify the newer of the two commit IDs that you want to diff.

8 Specify the attribute that you want to retrieve from the status object.

9 Specify the variable that will store the returned attribute from the status object. The variable must be initialized prior to calling the GITFN_DIFF_GET function.

10 Use the PUT statement to display the returned attribute in the SAS log.

The preceding statements produce this result:

```
attribute-out= returned-attribute
```

See Also

"GIT_DIFF Function" on page 794

GITFN_DIFF_IDX_F Function

Returns the changes for a file in the index.

Category: Git

See: "Using Git Functions in SAS"

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_DIFF_IDX_F(*directory*, *file-name*, *diff-content*)

Required Arguments

directory
specifies the pathname of a local repository on the SAS server.

file-name
specifies the file name of the file in the index to return changes for.

diff-content
specifies a character variable to return the content of the diff to.

Details

Overview

The GITFN_DIFF_IDX_F function returns changes made to a file in the index against the last committed repository version of that file. The functions assignment variable indicates whether the operation was successful or not. Changes to the file are stored in the variable specified for the *diff-content* argument. The variable specified for the *diff_content* argument must be initialized prior to calling the GITFN_DIFF_IDX_F function.

Return Codes

The GITFN_DIFF_IDX_F function has return codes to indicate whether the function was successful.

Table 3.48 *Return Codes*

Return Code	Description
0	Indicates the function was successful.
–1	Indicates the function failed. See the log for additional details.

Example

```
data _null_;
LENGTH your-diff-content-variable $ 1024;        /* 1 */
your-diff-content-variable="";                   /* 2 */
 rc=GITFN_DIFF_IDX_F(
  "your-local-repo",                             /* 3 */
  "your-file-name",                              /* 4 */
  your-diff-content-variable);                   /* 5 */
PUT rc=;                                         /* 6 */
PUT your-diff-content-variable=;                 /* 7 */
run;
```

1 Set the length of your *diff-content* variable high enough to store all of the changes to the file.

2 Initialize the *diff_content* variable.

3 Specify the pathname of your local repository.

4 Specify the local pathname of the file that you want to diff.

5 Specify the variable that will store the content of the diff.

6 Use the PUT statement to write the SAS return code to the log.

7 Use the PUT statement to write the diff content to the log.

The content of your diff will be stored inside your diff-content-variable. Each diff is enclosed in braces {} seen in the example output below:

```
rc=0
 your-diff-content-variable={ your-diff-content }
```

GITFN_DIFF Function

Returns the number of diffs between two commits in the local repository and creates a diff record object for the local repository.

Category: Git

See: For more information see Chapter 1, "SAS Functions and CALL Routines".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_DIFF(*directory*, *older-commit-ID*, *newer-commit-ID*)

Required Arguments

directory
specifies the pathname of a cloned Git repository on the SAS server.

older-commit-ID
specifies the older of the two commit IDs that you want to diff.

newer-commit-ID
specifies the newer of the two commit IDs that you want to diff.

Details

Overview

The GITFN_DIFF function returns the number of changes between two commits and creates a diff record object in the local repository.

Return Codes

The GITFN_DIFF function has return codes to indicate if the function was successful.

Table 3.49 *Return Codes*

Return Code	Description
–1	The libgit2 library is unavailable and no Git operations can be used.

Example: Successfully Returning the Number of Changes between Two Commits

```
data _null_;
 n=gitfn_diff(
  "your-repository",          /* 1 */
  "older-commit-ID",          /* 2 */
  "newer-commit-ID");         /* 3 */
 put n=;                      /* 4 */
run;
```

1 Specify a local repository on the SAS server.

2 Specify an older commit ID in the local repository to diff.

3 Specify a newer commit ID in the local repository to diff.

4 Use the PUT statement to return the number of changes to the SAS log.

The following log shows that there is 1 change between the two commits, so the variable *n* returns 1.

```
n=1
```

See Also

"GIT_DIFF_GET Function" on page 795

GITFN_IDX_ADD Function

Stages 1 to *n* number of files to commit to the local repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_IDX_ADD(*directory*, *file-1*, *status-1* <, ... ,*file-n*, *status-n*>)

Required Arguments

directory
specifies the path of a cloned Git repository on the SAS server.

file
specifies the local pathname of a file to stage in the local repository.

status
specifies the status of a file to stage in the local repository.

Details

Overview

The GITFN_IDX_ADD function stages files in the local repository to be committed.

Return Values

The GITFN_IDX_ADD function does not have return codes. Instead, the success of the function can be determined by using the GITFN_STATUS function and inspecting the value of the staged attribute.

Example: Stage a File in the Local Repository

In this example, we first use the GITFN_STATUS function to find the local pathname of a file and the status. Next, we use the GITFN_IDX_ADD function to stage the file, and then we confirm that the file was staged by checking the value of the status attribute using the GITFN_STATUS function again.

```
data _null_;
 n = GITFN_STATUS("your-
repository");                                          /*1*/
 rc = GITFN_IDX_ADD("your-repository", "your-file", "file-
attribute");     /*2*/
 n = GITFN_STATUS("your-
repository");                                          /*3*/
run;
```

1. Use the GITFN_STATUS function to determine the file's local pathname and the file's status.
2. Stage the file using the GITFN_IDX_ADD function by specifying the local repository and the file name and status of the file that you want to stage.
3. Validate that the file was staged using the GITFN_STATUS function and confirming that the value of the staged attribute has changed from False to True.

The preceding statements produce these results:

```
NOTE: Entry: your-file. Staged: False. Status: your-file-status.
NOTE: Entry: your-file. Staged: True. Status: your-file-status.
```

See Also

"GIT_INDEX_REMOVE Function" on page 805

GITFN_IDX_REMOVE Function

Unstages 1 to *n* number of files to commit to the local repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_IDX_REMOVE(*directory*, *file-1* <, ..., *file-n*>)

Required Arguments

directory
specifies the path of a cloned Git repository on the SAS server.

file
specifies the local pathname of a file to stage in the local repository.

Details

Overview

The GITFN_IDX_REMOVE function unstages files in the local repository to be committed. You must specify at least one file, but multiple files can be unstaged by specifying additional file arguments.

Return Codes

The GITFN_IDX_REMOVE function does not have return codes. Instead, the success of the function can be determined using the GITFN_STATUS function by inspecting the value of the staged attribute.

Example: Unstage a File in the Local Repository

In this example, we first use the GITFN_STATUS function to find the local pathname of a file. Next, we use the GITFN_IDX_REMOVE function to unstage the file, and then we confirm that the file was unstaged by checking the value of the Status attribute using the GITFN_STATUS function again.

```
data _null_;
 n = GITFN_STATUS("your-repository");                    /* 1 */
 rc = GITFN_IDX_REMOVE("your-repository", "your-file");  /* 2 */
 n = GITFN_STATUS("your-repository");                    /* 3 */
run;
```

1 Use the GITFN_STATUS function to determine the local pathname of the file.

2 Unstage the file using the GITFN_IDX_REMOVE function by specifying the local repository and the file you want to unstage.

3 Validate that the file was unstaged using the GITFN_STATUS function and confirming that the value of the staged attribute has changed from True to False.

The preceding statements produce these results:

```
NOTE: Entry: your-file. Staged: True. Status: your-file-status.
NOTE: Entry: your-file. Staged: False. Status: your-file-status.
```

See Also

“GIT_INDEX_ADD Function” on page 803

GITFN_MRG_BRANCH Function

Merges a Git branch into the currently checked-out branch.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_MRG_BRANCH(*directory*, *branch-name*, *author-name*, *author-email*)

Required Arguments

directory
specifies the pathname of a local repository on the SAS server.

branch-name
specifies the branch that is merged into the currently checked-out branch.

author-name
specifies the author's name for the branch merge.

author-email
specifies the author's email for the branch merge.

Details

Overview

The GITFN_MRG_BRANCH function merges a specified Git branch into the currently checked-out Git branch.

Return Codes

The GITFN_MRG_BRANCH function has return codes that indicate whether the function was successful.

Table 3.50 Return Codes

Return Code	Description
–2	Indicates that the index has conflicts. Check the log for conflicting files.
–1	Indicates that the merge failed. Check the log for additional details.
0	Indicates that the merge was successful.
1	Indicates that the branch was already up-to-date.

Example

```
data _null_;
 rc = GITFN_MRG_BRANCH(
  "your-local-repository",          /* 1 */
  "your-target-branch",             /* 2 */
  "your-author-name",               /* 3 */
  "your-author-email");             /* 4 */
 put rc=;                           /* 5 */
run;                                /* 6 */
```

1 Specify the pathname of your local Git repository.

2 Specify the name of the branch that is merged into your currently checked-out branch. To check out a different branch, use the GITFN_CO_BRANCH function.

3 Specify the name of the author for the merge that appears in the commit history.

4 Specify the email of the author for the merge that appears in the commit history.

5 Use the PUT statement to output the SAS return code to the log.

The preceding statements produce these results:

```
NOTE: Merge 'your-target-branch' into current branch 'your-current-branch'
successful.
 rc=0
```

GITFN_NEW_BRANCH Function

Creates a Git branch.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_NEW_BRANCH(*directory*, *commit-ID*, *branch-name*, *force*)

Required Arguments

directory
specifies the pathname of a cloned Git repository on the SAS server.

commit-ID
specifies which commit level the new branch is cloned from.

branch-name
specifies a name for the new branch.

force
specifies whether to overwrite existing branches with the same name or not to overwrite existing branches.

Details

Overview

The GITFN_NEW_BRANCH function creates a branch in the local repository at a specified commit level. Branch names can contain only letters and numbers. The force argument allows you to specify whether you want to overwrite an existing branch with the same name. Valid values for force are listed below:

0
do not overwrite an existing branch.

1
overwrite an existing branch.

Return Codes

The GITFN_NEW_BRANCH function does not have return codes. A successful branch creation returns a SAS note to the log stating that the branch was successfully created.

Example

```
data _null_;
```

```
rc = gitfn_new_branch(
 "your-directory",          /* 1 */
 "your-commit-ID",          /* 2 */
 "your-branch-name",        /* 3 */
 your-force-value);         /* 4 */
run;
```

1 Specify your local Git repository.

2 Specify the commit ID to build the new branch from.

3 Specify a name for the new branch.

4 Specify whether to overwrite an existing branch with the same name.

The preceding statements produce this result:

```
NOTE: Branch "your-branch-name" successfully created.
```

GITFN_PULL Function

Pulls changes from the remote repository into the local repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

Form 1: **GITFN_PULL**(*directory*)

Form 2: **GITFN_PULL**(*directory*, *user*, *password*)

Form 3: **GITFN_PULL**(*directory*, *ssh-user*, *ssh-password*,*ssh-public-key*, *ssh-private-key*)

Required Argument

directory
specifies the local repository that you want to pull changes to from the remote repository.

Arguments for User Name-Password Authentication

user
specifies the Git user name credential for the remote repository.

password
specifies the Git password credential for the remote repository.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

Arguments for SSH Authentication

ssh-user
specifies the SSH user name for the remote repository.

ssh-password
specifies the password for your SSH keys.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

Note: If your keys are not password-protected, specify empty quotation marks ("" or '').

ssh-public-key
specifies the pathname for the public ssh key file.

ssh-private-key
specifies the pathname for the private ssh key file.

Details

Overview

The GITFN_PULL function pulls changes from the remote repository to the local repository. If authentication is required, you must specify either your Git account credentials or your SSH credentials.

Return Codes

The GITFN_PULL function has return codes that indicate whether the pull was successful.

Table 3.51 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git functions can be used.

Return Code	Description
0	Indicates that the pull was successful and the local repository was updated.
1	Indicates that the local repository is up-to-date and there are no new changes to pull from the remote repository.
>1	Indicates that the libgit2 library is available, but there was an error with the pull. See the SAS log for additional details.

Examples

Example 1: Pull to Local Git Repository Using User Name and Password Authentication

```
data _null_;
 rc= gitfn_pull(                    /* 1 */
  "your-local-repository",          /* 2 */
  "your-Git-username",              /* 3 */
  "your-Git-password");             /* 4 */
 put rc=;                           /* 5 */
run;
```

1 Assign the GITFN_PUSH function to a variable.

2 Specify the local repository to pull changes to from the remote repository.

3 Specify your Git user name for authentication to the remote repository.

4 Specify your Git password for authentication to the remote repository.

5 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
NOTE: Head has been fastforwarded to match the remote repository.
 rc=0
```

Example 2: Pull to Local Git Repository Using SSH Authentication

```
data _null_;
 rc= gitfn_pull(                    /* 1 */
  "your-local-repository",          /* 2 */
  "your-ssh-username",              /* 3 */
  "your-ssh-password",              /* 4 */
  "your-ssh-public-key",            /* 5 */
  "your-ssh-private-key");          /* 6 */
 put rc=;                           /* 7 */
run;
```

1. Assign the GITFN_PULL function to a variable.
2. Specify the local repository to pull changes to from the remote repository.
3. Specify your SSH user name for authentication to the remote repository. This is the same SSH user name that you used to clone the repository with GITFN_CLONE.
4. Specify the password for your SSH key. If your SSH keys are not password-protected, specify empty quotation marks ("" or '').
5. Specify the pathname of your public SSH key.
6. Specify the pathname of your private SSH key.
7. Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
NOTE: Head has been fastforwarded to match the remote repository.
 rc=0
```

See Also

- "GIT_COMMIT Function" on page 782
- "GIT_PUSH Function" on page 812

GITFN_PUSH Function

Pushes the committed files in the local repository to the remote repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

Form 1: **GITFN_PUSH**(*directory*)

Form 2: **GITFN_PUSH**(*directory*, *user*, *password*)

Form 3: **GITFN_PUSH**(*directory*, *ssh-user*, *ssh-password*, *ssh-public-key*, *ssh-private-key*)

Required Argument

directory
specifies the local repository that you want to push to the remote repository.

Arguments for User-Password Authentication

user
specifies your Git user name credential for the remote repository.

password
specifies your Git password credential for the remote repository.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

Arguments for SSH Authentication

ssh-user
specifies the SSH user name for the remote repository.

ssh-password
specifies the password for your SSH keys.

> **IMPORTANT** You can encode the password using PROC PWENCODE to hide your password in the SAS log. Passwords that are not encoded appear as plain text in the SAS log.

Note: If your keys are not password-protected, specify empty quotation marks ("" or '').

ssh-public-key
specifies the pathname for the public ssh key file.

ssh-private-key
specifies the pathname for the private ssh key file.

Details

Overview

The GITFN_PUSH function pushes changes from the local repository to the remote repository.

Return Codes

There are no return codes from SAS for the GITFN_PUSH function. However, a successful push returns a SAS note in the SAS log stating that the push was successful.

Examples

Example 1: Push to Remote Git Repository Using User Name and Password Authentication

```
data _null_;
 rc= gitfn_push(                    /*1*/
  "your-local-repository",          /*2*/
  "your-Git-username",              /*3*/
  "your-Git-password");             /*4*/
run;
```

1 Assign the GITFN_PUSH function to a variable.

2 Specify the local repository to push to the remote repository.

3 Specify your Git user name for authentication to the remote repository.

4 Specify your Git password for authentication to the remote repository.

The preceding statements produce these results:

```
NOTE: Push successful!
```

Example 2: Push to Remote Git Repository Using SSH Authentication

```
data _null_;
 rc= gitfn_push(                    /*1*/
  "your-local-repository",          /*2*/
  "your-ssh-user-name",             /*3*/
  "your-ssh-key-password",          /*4*/
  "your-public-ssh-key",            /*5*/
  "your-private-ssh-key");          /*6*/
run;
```

1 Assign the GITFN_PUSH function to a variable.

2 Specify the local repository to commit to the remote repository.

3 Specify your SSH user name for authentication to the remote repository. This is the same SSH user name that you used to clone the repository with GITFN_CLONE.

4 Specify the password for your SSH key. If your SSH keys are not password-protected, specify empty quotation marks ("" or '').

5 Specify the pathname of your public SSH key.

6 Specify the pathname of your private SSH key.

The preceding statements produce these results:

```
NOTE: Push successful!
```

See Also

- “GIT_COMMIT Function” on page 782
- “GIT_PULL Function” on page 808

GITFN_RESET_FILE Function

Resets a file in the index to the local repository version.

Category: Git

See: For more information see “Using Git Functions in SAS”.

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see “Deprecated Git Functions”.

Syntax

GITFN_RESET_FILE(*directory*, *file-name*)

Required Arguments

directory
specifies the pathname of a local repository on the SAS server.

file-name
specifies the local name of a file in the index to reset.

Details

Overview

The GITFN_RESET_FILE function resets a file in the index to the last commit version in the local repository.

Return Codes

The GITFN_RESET_FILE function has return codes that indicate whether the function was successful.

Table 3.52 *Return Codes*

Return Code	Description
0	Indicates that the file reset was successful.
–1	Indicates that the file reset failed. See the log for additional details.

Example: Reset a File in the Local Repository.

```
data _null_;
 rc = GITFN_RESET_FILE(
  "your-local-repository",      /* 1 */
  "your-file");                 /* 2 */
 put rc=;                       /* 3 */
run;
```

1 Specify your local repository on the SAS server.

2 Specify the file to reset in the local repository.

3 Use the PUT statement to output the SAS return code to the log.

Thee preceding statements produce these results:

```
NOTE: your-file reset successfully.
 rc=0
```

GITFN_RESET Function

Resets the local repository to a specified commit.

Category: Git

See: For more information see “Using Git Functions in SAS”.

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see “Deprecated Git Functions”.

Syntax

GITFN_RESET(*directory*, *commit-ID*, *reset_type*)

Required Arguments

directory
specifies the pathname of a cloned Git repository on the SAS server.

commit-ID
specifies the commit ID to reset the local repository to.

reset_type
specifies the type of reset.

Details

Overview

The GITFN_RESET function resets the local repository to a specified commit from the repository's commit history. The type of reset specified by the *reset_type* argument determines what happens to the local repository.

Soft
resets the current branch to the prior commit. No changes are made to the staged changes (the index) or your working directory.

Mixed
resets the current branch to the prior commit and resets your staged changes (the index). No changes are made to your working directory.

Hard
resets the current branch to the prior commit, resets your staged changes (the index), and discards all changes in your working directory. You cannot undo a hard reset.

Return Codes

The GITFN_RESET function does not have return codes. A successful reset returns a note to the log stating that the reset was successful.

Example: Reset the local repository to a specified commit ID

```
data _null_;
rc = GITFN_RESET(
  "your-local-repository",        /* 1 */
  "your-commit-ID",               /* 2 */
  "your-reset-type");             /* 3 */
```

```
run;
```

1 Specify your local Git repository.

2 Specify a commit ID to reset to.

3 Specify the type of reset to perform.

The preceding statements produce this result:

```
NOTE: your-reset-type reset to commit:your-commit-ID successful. 1
```

1 A successful reset returns a SAS note to the log stating the reset-type, the commit-ID, and a message stating the reset was successful.

GITFN_STATUS Function

Returns the status objects for files in the local repository and creates a status record.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_STATUS(*directory*)

Required Argument

directory
specifies the pathname of a cloned Git repository on the SAS server.

Details

Overview

The GITFN_STATUS function returns the number of status objects in your local repository and creates a status record. The function also writes a note to the SAS log for each status object in the local repository. The note returns the attributes below:

Entry
name of the file.

Staged
: Boolean operator that indicates whether the file is staged or unstaged.

Status
: state of the file. The status can be “New”, “Modified”, or “Deleted”.

Return Codes

The GITFN_STATUS function has return codes that indicate whether the status operation was successful.

Table 3.53 *Return Codes*

Return Code	Description
–1	The libgit2 library is unavailable and no Git operations can be used.
–2	The libgit2 library is available, but the status function failed. See the log for details.

Example: Local Repository Has One Status Object

```
data _null_;
    n = GITFN_STATUS("your-local-repository-directory");
    put n=;
run;
```

The following log shows that the local repository has one status object for the new file (*your-file-name*) that has not been staged. Because there is one object, the *n* variable equals 1.

```
NOTE: Entry: your-file-name. Staged: False. Status: New.
n=1
```

See Also

- “GIT_STATUS_FREE Function” on page 835
- “GIT_STATUS_GET Function” on page 838

GITFN_STATUS_GET Function

Returns the specified attribute of the *n*th status object returned from GITFN_STATUS() in the local repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_STATUS_GET(*n*, *directory*, *attribute*, *attribute-store*)

Required Arguments

n
specifies the *n*th status object to retrieve attributes from.

directory
specifies the path of a cloned Git repository on the SAS server.

attribute
specifies the attribute of the status object to return. Valid attributes are "PATH", "STATUS", or "STAGED". Only one attribute can be specified. If a non-valid attribute is specified, an error occurs.

attribute-store
specifies a character variable that stores the returned attribute.

Details

Overview

The GITFN_STATUS_GET function returns the specified attribute from the *n*th status object in a local Git repository.

Return Codes

The GITFN_STATUS_GET function has return codes that indicate whether the function was successful.

Table 3.54 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git functions can be used.

Return Code	Description
–2	Indicates that the libgit2 library is available, but no status objects were found in the local repository. The GITFN_STATUS function should be run first to create status objects for the local repository.
0	Indicates that the specified *n*th status object was found and the specified attribute was returned.
1	Indicates that the specified *n*th status object was found, but the specified attribute was invalid.
2	Indicates that the specified repository has a status record, but the specified *n*th status object does not exist.

Example: Return the PATH Attribute from a Status Object

In this example, we have previously successfully run the GITFN_STATUS function on the local repository to create a status record. Now we can use the GITFN_STATUS_GET function to return an attribute from a status object in our local repository.

```
data _null_;
 length attribute-store $ 1024;      /*1*/
 attribute-store="";                 /*2*/
 rc = GITFN_STATUS_GET(              /*3*/
        n,                           /*4*/
       "directory",                  /*5*/
       "attribute",                  /*6*/
        attribute-store);            /*7*/
 put varstore=;                      /*8*/
 put rc=;                            /*9*/
run;
```

1 Set a length for a storage variable that stores the returned attribute. Ensure that the length is equal to or greater than the attribute that is returned. This variable is used for the *attribute-store* argument.

2 Initialize the storage character variable.

3 Assign a variable to store the return code from SAS.

4 Specify the numeric integer number of the status object that you want to retrieve an attribute from. Status objects are listed in numeric order in the SAS log using the GITFN_STATUS function.

5 Specify the location of the local Git repository that you want to retrieve a status record from.

6 Specify the attribute that you want to retrieve from the status object. Valid attributes are "PATH", "STAGED", or "STATUS".

7 Specify the variable that will store the returned attribute from the status object. The variable must be initialized prior to calling the GITFN_STATUS_GET function.

8 Use the PUT statement to display the returned attribute in the SAS log.

9 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce these results:

```
varstore=your-returned-attribute
 rc=0
```

GITFN_STATUSFREE Function

Clears the status record object that was created by GITFN_STATUS for the specified repository.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_STATUSFREE (*directory*)

Required Argument

directory
specifies the pathname of a cloned Git repository on the SAS server.

Details

Overview

The GITFN_STATUSFREE function clears the status record that is associated with a specified repository.

Return Codes

The GITFN_STATUSFREE function has four possible return codes that indicate whether the function was successful.

Table 3.55 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git operations can be used.
–2	Indicates that the libgit2 library is available, but the function failed. See the log for details.
0	Indicates that the status record was successfully cleared.
1	Indicates that no status record was found for the specified repository. Therefore, no status record could be cleared.

Example: Successfully Clearing a Status Record

```
data _null_;
 rc = GITFN_STATUSFREE("your-repository");      /*1*/
 put rc=;                                       /*2*/
run;
```

1 Specify a target local repository on the SAS server to clear a status record from.

2 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce this result:

```
rc=0
```

See Also

- “GIT_STATUS Function” on page 837
- “GIT_STATUS_GET Function” on page 838

GITFN_VERSION Function

Specifies whether libgit2 is available and if available, specifies the version that is being used.

Category: Git

See: For more information see "Using Git Functions in SAS".

The GITFN_ functions were deprecated and replaced with the GIT_ functions. For a list of the new GIT_ functions and the deprecated functions that they replace, see "Deprecated Git Functions".

Syntax

GITFN_VERSION()

Without Arguments

The GITFN_VERSION function has no arguments. If you try to pass arguments, an error occurs.

Details

Return Values

The GITFN_VERSION function has two possible return values:

Table 3.56 *Return Codes*

Return Code	Description
–1	Indicates that the libgit2 library is unavailable and no Git functions can be used.
version	Indicates the version of the libgit2 library that is being used.

Example

```
data _null_;
   version = GITFN_VERSION();       /*1*/
   put version=;                    /*2*/
run;
```

1 Assign a variable to store the return code from SAS.

2 Use the PUT statement to display the SAS return code in the SAS log.

The preceding statements produce this result:

```
version=0.27
```

GRAYCODE Function

Generates all subsets of *n* items in a minimal change order.

Category: Combinatorial

Restriction: The GRAYCODE function cannot be executed when you use the %SYSFUNC macro.

Syntax

GRAYCODE(*k*, *numeric-variable-1*, ..., *numeric-variable-n*)

GRAYCODE(*k*, *character-variable* <, *n* <, *in-out*>>)

Required Arguments

k

specifies a numeric variable. Initialize *k* to either of the following values before executing the GRAYCODE function:

- a negative number to cause GRAYCODE to initialize the subset to be empty
- the number of items in the initial set indicated by *numeric-variable-1* through *numeric-variable-n*, or *character-variable*, which must be an integer value between 0 and *n* inclusive

The value of *k* is updated when GRAYCODE is executed. The value that is returned is the number of items in the subset.

numeric-variable

specifies numeric variables that have values of 0 or 1 which are updated when GRAYCODE is executed. A value of 1 for *numeric-variable-j* indicates that the jth item is in the subset. A value of 0 for *numeric-variable-j* indicates that the jth item is not in the subset.

If you assign a negative value to *k* before you execute GRAYCODE, then you do not need to initialize *numeric-variable-1* through *numeric-variable-n* before executing GRAYCODE unless you want to suppress the note about uninitialized variables.

If you assign a value between 0 and *n* inclusive to *k* before you execute GRAYCODE, then you must initialize *numeric-variable-1* through *numeric-variable-n* to *k* values of 1 and *n-k* values of 0.

character-variable

specifies a character variable that has a length of at least *n* characters. The first *n* characters indicate which items are in the subset. By default, an "I" in the jth position indicates that the jth item is in the subset, and an "O" in the jth position indicates that the jth item is out of the subset. You can change the two characters by specifying the *in-out* argument.

If you assign a negative value to *k* before you execute GRAYCODE, then you do not need to initialize *character-variable* before executing GRAYCODE unless you want to suppress the note about an uninitialized variable.

If you assign a value between 0 and *n* inclusive to *k* before you execute GRAYCODE, then you must initialize *character-variable* to *k* characters that indicate an item is in the subset, and *n*-*k* characters that indicate an item is out of the subset.

Optional Arguments

n
specifies a numeric constant, variable, or expression. By default, *n* is the length of *character-variable*.

in-out
specifies a character constant, variable, or expression. The default value is "IO." The first character is used to indicate that an item is in the subset. The second character is used to indicate that an item is out of the subset.

Details

When you execute GRAYCODE with a negative value of *k*, the subset is initialized to be empty. The GRAYCODE function returns zero.

When you execute GRAYCODE with an integer value of *k* between 0 and *n* inclusive, one item is either added to the subset or removed from the subset, and the value of *k* is updated to equal the number of items in the subset. If the jth item is added to the subset or removed from the subset, the GRAYCODE function returns *j*.

To generate all subsets of *n* items, you can initialize *k* to a negative value and execute GRAYCODE in a loop that iterates 2**n* times. If you want to start with a non-empty subset, then initialize *k* to be the number of items in the subset, initialize the other arguments to specify the desired initial subset, and execute GRAYCODE in a loop that iterates 2**n*-1 times. The sequence of subsets that are generated by GRAYCODE is cyclical, so you can begin with any subset that you want.

Examples

Example 1: Using n=4 Numeric Variables and Negative Initial k

The following program uses numeric variables to generate subsets in a minimal change order.

```
data _null_;
   array x[4];
   n=dim(x);
   k=-1;
   nsubs=2**n;
```

```
   do i=1 to nsubs;
      rc=graycode(k, of x[*]);
      put i 5. +3 k= ' x=' x[*] +3 rc=;
   end;
run;
```

SAS writes the following output to the log:

```
 1   k=0  x=0 0 0 0    rc=0
 2   k=1  x=1 0 0 0    rc=1
 3   k=2  x=1 1 0 0    rc=2
 4   k=1  x=0 1 0 0    rc=1
 5   k=2  x=0 1 1 0    rc=3
 6   k=3  x=1 1 1 0    rc=1
 7   k=2  x=1 0 1 0    rc=2
 8   k=1  x=0 0 1 0    rc=1
 9   k=2  x=0 0 1 1    rc=4
10   k=3  x=1 0 1 1    rc=1
11   k=4  x=1 1 1 1    rc=2
12   k=3  x=0 1 1 1    rc=1
13   k=2  x=0 1 0 1    rc=3
14   k=3  x=1 1 0 1    rc=1
15   k=2  x=1 0 0 1    rc=2
16   k=1  x=0 0 0 1    rc=1
```

Example 2: Using a Character Variable and Positive Initial k

The following example uses a character variable to generate subsets in a minimal change order.

```
data _null_;
   x='++++';
   n=length(x);
   k=countc(x, '+');
   put '    1' +3 k= +2 x=;
   nsubs=2**n;
   do i=2 to nsubs;
      rc=graycode(k, x, n, '+-');
      put i 5. +3 k= +2 x= +3 rc=;
   end;
run;
```

SAS writes the following output to the log:

```
 1   k=4   x=++++
 2   k=3   x=-+++    rc=1
 3   k=2   x=-+-+    rc=3
 4   k=3   x=++-+    rc=1
 5   k=2   x=+--+    rc=2
 6   k=1   x=---+    rc=1
 7   k=0   x=----    rc=4
 8   k=1   x=+---    rc=1
 9   k=2   x=++--    rc=2
10   k=1   x=-+--    rc=1
11   k=2   x=-++-    rc=3
12   k=3   x=+++-    rc=1
13   k=2   x=+-+-    rc=2
14   k=1   x=--+-    rc=1
15   k=2   x=--++    rc=4
16   k=3   x=+-++    rc=1
```

See Also

CALL Routines:

- "CALL GRAYCODE Routine" on page 236

HARMEAN Function

Returns the harmonic mean.

Category: Descriptive Statistics

Syntax

HARMEAN(*argument* <, *argument*, ..., *argument-n*>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the harmonic mean of the 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 nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

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 nonmissing values. Unlike HARMEANZ, HARMEAN fuzzes the values of the arguments that are approximately zero.

Example

```
data
one;

   x1=harmean(1, 2, 4,
4);

   x2=harmean(., 4, 12,
24);

   x3=harmean(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=2 x2=8 x3=3.2
```

See Also

Functions:

- “GEOMEAN Function” on page 758
- “GEOMEANZ Function” on page 759
- “HARMEANZ Function” on page 891
- “MEAN Function” on page 1103

HARMEANZ Function

Returns the harmonic mean, using zero fuzzing.

Category: Descriptive Statistics

Syntax

HARMEANZ(*argument* <, *argument*, ..., *argument-n*>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the harmonic mean of the 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 nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean. The HARMEANZ function returns the harmonic mean of the nonmissing values. Unlike HARMEAN, HARMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

```
data
one;

   x1=harmeanz(1, 2, 4,
4);

   x2=harmeanz(., 4, 12,
24);

   x3=harmeanz(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=2 x2=8 x3=3.2
```

See Also

Functions:

- “GEOMEAN Function” on page 758
- “GEOMEANZ Function” on page 759
- “HARMEAN Function” on page 889
- “MEAN Function” on page 1103

HASH_FAST_HEX Function

Returns a message digest as a 16-byte hexadecimal string for a key value and a message consisting of one or more character strings or numbers, using the fast-hash algorithm.

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see “Internationalization Compatibility for SAS String Functions” in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

See: “Hashing Functions and Hash-Based Message Authentication Code” for information about hashing functions.

Syntax

HASH_FAST_HEX(key,<arg1..., argn>)

Required Argument

key
specifies an integer value from 0 to 2**64-1 called a key, seed, salt, or secret.

Optional Argument

arg1, ..., argn
specifies a character or numeric constant, variable, or expression.

> **TIP** Enclose a literal string of characters in quotation marks.

Details

See Why fast hash? for information about the fast-hash algorithm.

Operating Environment Information: Results from the HASH_FAST_HEX function depend on the character encoding of the message. For example, 'abc' is '616263'x in ASCII but '818283'x in EBCDIC, which results in different digests.

Example

An application of HASH_FAST is generating a variable to use as a row identifier. If all the rows in a large data set are distinct, then all the values of HASH_FAST(&seed, _all_) are likely to be distinct. Here is an example that uses a small data set:

```
data abc;
   input x $ y $ z $;
datalines;
abc . .
ab c .
a bc .
a b c
;

data digests;
   set abc;
   fast = hash_fast_hex( 12345, trimn(x), trimn(y), trimn(z) );
run;

proc print data=digests;
run;
```

HASH_XX_HEX Function

Returns a message digest as a 16-byte hexadecimal string for a key value and a message consisting of one or more character strings or numbers.

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see "Internationalization Compatibility for SAS String Functions" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

See: "Hashing Functions and Hash-Based Message Authentication Code" for information about hashing functions.

Syntax

HASH_XX_HEX(key,<arg1..., argn>)

Required Argument

key
specifies an integer value from 0 to 2**64-1 called a key, seed, salt, or secret.

Optional Argument

arg1, ..., argn
specifies a character or numeric constant, variable, or expression.

> **TIP** Enclose a literal string of characters in quotation marks.

Details

See xxHash - Extremely fast hash algorithm for information about the XXHASH algorithm.

Operating Environment Information: Results from the HASH_FAST_HEX function depend on the character encoding of the message. For example, 'abc' is '616263'x in ASCII but '818283'x in EBCDIC, which results in different digests.

Example

An application of HASH_XX_HEX is generating a variable to use as a row identifier. If all the rows in a large data set are distinct, then all the values of HASH_XX_HEX(&seed, _all_) are likely to be distinct. Here is an example that uses a small data set:

```
data abc;
   input x $ y $ z $;
datalines;
abc . .
ab c .
a bc .
a b c
;

data digests;
   set abc;
   xx = hash_xx_hex( 12345, trimn(x), trimn(y), trimn(z) );
run;

proc print data=digests;
run;
```

HASHING Function

Returns a message digest as a hexadecimal string for a message consisting of a character string, using different methods.

Category: Character

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

HASHING(*method*, *message,*<*flag*>);

Required Arguments

method
specifies one of the following hashing methods as a character constant, variable, or expression in uppercase, lowercase, or mixed case:

- MD5
- SHA1
- SHA256
- SHA384

- SHA512
- CRC32

message
specifies a character constant, variable, or expression.

Note: If the message is a character constant and contains invalid characters, write the message as a hexadecimal constant.

Optional Argument

flag
specifies whether the *message* is in hexadecimal representation.

0 has no effect.

1 specifies that the expression in the argument *message* is in hexadecimal representation.

Details

If *method* is invalid, the returned digest is blank, and a note, warning, or error message is issued stating that the argument is invalid.

Example

This example generates data from the HASHING function:

```
data null;
   message = "The quick brown fox jumps over the lazy dog";
   md5 = hashing('md5',message);
   sha1 = hashing('sha1',message);
   sha256 = hashing('sha256',message);
   put md5= / sha1= / sha256=;
 run;
```

These statements produce these results:

```
md5=9E107D9D372BB6826BD81D3542A419D6
sha1=2FD4E1C67A2D28FCED849EE1BB76E7391B93EB12
sha256=D7A8FBB307D7809469CA9ABCB0082E4F8D5651E46D3CDB762D02D0BF37C9E592
```

HASHING_FILE Function

Returns a message digest as a hexadecimal string for a message consisting of an entire file, using different hashing methods.

Category: Character

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for more information about hashing functions and HMAC.

Syntax

HASHING_FILE(*method*, *file,* <*flag*>);

Required Arguments

method
specifies one of the following hashing methods as a character constant, variable, or expression in uppercase, lowercase, or mixed case:

- MD5
- SH1
- SHA256
- SHA384
- SHA512
- CRC32

file
specifies a file name, fileref, or file path as a character constant, variable, or expression.

Optional Argument

flag
specifies an integer constant, variable, or expression with one of the following values:

0 has no effect.

1 specifies that the contents of the file are in hexadecimal representation.

4 specifies that the *file* argument is a fileref, not a file name or a file path.

5 specifies values 1 and 4.

Details

If the character string in the *file* argument is invalid, the return digest is blank.

Example

This example generates results for the HASHING_FILE function:

```
filename abc temp recfm=f lrecl=5;
data _null_;
   file abc;
   put '3334353637'x    /*34567 in ASCII*/;
run;

data _null_;
   x = hashing_file('sha256','abc',4);
   put x=;
run;
```

These statements produce this result:

```
x=831D606C1FF4CD3B74522885194FC0DED5F5BE1E043A6A46E59B08896F452657
```

HASHING_HMAC Function

Returns a message digest as a hexadecimal string for a character HMAC key value and a message consisting of a character string, using various methods.

Category: Character

Note: For more information, see Hash-based message authentication code (HMAC).

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

HASHING_HMAC(*method*, *key*, *message*,<*flag*>);

Required Arguments

method
specifies one of the following hashing methods as a character constant, variable, or expression in uppercase, lowercase, or mixed case:

- MD5

- SH1
- SHA256
- SHA384
- SHA512
- CRC32

key
specifies a secret key as a character constant, variable, or expression in uppercase, lowercase, or mixed case.

message
specifies a character constant, variable, or expression.

Note: If the message is a character constant and contains invalid characters, write the message as a hexadecimal constant.

Optional Argument

flag
specifies whether the *message* or *key* is in hexadecimal.

0	has no effect.
1	specifies that the expression in the argument *message* is in hexadecimal.
2	specifies that *key* is in hexadecimal.
3	specifies that *message* and *key* are in hexadecimal.

Details

If *method* is invalid, the returned digest is blank, and a note, warning, or error message is issued stating that the argument is invalid.

If the number of bytes in the key is less than the block size of the hash function, the *key* is padded on the right with extra zero bytes up to the block size.

Example

This example generates results from the HASHING_HMAC function:

```
data null;
   message = "The quick brown fox jumps over the lazy dog";
   md5 = hashing_hmac('md5','key',message);
   sha1 = hashing_hmac('sha1','key',message);
   sha256 = hashing_hmac('sha256','key',message);
   put md5= / sha1= / sha256=;
 run;
```

These statements produce these results:

```
md5=80070713463E7749B90C2DC24911E275
sha1=DE7C9B85B8B78AA6BC8A7A36F70A90701C9DB4D9
sha256=F7BC83F430538424B13298E6AA6FB143EF4D59A14946175997479DBC2D1A3CD8
```

HASHING_HMAC_FILE Function

Returns a message digest as a hexadecimal string for an HMAC key value consisting of a character string and a message consisting of an entire file, using different methods.

Category: Character

Note: For more information, see Hash-based message authentication code (HMAC).

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

HASHING_HMAC_FILE(*method*, *key*, *file,* <*flag*>);

Required Arguments

method
specifies one of the following hashing methods as a character constant, variable, or expression in uppercase, lowercase, or mixed case:

- MD5
- SH1
- SHA256
- SHA384
- SHA512
- CRC32

key
specifies a secret key as a character constant, variable, or expression.

file
specifies a file name, fileref, or file path as a character constant, variable, or expression.

Optional Argument

flag
specifies an integer constant, variable, or expression with one of the following values:

0 has no effect.

1 specifies that the contents of the file are in hexadecimal values.

2 specifies that the *key* is in hexadecimal.

3 specifies that the *file* contents and *key* are in hexadecimal.

4 specifies that the *file* argument is a fileref, not a file name or a file path.

5 specifies values 1 and 4.

Details

If *method* is invalid, the returned digest is blank, and a note, warning, or error message is issued stating that the argument is invalid.

If the number of bytes in the *key* is less than the block size of the hash function, *key* is padded on the right with extra zero bytes up to the block size.

Example

This example generates results for the HASHING_HMAC_FILE function:

```
filename abc temp recfm=f lrecl=5;
data _null_;
   file abc;
   put '3334353637'x    /*34567 in ASCII*/;
run;

data _null_;
   x = hashing_hmac_file('sha256','313233','abc' ,4);
   put x=;
run;
```

These statements produce this result:

```
x=4B74AD28DBB55FD0A06D5E94A858C3EAD0A5679FE318A5DF6865079401DE1C61
```

HASHING_HMAC_INIT Function

Initializes a running HMAC hash and returns a numeric handle for use with HASHING_PART and HASHING_TERM.

Category: Character

Note: For more information, see Hash-based message authentication code (HMAC).

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

handle=**HASHING_HMAC_INIT**(*method*, *key,* <*flag*>);

Required Arguments

method
specifies one of the following hashing methods as a character constant, variable, or expression in uppercase, lowercase, or mixed case:

- MD5
- SHA1
- SHA256
- SHA384
- SHA512
- CRC32

key
specifies a secret key as a character constant, variable, or expression.

flag
specifies if the *key* is in hexadecimal values.

0 has no effect.

2 specifies that *key* is in hexadecimal (HMAC usage).

Details

HASHING_HMAC_INIT returns a handle as a positive integer if *method* is recognized. If *method* is invalid, the returned handle has a numeric missing value, and a note, warning, or error message is issued stating that the argument is invalid.

The handle can be passed to HASHING_PART and HASHING_TERM to compute a running hash. The handle cannot be passed from one DATA step to the next or saved for later use.

Example

This example generates results that use the HASHING_TERM, HASHING_PART, and HASHING_HMAC_INIT functions:

```
data _null_;
```

```
message = "The quick brown fox jumps over the lazy dog";
l = length(message);
md5h = hashing_hmac_init('md5','key');
sha1h = hashing_hmac_init('sha1','key');
sha256h = hashing_hmac_init('sha256','key');
array handles md5h sha1h sha256h;
do i=1 to l;
   do over handles;
   rc = hashing_part(handles,substr(message,i,1));
   end;
end;
md5 = hashing_term(md5h);
sha1 = hashing_term(sha1h);
sha256 = hashing_term(sha256h);
put md5= / sha1= / sha256=;
run;
```

These statements produce these results:

```
md5=80070713463E7749B90C2DC24911E275
sha1=DE7C9B85B8B78AA6BC8A7A36F70A90701C9DB4D9
sha256=F7BC83F430538424B13298E6AA6FB143EF4D59A14946175997479DBC2D1A3CD8
```

HASHING_INIT Function

Initializes a running hash and returns a numeric handle for use with HASHING_PART and HASHING_TERM.

Category: Character

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

handle=**HASHING_INIT**(*method*);

Required Argument

method
specifies one of the following hashing methods as a character constant, variable, or expression in uppercase, lowercase, or mixed case:

- MD5
- SHA1
- SHA256
- SHA384

- SHA512
- CRC32

Details

HASHING_INIT returns handle as a positive integer if *method* is recognized. If *method* is invalid, the returned handle has a numeric missing value, and a note, warning, or error message is issued stating that the argument is invalid.

The handle can be passed to HASHING_PART and HASHING_TERM to compute a running hash. The handle cannot be passed from one DATA step to the next or saved for later use.

Example

This example generates results from the HASHING_INIT function:

```
data test;
   md5h = hashing_init('md5');
   sha1h = hashing_init('sha1');
   sha256h = hashing_init('sha256');
   put md5h=;
   put sha1h=;
   put sha256h=;
run;
```

These statements produce these results:

```
md5h=1
sha1h=2
sha256h=3
```

HASHING_PART Function

Provides part of a message for a running hash and returns a numeric value of 1 for a valid handle.

Category: Character

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

HASHING_PART(*handle*, *message_part,* <*flag*>);

Required Arguments

handle
specifies the handle that is returned by HASHING_INIT or HASHING_HMAC_INIT.

If the handle is invalid, HASHING_PART returns a numeric missing value, an error message is issued, and the program step might terminate depending on the calling environment.

message_part
specifies a character constant, variable, or expression.

Optional Argument

flag
specifies if the *message_part* is in hexadecimal values.

0 has no effect.

1 specifies that the expression in the argument *message_part* is in hexadecimal.

Details

The HASHING_PART function computes a running hash from the character string in *message_part*. HASHING_PART can be called multiple times in support of piecemeal data.

You must get a handle from HASHING_INIT or HASHING_HMAC_INIT before calling HASHING_PART. After calling HASHING_PART, you can get the final digest by calling HASHING_TERM.

Example

This example generates results that use the HASHING_TERM, HASHING_PART, and HASHING_HMAC_INIT functions:

```
data _null_;
   message = "The quick brown fox jumps over the lazy dog";
   l = length(message);
   md5h = hashing_hmac_init('md5','key');
   sha1h = hashing_hmac_init('sha1','key');
   sha256h = hashing_hmac_init('sha256','key');
   array handles md5h sha1h sha256h;
   do i=1 to l;
      do over handles;
      rc = hashing_part(handles,substr(message,i,1));
      end;
   end;
   md5 = hashing_term(md5h);
```

```
  sha1 = hashing_term(sha1h);
  sha256 = hashing_term(sha256h);
  put md5= / sha1= / sha256=;
run;
```

These statements produce these results:

```
md5=80070713463E7749B90C2DC24911E275
sha1=DE7C9B85B8B78AA6BC8A7A36F70A90701C9DB4D9
sha256=F7BC83F430538424B13298E6AA6FB143EF4D59A14946175997479DBC2D1A3CD8
```

HASHING_TERM Function

Returns the final digest in hexadecimal representation for the running hash.

Category: Character

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

HASHING_TERM(*handle*);

Required Argument

handle
specifies the handle that is returned by HASHING_INIT or HASHING_HMAC_INIT. If the handle is invalid, HASHING_TERM returns a numeric missing value, an error message is issued, and the program step might terminate depending on the calling environment.

Details

The HASHING_TERM function computes the final digest from the running hash. The return digest is in hexadecimal representation.

You must get a handle from HASHING_INIT or HASHING_HMAC_INIT before calling HASHING_TERM, and you must call HASHING_PART. If the final digest cannot be computed, the result is blank, an error message is issued, and the program step might terminate depending on the calling environment.

After calling HASHING_TERM, the handle cannot be used again without first calling HASHING_INIT or HASHING_HMAC_INIT.

Example

This example generates results that use the HASHING_TERM, HASHING_PART, and HASHING_HMAC_INIT functions:

```
data _null_;
   message = "The quick brown fox jumps over the lazy dog";
   l = length(message);
   md5h = hashing_hmac_init('md5','key');
   sha1h = hashing_hmac_init('sha1','key');
   sha256h = hashing_hmac_init('sha256','key');
   array handles md5h sha1h sha256h;
   do i=1 to l;
      do over handles;
      rc = hashing_part(handles,substr(message,i,1));
      end;
   end;
   md5 = hashing_term(md5h);
   sha1 = hashing_term(sha1h);
   sha256 = hashing_term(sha256h);
   put md5= / sha1= / sha256=;
 run;
```

These statements produce these results:

```
md5=80070713463E7749B90C2DC24911E275
sha1=DE7C9B85B8B78AA6BC8A7A36F70A90701C9DB4D9
sha256=F7BC83F430538424B13298E6AA6FB143EF4D59A14946175997479DBC2D1A3CD8
```

HBOUND Function

Returns the upper bound of an array.

Category: Array

Syntax

HBOUND<*n*> (*array-name*)

HBOUND(*array-name*, *bound-n*)

Required Arguments

array-name
is the name of an array that was defined previously in the same DATA step.

bound-n
is a numeric constant, variable, or expression that specifies the dimension for which you want to know the upper bound. Use *bound-n* only if *n* is not specified.

Optional Argument

n
is an integer constant that specifies the dimension for which you want to know the upper bound. If no *n* value is specified, the HBOUND function returns the upper bound of the first dimension of the array.

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. Use HBOUND in array processing to avoid changing the upper bound of an iterative DO group each time you change the bounds of the array. HBOUND and LBOUND can be used together to return the values of the upper and lower bounds of an array dimension.

Comparisons

- HBOUND returns the literal value of the upper bound of an array dimension.
- DIM always returns a total count of the number of elements in an array dimension.

Note: This distinction is important when the lower bound of an array dimension has a value other than 1 and the upper bound has a value other than the total number of elements in the array dimension.

Examples

Example 1: One-Dimensional Array

In this example, HBOUND returns the upper bound of the dimension, a value of 5. Therefore, SAS repeats the statements in the DO loop five times.

```
data
one;

   array big{5} weight sex height state
city;

     do i=1 to
hbound(big);

   put
i=;
```

```
end;

run;
```

These statements produce these results:

```
i=1
i=2
i=3
i=4
i=5
```

Example 2: Multidimensional Array

This example shows two ways of specifying the HBOUND function for multidimensional arrays.

```
data
one;

   array mult{2:6,4:13,2} mult1-
mult100;


val1a=HBOUND(MULT);


val1b=HBOUND(MULT,1);




val2a=HBOUND2(MULT);


val2b=HBOUND(MULT,2);




val3a=HBOUND3(MULT);


val3b=HBOUND(MULT,3);

run;



proc
print;

run;
```

See Also

Functions:

- "DIM Function" on page 545
- "LBOUND Function" on page 1044

Statements:

- "ARRAY Statement" in *SAS DATA Step Statements for SAS Viya Workbench*
- "Array Reference Statement" in *SAS DATA Step Statements for SAS Viya Workbench*

Other References:

- "Using Arrays" in *SAS Language Programmer's Guide for SAS Viya Workbench*

HMS Function

Returns a SAS time value from hour, minute, and second values.

Category: Date and Time

Syntax

HMS(*hour*, *minute*, *second*)

Required Arguments

hour
is numeric.

minute
is numeric.

second
is numeric.

Details

The HMS function returns a positive numeric value that represents a SAS time value.

Example

```
data _null_;
   hrid=hms(12, 45, 10);
   put hrid / hrid time.;
run;
```

These statements produce these results:

```
45910
12:45:10
```

See Also

Functions:

- “DHMS Function” on page 540
- “HOUR Function” on page 934
- “MINUTE Function” on page 1107
- “SECOND Function” on page 1381

HOLIDAY Function

Returns a SAS date value of a specified holiday for a specified year.

Category: Date and Time

Syntax

HOLIDAY('*holiday*', *year*)

Required 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 3.57 Holiday Values and Their Descriptions

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canada Day	July 1
CANADAOBSERVED	Canada 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
JUNETEENTH	Juneteenth	June 19
JUNETEENTHUSG	Juneteenth - U.S. government-observed	Juneteenth date observed by U.S. government for Monday–Friday schedule
JUNETEENTHUSPS	Juneteenth - U.S. post office -observed	Juneteenth date observed by U.S. government for Monday–Saturday schedule (U.S. Postal Service)
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)
MOTHERS	Mother's Day	2nd Sunday in May
NEWYEAR	New Year's Day	January 1

Holiday Value	Description	Date Celebrated
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

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.

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 Table 3.65 on page 912 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', 2021);` returns the same value as `NWKDOM(4, 5, 11, 2021);`.

In other cases, the HOLIDAY function and the MDY function return the same result. For example, the statement `HOLIDAY('CHRISTMAS', 2021);` returns the same value as `MDY(12, 25, 2021);`.

Example

```
data
one;

   thanks = holiday('thanksgiving',
2021);

   format thanks
date9.;

   put
thanks=;



   boxing = holiday('boxing',
2021);

   format boxing
date9.;

   put
boxing=;



   easter = holiday('easter',
2021);

   format easter
date9.;
```

```
   put
easter=;


   canada = holiday('canada',
2021);

   format canada
date9.;

   put
canada=;


   fathers = holiday('fathers',
2021);

   format fathers
date9.;

   put
fathers=;


   valentines = holiday('valentines',
2021);

   format valentines
date9.;

   put
valentines=;


   victoria = holiday('victoria',
2021);

   format victoria
date9.;

   put
victoria=;

run;
```

SAS writes these results to the log:

```
thanks=25NOV2021
boxing=26DEC2021
easter=04APR2021
canada=01JUL2021
fathers=20JUN2021
valentines=14FEB2021
victoria=24MAY2021
```

See Also

Functions:

- “MDY Function” on page 1102
- “NWKDOM Function” on page 1180

HOLIDAYCK Function

Returns the number of occurrences of the holiday value between date1 and date2.

Category: Date and Time

Syntax

HOLIDAYCK('*holiday*',*date1*, *date2* <,'*locale*'>)

Required Arguments

'holiday'
is a character constant, variable, or expression that specifies one of the values that are listed in the following table. The *holiday* variable can also be defined using the DATEKEYS procedure and is made available to the function by using the EVENTDS= system option.

Table 3.58 *Holiday Values and Their Descriptions*

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canada Day	July 1
CANADAOBSERVED	Canada Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25

Holiday Value	Description	Date Celebrated
COLUMBUS	Columbus Day	2nd Monday in October
JUNETEENTH	Juneteenth	June 19
JUNETEENTHUSG	Juneteenth - U.S. government-observed	Juneteenth date observed by U.S. government for Monday–Friday schedule
JUNETEENTHUSPS	Juneteenth - U.S. post office -observed	Juneteenth date observed by U.S. government for Monday–Saturday schedule (U.S. Postal Service)
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)
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

Holiday Value	Description	Date Celebrated
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

date1
specifies the beginning date value in the form '*ddmonyy*'d or '*ddmonyyyy*'d.

dd	is a two-digit integer that represents the day of the month.
mon	is a three-character string that represents the month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.
yy or *yyyy*	is a two- or four-digit integer that represents the year. If only two digits are used, the YEARCUTOFF= option is valid.

date2
specifies the end date value in the form '*ddmonyy*'d or '*ddmonyyyy*'d.

dd	is a two-digit integer that represents the day of the month.
mon	is a three-character string that represents the month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.
yy or *yyyy*	is a two- or four-digit integer that represents the year. If only two digits are used, the YEARCUTOFF= option is valid.

Optional Argument

'locale'
specifies the POSIX value. See "Locale-to-Encoding Mapping" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Details

The EVENTDS= option makes the date keys available. You can define the date keys with the DATEKEYS procedure. Also, you can assign a locale to a date key with the DATEKEYS procedure. If you do not assign the locale to the date keys, then the predefined date keys do not have locales. If the locale argument is specified, then the date key must be defined using the DATEKEYS procedure, or a predefined date key needs to have a locale set that uses the DATEKEYS procedure. Then the new definition needs to be made available with the EVENTDS= system option.

Example

The following example specifies the Easter holiday when it occurs in March and April:

```
data;
   do year = 2001 to 2021;
      EasterInMarch=HOLIDAYCK('EASTER',MDY(3,1,year),mdy(3,31,year));
      EasterInApril=HOLIDAYCK('EASTER',MDY(4,1,year),mdy(4,30,year));
      output;
   end;
run;
```

These statements produce these results:

Figure 3.2 *Easter Holiday in March and April*

year	EasterInMarch	EasterInApril
2001	0	1
2002	1	0
2003	0	1
2004	0	1
2005	1	0
2006	0	1
2007	0	1
2008	1	0
2009	0	1
2010	0	1
2011	0	1
2012	0	1
2013	1	0
2014	0	1
2015	0	1
2016	1	0
2017	0	1
2018	0	1
2019	0	1
2020	0	1
2021	0	1

HOLIDAYCOUNT Function

Returns the number of holidays defined for a SAS date value.

Category: Date and Time

Syntax

HOLIDAYCOUNT(*date*<,'*locale*'>)

Required Argument

date
specifies a SAS date value '*ddmonyy*'d or '*ddmonyyyy*'d.

dd is a two-digit integer that represents the day of the month.

mon is a three-character string that represents the month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.

yy or *yyyy* is a two- or four-digit integer that represents the year. If only two digits are used, the YEARCUTOFF= option is valid.

Optional Argument

'locale'
specifies the POSIX locale value. See "Locale-to-Encoding Mapping" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Details

The EVENTDS= option makes the date keys available. You can define the date keys with the DATEKEYS procedure. Also, you assign a locale to a date key with the DATEKEYS procedure. If you do not assign the locale to the date keys, then the predefined date keys do not have locales. If the locale argument is specified, then the date key must be defined using the DATEKEYS procedure, or a predefined date key needs to have a locale set that uses the DATEKEYS procedure. Then the new definition needs to be made available with the EVENTDS= system option.

Example

The following example specifies the number of holidays:

```
data a;
   length Holiday $32;
   format date weekdatx17.;
   do year=2015 to 2021;
      date=mdy(11,11,year);
      ObsVetNov11=holidaycount(date);
      do index=1 to ObsVetNov11;
         Holiday=holidayname(date,index);
         output;
      end;
   end;
run;
proc print data=a;
run;
```

These statements produce these results:

Output 3.36 *The Number of Veterans Day Holidays*

The SAS System

Obs	Holiday	date	year	ObsVetNov11	index
1	VETERANS	Wed, 11 Nov 2015	2015	3	1
2	VETERANSUSG	Wed, 11 Nov 2015	2015	3	2
3	VETERANSUSPS	Wed, 11 Nov 2015	2015	3	3
4	VETERANS	Fri, 11 Nov 2016	2016	3	1
5	VETERANSUSG	Fri, 11 Nov 2016	2016	3	2
6	VETERANSUSPS	Fri, 11 Nov 2016	2016	3	3
7	VETERANS	Sat, 11 Nov 2017	2017	2	1
8	VETERANSUSPS	Sat, 11 Nov 2017	2017	2	2
9	VETERANS	Sun, 11 Nov 2018	2018	1	1
10	VETERANS	Mon, 11 Nov 2019	2019	3	1
11	VETERANSUSG	Mon, 11 Nov 2019	2019	3	2
12	VETERANSUSPS	Mon, 11 Nov 2019	2019	3	3
13	VETERANS	Wed, 11 Nov 2020	2020	3	1
14	VETERANSUSG	Wed, 11 Nov 2020	2020	3	2
15	VETERANSUSPS	Wed, 11 Nov 2020	2020	3	3
16	VETERANS	Thu, 11 Nov 2021	2021	3	1
17	VETERANSUSG	Thu, 11 Nov 2021	2021	3	2
18	VETERANSUSPS	Thu, 11 Nov 2021	2021	3	3

HOLIDAYNAME Function

Returns the name of the holiday that corresponds to the SAS date or a blank string if a holiday is not defined for the SAS date.

Category: Date and Time

Syntax

HOLIDAYNAME(*date*<,*n*<,'*locale*'>>)

Required Argument

date
specifies a SAS date value in the form '*ddmonyy*'d or '*ddmonyyyy*'d.

dd	is a two-digit integer that represents the day of the month.
mon	is a three-character string that represents the month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.
yy or *yyyy*	is a two- or four-digit integer that represents the year. If only two digits are used, the YEARCUTOFF= option is valid.

Optional Arguments

n
specifies the index.

'locale'
specifies the POSIX locale value. See "Locale-to-Encoding Mapping" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Details

The EVENTDS= option makes the date keys available. You can define the date keys with the DATEKEYS procedure. Also, you assign a locale to a date key with the DATEKEYS procedure. If you do not assign the locale to the date keys, then the predefined date keys do not have locales. If the locale argument is specified, then the date key must be defined using the DATEKEYS procedure, or a predefined date key needs to have a locale set that uses the DATEKEYS procedure. Then the new definition needs to be made available with the EVENTDS= system option.

Example

The following example specifies the name of the holiday:

```
data a;
   length Holiday $32;
   format date weekdatx17.;
   do year=2015 to 2021;
      date=mdy(11,11,year);
      ObsVetNov11=holidaycount(date);
      do index=1 to ObsVetNov11;
         Holiday=holidayname(date,index);
         output;
      end;
   end;
run;
proc print data=a;
run;
```

These statements produce these results:

Output 3.37 *The Name of the Holiday That Corresponds to the SAS Date*

The SAS System

Obs	Holiday	date	year	ObsVetNov11	index
1	VETERANS	Wed, 11 Nov 2015	2015	3	1
2	VETERANSUSG	Wed, 11 Nov 2015	2015	3	2
3	VETERANSUSPS	Wed, 11 Nov 2015	2015	3	3
4	VETERANS	Fri, 11 Nov 2016	2016	3	1
5	VETERANSUSG	Fri, 11 Nov 2016	2016	3	2
6	VETERANSUSPS	Fri, 11 Nov 2016	2016	3	3
7	VETERANS	Sat, 11 Nov 2017	2017	2	1
8	VETERANSUSPS	Sat, 11 Nov 2017	2017	2	2
9	VETERANS	Sun, 11 Nov 2018	2018	1	1
10	VETERANS	Mon, 11 Nov 2019	2019	3	1
11	VETERANSUSG	Mon, 11 Nov 2019	2019	3	2
12	VETERANSUSPS	Mon, 11 Nov 2019	2019	3	3
13	VETERANS	Wed, 11 Nov 2020	2020	3	1
14	VETERANSUSG	Wed, 11 Nov 2020	2020	3	2
15	VETERANSUSPS	Wed, 11 Nov 2020	2020	3	3
16	VETERANS	Thu, 11 Nov 2021	2021	3	1
17	VETERANSUSG	Thu, 11 Nov 2021	2021	3	2
18	VETERANSUSPS	Thu, 11 Nov 2021	2021	3	3

HOLIDAYNX Function

Returns the *n*th occurrence of the holiday relative to the date argument.

Category: Date and Time

Syntax

HOLIDAYNX('*holiday*',*date*<,*n*<,'*locale*'>>)

Required Arguments

'holiday'
is a character constant, variable, or expression that specifies one of the values that are listed in the following table. The *holiday* variable can also be defined

using the DATEKEYS procedure and is made available to the function by using the EVENTDS= system option.

Table 3.59 *Holiday Values and Their Descriptions*

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canada Day	July 1
CANADAOBSERVED	Canada Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25
COLUMBUS	Columbus Day	2nd Monday in October
JUNETEENTH	Juneteenth	June 19
JUNETEENTHUSG	Juneteenth - U.S. government-observed	Juneteenth date observed by U.S. government for Monday–Friday schedule
JUNETEENTHUSPS	Juneteenth - U.S. post office -observed	Juneteenth date observed by U.S. government for Monday–Saturday schedule (U.S. Postal Service)
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)
MOTHERS	Mother's Day	2nd Sunday in May

Holiday Value	Description	Date Celebrated
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

date
specifies the beginning date value in the form '*ddmonyy*'D or '*ddmonyyyy*'D.

dd	is a two-digit integer that represents the day of the month.
mon	is a three-character string that represents the month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.
yy or *yyyy*	is a two- or four-digit integer that represents the year. If only two digits are used, the YEARCUTOFF= option is valid.

Optional Arguments

n
specifies the number of occurrences before or after the date. Specifying 0 for *n* indicates the occurrence on or immediately preceding the date. Specifying 1 for *n* indicates the next occurrence immediately after the date.

'locale'
specifies the POSIX locale value. See "Locale-to-Encoding Mapping" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Details

The *locale* argument is optional and should be the POSIX locale string. When *locale* is specified, the holiday date key should be defined using the DATEKEYS procedure. If the locale is specified, then only date keys with the specified locale or no locale defined that uses the EVENTDS= option are available.

Example

The following example specifies the nth occurrence of the holiday:

```
data holidays;
input holiday $char18.;
datalines;
Easter
Christmas
;
run;
data WhenIs;
     set holidays;
     Today='25DEC2021'D;
     put Today=;
     put holiday=;
     NextHoliday = HOLIDAYNX(holiday,today,1);
     PreviousHoliday = HOLIDAYNX(holiday,NextHoliday,-1);
     if ( PreviousHoliday = today ) then do;
         ThisDayIs=holiday;
         put ThisDayIs=;
         PreviousHoliday = HOLIDAYNX(holiday,today,-1);
     end;
     put PreviousHoliday=;
     put NextHoliday=;
     format Today NextHoliday PreviousHoliday DATE.;
run;
```

SAS writes these results to the log:

```
Today=25DEC21
holiday=Easter
PreviousHoliday=04APR21
NextHoliday=17APR22
Today=25DEC21
holiday=Christmas
ThisDayIs=Christmas
PreviousHoliday=25DEC20
NextHoliday=25DEC22
```

HOLIDAYNY Function

Returns the *n*th occurrence of the holiday for the year.

Category: Date and Time

Syntax

HOLIDAYNY('*holiday*',*year*<,*n*<,'*locale*'>>)

Required Arguments

'holiday'
is a character constant, variable, or expression that specifies one of the values that are listed in the following table. The *holiday* variable can also be defined using the DATEKEYS procedure and is made available to the function by using the EVENTDS= system option.

Table 3.60 *Holiday Values and Their Descriptions*

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canada Day	July 1
CANADAOBSERVED	Canada Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25
COLUMBUS	Columbus Day	2nd Monday in October
JUNETEENTH	Juneteenth	June 19
JUNETEENTHUSG	Juneteenth - U.S. government-observed	Juneteenth date observed by U.S. government for Monday–Friday schedule
JUNETEENTHUSPS	Juneteenth - U.S. post office -observed	Juneteenth date observed by U.S. government for Monday–Saturday

Holiday Value	Description	Date Celebrated
		schedule (U.S. Postal Service)
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)
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)

Holiday Value	Description	Date Celebrated
VICTORIA	Victoria Day	Monday on or preceding May 24

year
is a numeric constant, variable, or expression that specifies a four-digit year. If you use a two-digit year, you must specify the YEARCUTOFF= system option.

Optional Arguments

n
specifies the occurrence of the holiday within the year.

'locale'
specifies the POSIX locale value. See "Locale-to-Encoding Mapping" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Details

The EVENTDS= option makes the date keys available. You can define the date keys with the DATEKEYS procedure. Also, you can assign a locale to a date key with the DATEKEYS procedure. If you do not assign the locale to the date keys, then the predefined date keys do not have locales. If the locale argument is specified, then the date key must be defined using the DATEKEYS procedure, or a predefined date key needs to have a locale set that uses the DATEKEYS procedure. Then the new definition needs to be made available with the EVENTDS= system option.

Example

The following example specifies the *n*th occurrence of the holiday for the year.

```
proc datekeys;
   datekeydef Sabbath=Saturday;
   datekeydata out=Sabbath;
run;

option eventds=(Sabbath);

data a;
   count = holidayck('Sabbath','01JAN2021'd,'31DEC2021'd);
   do i=1 to count;
        Sabbath=holidayny('Sabbath',2021,i);
        put Sabbath=;
     end;
     format Sabbath weekdatx17.;
run;
```

SAS writes these results to the log:

```
Sabbath=Sat, 2 Jan 2021
Sabbath=Sat, 9 Jan 2021
Sabbath=Sat, 16 Jan 2021
Sabbath=Sat, 23 Jan 2021
Sabbath=Sat, 30 Jan 2021
Sabbath=Sat, 6 Feb 2021
Sabbath=Sat, 13 Feb 2021
Sabbath=Sat, 20 Feb 2021
Sabbath=Sat, 27 Feb 2021
Sabbath=Sat, 6 Mar 2021
Sabbath=Sat, 13 Mar 2021
Sabbath=Sat, 20 Mar 2021
Sabbath=Sat, 27 Mar 2021
Sabbath=Sat, 3 Apr 2021
Sabbath=Sat, 10 Apr 2021
Sabbath=Sat, 17 Apr 2021
Sabbath=Sat, 24 Apr 2021
Sabbath=Sat, 1 May 2021
Sabbath=Sat, 8 May 2021
Sabbath=Sat, 15 May 2021
Sabbath=Sat, 22 May 2021
Sabbath=Sat, 29 May 2021
Sabbath=Sat, 5 Jun 2021
Sabbath=Sat, 12 Jun 2021
Sabbath=Sat, 19 Jun 2021
Sabbath=Sat, 26 Jun 2021
Sabbath=Sat, 3 Jul 2021
Sabbath=Sat, 10 Jul 2021
Sabbath=Sat, 17 Jul 2021
Sabbath=Sat, 24 Jul 2021
Sabbath=Sat, 31 Jul 2021
Sabbath=Sat, 7 Aug 2021
Sabbath=Sat, 14 Aug 2021
Sabbath=Sat, 21 Aug 2021
Sabbath=Sat, 28 Aug 2021
Sabbath=Sat, 4 Sep 2021
Sabbath=Sat, 11 Sep 2021
Sabbath=Sat, 18 Sep 2021
Sabbath=Sat, 25 Sep 2021
Sabbath=Sat, 2 Oct 2021
Sabbath=Sat, 9 Oct 2021
Sabbath=Sat, 16 Oct 2021
Sabbath=Sat, 23 Oct 2021
Sabbath=Sat, 30 Oct 2021
Sabbath=Sat, 6 Nov 2021
Sabbath=Sat, 13 Nov 2021
Sabbath=Sat, 20 Nov 2021
Sabbath=Sat, 27 Nov 2021
Sabbath=Sat, 4 Dec 2021
Sabbath=Sat, 11 Dec 2021
Sabbath=Sat, 18 Dec 2021
Sabbath=Sat, 25 Dec 2021
```

HOLIDAYTEST Function

Returns 1 if the holiday occurs on the SAS date value.

Category: Date and Time

Syntax

HOLIDAYTEST('*holiday*',*date*<,'*locale*'>)

Required Arguments

'holiday'

is a character constant, variable, or expression that specifies one of the values that are listed in the following table. The *holiday* variable can also be defined using the DATEKEYS procedure and is made available to the function that uses the EVENTDS= system option.

Table 3.61 *Holiday Values and Their Descriptions*

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canada Day	July 1
CANADAOBSERVED	Canada Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25
COLUMBUS	Columbus Day	2nd Monday in October
JUNETEENTH	Juneteenth	June 19
JUNETEENTHUSG	Juneteenth - U.S. government-observed	Juneteenth date observed by U.S. government for Monday–Friday schedule
JUNETEENTHUSPS	Juneteenth - U.S. post office -observed	Juneteenth date observed by U.S. government for Monday–Saturday schedule (U.S. Postal Service)
EASTER	Easter Sunday	date varies
FATHERS	Father's Day	3rd Sunday in June
HALLOWEEN	Halloween	October 31

Holiday Value	Description	Date Celebrated
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)
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

date

specifies the beginning date value in the form '*ddmonyy*'d or '*ddmonyyyy*'d.

dd is a two-digit integer that represents the day of the month.

mon	is a three-character string that represents the month: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC.
yy or *yyyy*	is a two- or four-digit integer that represents the year. If only two digits are used, the YEARCUTOFF= option is valid.

Optional Argument

'locale'
specifies the POSIX locale value. See "Locale-to-Encoding Mapping" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Details

The EVENTDS= option makes the date keys available. You can define the date keys with the DATEKEYS procedure. Also, you can assign a locale to a date key with the DATEKEYS procedure. If you do not assign the locale to the date keys, then the predefined date keys do not have locales. If the locale argument is specified, then the date key must be defined using the DATEKEYS procedure, or a predefined date key needs to have a locale set that uses the DATEKEYS procedure. Then the new definition needs to be made available with the EVENTDS= system option.

Example

The following example specifies whether the holiday occurs on the SAS date:

```
data a;
   length is $9;
   holiday='Christmas';
   test=holidaytest(holiday,today());
   if (test=1) then is='today';
      else is='not today';
   put holiday= is=;
run;
```

SAS writes these results to the log:

```
holiday=Christmas is=not today
```

HOUR Function

Returns the hour from a SAS time or datetime value.

Category: Date and Time

Syntax

HOUR(*time* | *datetime*)

Required Arguments

time
is a numeric constant, variable, or expression that specifies a SAS time value.

datetime
is a numeric constant, variable, or expression that specifies a SAS datetime value.

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

```
data
one;


now='1:30't;


h=hour(now);

   put
h=;

run;
```

These statements produce this result:

```
h=1
```

See Also

Functions:

- “SECOND Function” on page 1381

HTMLDECODE Function

Decodes a string that contains HTML numeric character references or HTML character entity references, and returns the decoded string.

Category: Web Tools

Syntax

HTMLDECODE(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression.

Details

The HTMLDECODE function recognizes the following character entity references:

Character Entity Reference	Decoded Character
&	&
<	<
>	>
"	"
'	'
&copy	©
&reg	®
&trade	™

Unrecognized entities (&<name>;) are left unmodified in the output string.

The HTMLDECODE function recognizes numeric entity references that are of the form

`&#nnn;`
where *nnn* specifies a decimal number that contains one or more digits.

`&#Xnnn;`
where *nnn* specifies a hexadecimal number that contains one or more digits.

Note: Numeric character references that cannot be represented in the current SAS session encoding will not be decoded. The reference will be copied unchanged to the output string.

Example

```
data
one;

   x1=htmldecode('not a
<tag>');


x2=htmldecode('&');

   x3=htmldecode
('ABC');

   x4=htmldecode
('©®™');

   put
_all_;

run;
```

These statements produce these results:

```
x1=not a <tag> x2='&' x3=ABC x4=© ® ™
```

See Also

Functions:

- “HTMLENCODE Function” on page 937

HTMLENCODE Function

Encodes characters using HTML character entity references, and returns the encoded string.

Category: Web Tools

Syntax

HTMLENCODE(*expression*, <*options*>)

Required Argument

expression
specifies a character constant, variable, or expression. By default, any greater-than (>), less-than (<), and ampersand (&) characters are encoded as `>`, `<`, and `&`, respectively. This behavior can be modified with the *option* argument.

Note: The encoded string can be longer than the output string. You should take the additional length into consideration when you define your output variable. If the encoded string exceeds the maximum length that is defined, the output string might be truncated.

Optional Argument

option
is a character constant, variable, or expression that specifies the type of characters to encode. If you use more than one option, separate the options by spaces.

The following options are available:

Option	Character	Character Entity Reference	Description
amp	&	&	The HTMLENCODE function encodes these characters by default. If you need to encode these characters only, then you do not need to specify the options argument. However, if you specify any value for the options argument, then the defaults are overridden, and you must explicitly specify the options for all of the characters that you want to encode.
gt	>	>	
lt	<	<	
apos	'	'	Use this option to encode the apostrophe (') character in text that is used in an HTML or XML tag attribute.

Option	Character	Character Entity Reference	Description
quot	"	"	Use this option to encode the double quotation mark (") character in text that is used in an HTML or XML tag attribute.
7bit	any character that is not represented in 7-bit ASCII encoding	&#x*nnn*; (Unicode)	*nnn* is a one or more digit hexadecimal number. Encode these characters to create HTML or XML that is easily transferred through communication paths that might support only 7-bit ASCII encodings (for example, ftp or email).
copy	©	©	Use this option in the HTMLENCODE function to turn on the encoding for this character. The value of val is **&copy** after the following code is processed: `val = htmlencode('©', 'copy');`
reg	®	®	Use this option in the HTMLENCODE function to turn on the encoding for this character. The value of val is **&reg** after the following code is processed: `val = htmlencode('®', 'reg');`
trade	™	™	Use this option in the HTMLENCODE function to turn on the encoding for this character. The value of val is **&trade** after the following code is processed: `val = htmlencode('™', 'trade');`

Example

```
data
one;

   v1=htmlencode("John's test
<tag>");
```

```
   v2=htmlencode("John's test
<tag>",'apos');

   v3=htmlencode('John "Jon"
Smith,tag>','quot');

   v4=htmlencode("'A&B&C'", 'amp lt gt
apos');

   v5=htmlencode('80'x,
'7bit');

   copy = htmlencode('©',
'copy');

   reg = htmlencode('®',
'reg');

   trade = htmlencode('™',
'trade');

   put v1= v2= v3= v4= v5= copy= reg=
trade=;

run;
```

These statements produce these results:

```
v1=John's test &lt;tag&gt;
v2=John&apos;s test <tag>
v3=John &quot;Jon&quot; Smith <tag>
v4=&apos;A&amp;B&amp;C&apos;
v5=&#x20AC;
copy=&copy;
reg=&reg;
trade=&trade;
```

See Also

Functions:

- “HTMLDECODE Function” on page 936

IBESSEL Function

Returns the value of the modified Bessel function.

Category: Mathematical

Syntax

IBESSEL(*nu*, *x*, *kode*)

Required Arguments

nu

specifies a numeric constant, variable, or expression.

Range *nu* ≥ 0

x

specifies a numeric constant, variable, or expression.

Range *x* ≥ 0

kode

is a numeric constant, variable, or expression that specifies a nonnegative integer.

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

```
data
one;

   x=ibessel(2, 2,
0);

   y=ibessel(2, 2,
1);

   put x=
y=;

run;
```

These statements produce these results:

```
x=0.6889484477 y=0.0932390333
```

IFC Function

Returns a character value based on whether an expression is true, false, or missing.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

IFC(*logical-expression*, *value-returned-when-true*, *value-returned-when-false* <, *value-returned-when-missing*>)

Required Arguments

logical-expression
specifies a numeric constant, variable, or expression.

value-returned-when-true
specifies a character constant, variable, or expression that is returned when the value of *logical-expression* is true.

value-returned-when-false
specifies a character constant, variable, or expression that is returned when the value of *logical-expression* is false.

Optional Argument

value-returned-when-missing
specifies a character constant, variable, or expression that is returned when the value of *logical-expression* is missing.

Details

Length of Returned Variable

In a DATA step, if the IFC 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.

The Basics

The IFC function uses conditional logic that enables you to select among several values based on the value of a logical expression.

IFC evaluates the first argument, *logical-expression*. If *logical-expression* is true (that is, not zero and not missing), then IFC returns the value in the second argument. If *logical-expression* is a missing value, and you have a fourth argument, then IFC returns the value in the fourth argument. Otherwise, if *logical-expression* is false, IFC returns the value in the third argument.

The IFC function is useful in DATA step expressions, and even more useful in WHERE clauses and other expressions where it is not convenient or possible to use an IF/THEN/ELSE construct.

Comparisons

The IFC function is similar to the IFN function except that IFC returns a character value while IFN returns a numeric value.

Examples

Example 1

In the following example, IFC evaluates the expression `grade>80` to implement the logic that determines the performance of several members on a team.

```
data _null_;
   input name $ grade;
   performance = ifc(grade>80, 'Pass             ', 'Needs
Improvement');
   put name= performance=;
   datalines;
John 74
Kareem 89
Kati 100
Maria 92
;
```

SAS writes the following output to the log:

```
name=John performance=Needs Improvement
name=Kareem performance=Pass
name=Kati performance=Pass
name=Maria performance=Pass
```

Example 2

This example uses an IF/THEN/ELSE construct to generate the same output that is generated by the IFC function.

```
data _null_;
```

```
   input name $ grade;
   if grade>80 then performance='Pass             ';
      else performance = 'Needs Improvement';
   put name= performance=;
   datalines;
John 74
Sam 89
Kati 100
Maria 92
;
```

SAS writes the following output to the log:

```
name=John performance=Needs Improvement
name=Sam performance=Pass
name=Kati performance=Pass
name=Maria performance=Pass
```

See Also

Functions:

- "IFN Function" on page 944

IFN Function

Returns a numeric value based on whether an expression is true, false, or missing.

Category: Mathematical

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

IFN(*logical-expression*, *value-returned-when-true*, *value-returned-when-false* <, *value-returned-when-missing*>)

Required Arguments

logical-expression
specifies a numeric constant, variable, or expression.

value-returned-when-true
specifies a numeric constant, variable, or expression that is returned when the value of *logical-expression* is true.

value-returned-when-false
specifies a numeric constant, variable, or expression that is returned when the value of *logical-expression* is false.

Optional Argument

value-returned-when-missing
specifies a numeric constant, variable or expression that is returned when the value of *logical-expression* is missing.

Details

The IFN function uses conditional logic that enables you to select among several values based on the value of a logical expression.

IFN evaluates the first argument, then *logical-expression*. If *logical-expression* is true (that is, not zero and not missing), then IFN returns the value in the second argument. If *logical-expression* is a missing value, and you have a fourth argument, then IFN returns the value in the fourth argument. Otherwise, if *logical-expression* is false, IFN returns the value in the third argument.

The IFN function, an IF/THEN/ELSE construct, or a WHERE statement can produce the same results. (See examples.) However, the IFN function is useful in DATA step expressions when it is not convenient or possible to use an IF/THEN/ELSE construct or a WHERE statement.

Comparisons

The IFN function is similar to the IFC function, except that IFN returns a numeric value whereas IFC returns a character value.

Examples

Example 1: Calculating Commission Using the IFN Function

In the following example, IFN evaluates the expression `TotalSales > 10000`. If total sales exceeds $10,000, then the sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales.

```
data _null_;
   input TotalSales;
   commission=ifn(TotalSales > 10000, TotalSales*.05, TotalSales*.02);
   put commission=;
   datalines;
25000
10000
500
```

```
10300
;
```

SAS writes the following output to the log:

```
commission=1250
commission=200
commission=10
commission=515
```

Example 2: Calculating Commission Using an IF/THEN/ELSE Construct

In the following example, an IF/THEN/ELSE construct evaluates the expression `TotalSales > 10000`. If total sales exceeds $10,000, then the sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales.

```
data _null_;
   input TotalSales;
   if TotalSales > 10000 then commission = .05 * TotalSales;
      else commission = .02 * TotalSales;
   put commission=;
   datalines;
25000
10000
500
10300
;
```

SAS writes the following output to the log:

```
commission=1250
commission=200
commission=10
commission=515
```

Example 3: Calculating Commission Using a WHERE Statement

In the following example, a WHERE statement evaluates the expression `TotalSales > 10000`. If total sales exceeds $10,000, then the sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales. The output shows only those salespeople whose total sales exceed $10,000.

```
data sales;
   input SalesPerson $ TotalSales;
   datalines;
Michaels 25000
Janowski 10000
Chen 500
Gupta 10300
;
data commission;
   set sales;
   where TotalSales > 10000;
```

```
   commission = TotalSales * .05;
run;
proc print data=commission;
   title 'Commission for Total Sales > 1000';
run;
```

Output 3.38 *Output from a WHERE Statement*

Commission for Total Sales > 1000

Obs	SalesPerson	TotalSales	commission
1	Michaels	25000	1250
2	Gupta	10300	515

See Also

Functions:

- "IFC Function" on page 942

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.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: DBCS equivalent function is KINDEX in *SAS National Language Support (NLS): Reference Guide*. See "DBCS Compatibility" on page 948.

Syntax

INDEX(*source*, *excerpt*)

Required Arguments

source
specifies a character constant, variable, or expression to search.

excerpt
is a character constant, variable, or expression that specifies the string of characters to search for in *source*.

Tips Enclose a literal string of characters in quotation marks.

Both leading and trailing spaces are considered part of the *excerpt* argument. To remove trailing spaces, include the TRIM function with the *excerpt* variable inside the INDEX function.

Details

The Basics

The INDEX function searches *source*, from left to right, for the first occurrence of the string specified in *excerpt*, and returns the position in *source* of the string's first character. If the string is not found in *source*, INDEX returns a value of 0. If there are multiple occurrences of the string, INDEX returns only the position of the first occurrence.

The INDEX function returns the character position when processing a varchar column. The INDEX function returns the byte position when processing a character column.

DBCS Compatibility

The DBCS equivalent function is KINDEX. For more information, see "KINDEX Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Examples

Example 1: Finding the Position of a Variable in the Source String

The following example finds the first position of the *excerpt* argument in *source*.

```
data _null_;
   a = 'ABC.DEF(X=Y)';
   b = 'X=Y';
   x = index(a, b);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=9
```

Example 2: Removing Trailing Spaces When You Use the INDEX Function with the TRIM Function

The following example shows the results when you use the INDEX function with and without the TRIM function. If you use INDEX without the TRIM function, leading and trailing spaces are considered part of the *excerpt* argument. If you use INDEX with the TRIM function, TRIM removes trailing spaces from the *excerpt* argument as you can see in this example. Note that the TRIM function is used inside the INDEX function.

```
options nodate nostimer ls=78 ps=60;
data _null_;
   length a b $14;
   a='ABC.DEF (X=Y)';
   b='X=Y';
   q=index(a, b);
   w=index(a, trim(b));
   put q= w=;
run;
```

SAS writes the following output to the log:

```
q=0 w=10
```

See Also

Functions:

- “FIND Function” on page 679
- “INDEXC Function” on page 949
- “INDEXW Function” on page 951

INDEXC Function

Searches a character expression for any of the specified characters, and returns the position of that character.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: DBCS equivalent function is KINDEXC in *SAS National Language Support (NLS): Reference Guide*.

Syntax

INDEXC(*source*, *excerpt-1* <, ...*excerpt-n*>)

Required Arguments

source
specifies a character constant, variable, or expression to search.

excerpt
specifies the character constant, variable, or expression to search for in *source*.

Tip If you specify more than one excerpt, separate them with a comma.

Details

The INDEXC function searches *source*, from left to right, for the first occurrence of any character present in the excerpts and returns the position in *source* of that character. If none of the characters in *excerpt-1* through *excerpt-n* in *source* are found, INDEXC returns a value of 0.

Comparisons

The INDEXC function searches for the first occurrence of any individual character that is present within the character string, whereas the INDEX function searches for the first occurrence of the character string as a substring. The FINDC function provides more options.

Example

```
data
one;

   a='ABC.DEP
(X2=Y1)';

   x=indexc(a, '0123', ';
()=.');

   put
x=;
```

```
   b='have a good
day';

   x=indexc(b, 'pleasant',
'very');

   put
x=;

run;
```

These statements produce these results:

```
x=4
x=2
```

See Also

Functions:

- “FINDC Function” on page 683
- “INDEX Function” on page 947
- “INDEXW Function” on page 951

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.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Syntax

INDEXW(*source*, *excerpt* <, *delimiters*>)

Required Arguments

source
specifies a character constant, variable, or expression to search.

excerpt
specifies a character constant, variable, or expression to search for in *source*. SAS removes leading and trailing delimiters from *excerpt*.

Optional Argument

delimiter
specifies a character constant, variable, or expression containing the characters that you want INDEXW to use as delimiters in the character string. The default delimiter is the blank character.

Details

The INDEXW function searches *source*, from left to right, for the first occurrence of *excerpt* and returns the position in *source* of the substring's first character. If the substring is not found in *source*, 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 *source*, and the end of *source*. If you use an alternate delimiter, then INDEXW does not recognize the end of the text as the end data.

INDEXW has the following behavior when the second argument contains blank spaces or has a length of 0:

- If both *source* and *excerpt* contain only blank spaces or have a length of 0, then INDEXW returns a value of 1.
- If *excerpt* contains only blank spaces or has a length of 0, and *source* 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. The FINDW function provides more options.

Examples

Example 1: INDEXW Examples

```
data
one;
```

```
   s='asdf adog
dog';

   p='dog
';

   a=indexw(s,
p);

   put
a=;



   s='abcdef
x=y';


p='def';

   b=indexw(s,
p);

   put
b=;



   c="abc,def@
xyz";

   abc=indexw(c, " abc ",
"@");

   put
abc=;



   d="abc,def@
xyz";

   comma=indexw(d, ",",
"@");

   put
comma=;



   e='abc,def%
xyz';

   def=indexw(e, 'def',
'%,');
```

```
   put
def=;


   f="abc,def@
xyz";
   at=indexw(f, "@",
"@");
   put
at=;


   g="abc,def@
xyz";
   xyz=indexw(g, " xyz",
"@");
   put
xyz=;


   h=indexw(trimn(' '),
' ');
   put
h=;


   i=indexw(' x y ', trimn('
'));
   put
i=;

run;
```

These statements produce these results:

```
a=11
b=0
abc=0
comma=0
def=5
at=0
xyz=9
h=1
i=0
```

Example 2: Using a Semicolon (;) as the Delimiter

The following example shows how to use the semicolon delimiter in a SAS program that also calls the CATX function. A semicolon delimiter must be in place after each call to CATX, and the second argument in the INDEXW function must be trimmed or searches will not be successful.

```
data temp;
   infile datalines;
   input name $12.;
   datalines;
abcdef
abcdef
;
run;
data temp2;
   set temp;
   format name_list $1024.;
   retain name_list ' ';
   exists=indexw(name_list, trim(name), ';');
   if exists=0 then do
      name_list=catx(';', name_list, name)||';' ;
      name_count +1;
      put exists= ;
      put name_list= ;
      put name_count= ;
      end;
    run;
```

These statements produce these results:

```
exists=0
name_list=abcdef;
name_count=1
```

In this example, the first time CATX is called *name_list* is blank and the value of *name* is 'abcdef'. CATX returns 'abcdef' with no semicolon appended. However, when INDEXW is called the second time, the value of *name_list* is 'abcdef' followed by 1018 (1024–6) blanks, and the value of *name* is 'abcdef' followed by six blanks. Because the third argument in INDEXW is a semicolon (;), the blanks are significant and do not denote a word boundary. Therefore, the second argument cannot be found in the first argument.

If the example has no blanks, the behavior of INDEXW is easier to understand. In the following example, we expect the value of *x* to be 0 because the complete word ABCDE was not found in the first argument:

```
x=indexw('ABCDEF;XYZ', 'ABCDE', ';');
```

The only values for the second argument that would return a nonzero result are ABCDEF and XYZ.

Example 3: Using a Space as the Delimiter

The following example uses a space as a delimiter:

```
data temp;
```

```
   infile datalines;
   input name $12.;
   datalines;
abcdef
abcdef
;
run;
data temp2;
   set temp;
   format name_list $1024.;
   retain name_list ' ';
   exists=indexw(name_list, name, ' ');
   if exists=0 then do
      name_list=catx(' ', name_list, name) ;
      name_count +1;
      put exists= ;
      put name_list= ;
      put name_count= ;
      end;
run;
```

These statements produce these results:

```
exists=0
name_list=abcdef
name_count=1
```

See Also

Functions:

- “FINDW Function” on page 692
- “INDEX Function” on page 947
- “INDEXC Function” on page 949

INPUT Function

Returns the value that is produced when SAS converts an expression by using the specified informat.

Category: Special

Syntax

INPUT(*source*, <? | ??> *informat*.)

Required Arguments

source
specifies a character constant, variable, or expression to which you want to apply a specific informat.

? or ??
specifies the optional question mark (?) and double question mark (??) modifiers that suppress the printing of notes and input lines when invalid data values are read. The ? modifier suppresses the invalid data message. The ?? modifier suppresses the invalid data message and prevents the automatic variable _ERROR_ from being set to 1 when invalid data is read.

informat.
is the SAS informat that you want to apply to the source. This argument must be the name of an informat followed by a period. The argument cannot be a character constant, variable, or expression.

Details

If the INPUT function returns a character value to a variable that has not yet been assigned a length, the variable length is determined by the width of the informat.

The INPUT function enables you to convert the value of *source* by using a specified informat. The informat determines whether the result is numeric or character. Use INPUT to convert character values to numeric values or other character values.

Comparisons

The INPUT function returns the value produced when a SAS expression is converted using a specified informat. You must use an assignment statement to store that value in a variable. The INPUT statement uses an informat to read a data value. Storing that value in a variable is optional.

The INPUT function requires the informat to be specified as a name followed by a period and optional decimal specification. The INPUTC and INPUTN functions allow the informat to be specified as a character constant, variable, or expression.

Examples

Example 1: Converting Character Values to Numeric Values

This example uses the INPUT function to convert a character value to a numeric value and store the value in another variable. The COMMA9. informat reads the value of the SALE variable, stripping the commas. The resulting value, 2115353, is stored in FMTSALE.

```
data testin;
```

```
   input sale $9.;
   fmtsale=input(sale, comma9.);
   datalines;
2,115,353
;
```

Example 2: Using PUT and INPUT Functions

In this example, PUT returns a numeric value as a character string. The value 122591 is assigned to the CHARDATE variable. INPUT returns the value of the character string as a SAS date value by using a SAS date informat. The value 11681 is stored in the SASDATE variable.

```
data
one;


numdate=122591;

   chardate=put(numdate,
z6.);

   sasdate=input(chardate,
mmddyy6.);

   put
_all_;

run;
```

```
numdate=122591 chardate=122591 sasdate=11681 _ERROR_=0 _N_=1
```

Example 3: Suppressing Error Messages

In this example, the question mark (?) modifier tells SAS not to print the invalid data notes if it finds data errors. The automatic variable _ERROR_ is set to 1, and input data lines are written to the SAS log.

```
y=input(x, ? 3.1);
```

The double question mark (??) modifier suppresses the printing of invalid data notes and input lines and prevents the automatic variable _ERROR_ from being set to 1 when invalid data is read. Therefore, these two examples produce the same result:

- `y=input(x, ?? 2.);`
- `y=input(x, ? 2.); _error_=0;`

```
data
one;


x='abc';
```

```
   y=input(x, ?
3.1);



 /* The double question mark (??) modifier suppresses the printing of
invalid data notes and input lines and prevents the automatic variable
_ERROR_ from being
set to 1 when invalid data is read. Therefore, these two examples
produce the
same result
*/


   y=input(x, ??
2.);

   y=input(x, ?
2.);

run;
```

These statements produce these results:

```
x=abc y=. _ERROR_=1 _N_=1
```

See Also

Functions:

- “INPUTC Function” on page 959
- “INPUTN Function” on page 962
- “PUT Function” on page 1281
- “PUTC Function” on page 1284
- “PUTN Function” on page 1288

Statements:

- “INPUT Statement” in *SAS DATA Step Statements for SAS Viya Workbench*

INPUTC Function

Enables you to specify a character informat at run time.

Category: Special

Note: This function supports the VARCHAR type.

Syntax

INPUTC(*source*, *informat* <, *w*>)

Required Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

informat
is a character constant, variable, or expression that contains the character informat that you want to apply to *source*.

Optional Argument

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.

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 set to 200.

Comparisons

The INPUTC function enables you to specify a character informat at run time. Using the INPUT function is faster because you specify the informat at compile time.

Example

This example shows how to specify character informats. The PROC FORMAT step in this example creates a format, TYPEFMT., that formats the variable values 1, 2, and 3 with the name of one of the three informats that this step also creates. The informats store responses of "positive," "negative," and "neutral" as different words, depending on the type of question. After PROC FORMAT creates the format and informats, the DATA step creates a SAS data set from raw data consisting of a number identifying the type of question and a response. After reading a record, the DATA step uses the value of TYPE to create a variable, RESPINF, that contains the value of the appropriate informat for the current type of question. The DATA step also creates another variable, WORD, whose value is the appropriate word for a

response. The INPUTC function assigns the value of WORD based on the type of question and the appropriate informat.

```
proc format;
   value typefmt 1='$groupx'
                 2='$groupy'
                 3='$groupz';
   invalue $groupx 'positive'='agree'
                   'negative'='disagree'
                   'neutral'='notsure';
   invalue $groupy 'positive'='accept'
                   'negative'='reject'
                   'neutral'='possible';
   invalue $groupz 'positive'='pass'
                   'negative'='fail'
                   'neutral'='retest';
run;
data answers;
   input type response $;
   respinformat=put(type, typefmt.);
   word=inputc(response, respinformat);
   datalines;
1 positive
1 negative
1 neutral
2 positive
2 negative
2 neutral
3 positive
3 negative
3 neutral
;
```

The value of WORD for the first observation is `agree`. The value of WORD for the last observation is `retest`.

See Also

Functions:

- “INPUT Function” on page 956
- “INPUTN Function” on page 962
- “PUT Function” on page 1281
- “PUTC Function” on page 1284
- “PUTN Function” on page 1288

INPUTN Function

Enables you to specify a numeric informat at run time.

Category: Special

Note: This function supports the VARCHAR type.

Syntax

INPUTN(*source*, *informat* <, *w*<, *d*>>)

Required Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

informat
is a character constant, variable, or expression that contains the numeric informat that you want to apply to *source*.

Optional Arguments

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.

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.

Comparisons

The INPUTN function enables you to specify a numeric informat at run time. Using the INPUT function is faster because you specify the informat at compile time.

Example: Specifying Numeric Informats

This example shows how to specify numeric informats. The PROC FORMAT step creates a format, READDATE., that formats the variable values 1 and 2 with the name of a SAS date informat. The DATA step creates a SAS data set from raw data originally from two different sources (indicated by the value of the variable SOURCE). Each source specified dates differently. After reading a record, the DATA step uses the value of SOURCE to create a variable, DATEINFORMAT, that contains the value of the appropriate informat for reading the date. The DATA step also creates a new variable, NEWDATE, whose value is a SAS date. The INPUTN function assigns the value of NEWDATE based on the source of the observation and the appropriate informat.

```
proc format;
   value readdate 1='date7.'
                  2='mmddyy8.';
run;
options yearcutoff=1940;
data fixdates(drop=start dateinformat);
   length jobdesc $12;
   input source id lname $ jobdesc $ start $;
   dateinformat=put(source, readdate.);
   newdate=inputn(start, dateinformat);
   datalines;
1 1604 Ziminski writer 09aug99
1 2010 Clavell editor 26jan95
2 1833 Rivera writer 10/25/98
2 2222 Barnes proofreader 3/26/12
;
```

See Also

Functions:

- “INPUT Function” on page 956
- “INPUTC Function” on page 959
- “PUT Function” on page 1281
- “PUTC Function” on page 1284
- “PUTN Function” on page 1288

INT Function

Returns the integer value, fuzzed to avoid unexpected floating-point results.

Category: Rounding and Truncation

Syntax

INT(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The INT function returns the integer portion of the argument (truncates the decimal portion). If the argument's value is within 1E-12 of an integer, the function results in that integer. If the value of *argument* is positive, the INT function has the same result as the FLOOR function. If the value of *argument* 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 an integer, the INT function fuzzes the result to be equal to that integer. The INTZ function does not fuzz the result. Therefore, with the INTZ function you might get unexpected results.

Example

This example demonstrates the INT and INTZ functions.

```
data _null_;
   one       = 1;
   int_one   = int(one);
   intz_one  = intz(one);

   one_minus_delta       = 1 - 1.e-12;
   int_one_minus_delta   = int(one_minus_delta);
   intz_one_minus_delta  = intz(one_minus_delta);

   put 'INFO: ' one= int_one= intz_one=;
   put 'INFO: ' one_minus_delta= int_one_minus_delta=
intz_one_minus_delta=;
run;
```

These statements produce these results:

```
INFO: one=1 int_one=1 intz_one=1
INFO: one_minus_delta=1 int_one_minus_delta=1 intz_one_minus_delta=0
```

See Also

Functions:

- "CEIL Function" on page 424
- "FLOOR Function" on page 710
- "INTZ Function" on page 1020

INTCINDEX Function

Returns the cycle index when a date, time, or datetime interval and value are specified.

Category: Date and Time

Syntax

INTCINDEX(*interval* <<*multiple*.<*shift-index*>>>, *date-time-value*)

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

> **TIP** If *interval* is a character constant, then enclose the value in quotation marks.

> **TIP** Valid values for *interval* depend on whether *date-time-value* is a date, time, or datetime value.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>

The three parts of the interval name are as follows:

interval

specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

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, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH intervals shift by MONTH periods 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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

date-time-value
specifies a date, time, or datetime value that represents a time period of a specified interval.

Details

The INTCINDEX function returns the index of the seasonal cycle when you specify an interval and a SAS date, time, or datetime 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, 2021, is the sixth day of the 35th week of the year.

```
cycle_index=intcindex('day', '01SEP2021'd );
```

For more information about working with date and time intervals, see "Date and Time Intervals" on page 40.

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

"About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

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.

```
cycle_index=intcindex('day', '04APR2021'd);
```

In this example, the INTINDEX function returns the day of the week. In the example

```
index=intindex('day', '04APR2021'd);
```

In this example, the INTCINDEX function returns the hour of the day.

```
cycle_index=intcindex('minute', '01Sep2021:00:00:00'dt);
```

In this example, the INTINDEX function returns the minute of the hour.

```
index=intindex('minute', '01Sep2021:00:00:00'dt);
```

In the example `intseas(intcycle('interval'));`, the INTSEAS function returns the maximum number that could be returned by `intcindex('interval', date);`.

Example

```
data
one;

   cycle_index1=intcindex('day',
'01SEP2021'd);

   put
cycle_index1=;



   cycle_index2=intcindex('dtqtr',
'23MAY2021:05:03:01'dt);

   put
cycle_index2=;



   cycle_index3=intcindex('tenday',
'13DEC2021'd);

   put
cycle_index3=;
```

```
   cycle_index4=intcindex('minute',
'23:13:02't);

   put
cycle_index4=;



var1='semimonth';

   cycle_index5=intcindex(var1,
'05MAY2021:10:54:03'dt);

   put
cycle_index5=;

run;
```

SAS writes these results to the log:

```
cycle_index1=35
cycle_index2=1
cycle_index3=1
cycle_index4=24
cycle_index5=1
```

See Also

Functions:

- “INTCYCLE Function” on page 978
- “INTINDEX Function” on page 993
- “INTSEAS Function” on page 1013

INTCK Function

Returns the number of interval boundaries of a given kind that lie between two dates, times, or datetime values.

Category: Date and Time

Syntax

INTCK(*interval* <*multiple*> <.*shift-index*>, *start-date*, *end-date*, <'*method*'>)

INTCK(*custom-interval*, *start-date*, *end-date*, <'*method*'>)

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

The type of interval (date, datetime, or time) must match the type of value in *start-date*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>

The three parts of the interval name are listed below:

interval

specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40

custom-interval

specifies a user-defined interval that is defined by a SAS data set. Each observation contains two variables, *begin* and *end*.

Requirement You must use the INTERVALDS system option if you use the *custom-interval* variable.

See "Details" on page 971

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

Restrictions The shift index cannot be greater than the number of subperiods in the entire 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, MONTH type intervals shift by MONTH subperiods by default. Thus, 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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40

start-date
specifies a SAS expression that represents the starting SAS date, time, or datetime value.

end-date
specifies a SAS expression that represents the ending SAS date, time, or datetime value.

Optional Argument

'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.

The continuous method is useful for calculating anniversaries. For example, you can calculate the number of years a couple is married by executing the following program:

```
data b;
   WeddingDay='14feb2021'd;
   Today=today();
   YearsMarried=intck('YEAR', WeddingDay, today(), 'C');
   format WeddingDay Today date9.;
put WeddingDay=;
put Today=;
put YearsMarried=;
run;
```

The results are:

```
WeddingDay=14FEB2021
Today=05MAR2021
YearsMarried=0
```

For the CONTINUOUS method, the distance in months between February 14, 2021, and March 12, 2021, is zero months.

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, 2021, and February 1, 2021, is one month.

Alias D or DISC

Default DISCRETE

Details

Calendar Interval Calculations

All values within a discrete time interval are interpreted as being equivalent. This means that the dates of January 1, 2021 and January 15, 2021 are equivalent when you specify a monthly interval. Both of these dates represent the interval that begins on January 1, 2021 and ends on January 31, 2021. You can use the date for the beginning of the interval (January 1, 2021) or the date for the end of the interval (January 31, 2021) to identify the interval. These dates represent all of the dates within the monthly interval.

In the following example, the *start-date* ('14JAN2021'd), is equivalent to the first quarter of 2021.

```
intck('qtr', '14JAN2021'd, '02SEP2021'd);
```

The *end-date* ('02SEP2021'd) is equivalent to the third quarter of 2021. 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 function `INTCK('MONTH', '1jan2021'd, '31jan2021'd)` returns 0, because the two dates are within the same month.
- The function `INTCK('MONTH', '31jan2021'd, '1feb2021'd)` returns 1, because the two dates lie in different months that are one month apart.
- The function `INTCK('MONTH', '1feb2021'd, '31jan21'd)` 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.)

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.

In this example, there are five months between Oct 31 and March 31. Five months do not equal two quarters in continuous method.

```
2020            2021
Oct | Nov | Dec | Jan | Feb | Mar | Apr |
0     1     2     3     4     5     6      <- # of months
0                 1                 2      <- # of quarters
```

INTCK returns the number of times that an interval boundary is crossed. This number is the delta. If you want to include the current month, you add one number. This number is the span. Sunday to Wednesday spans four days: Sunday, Monday, Tuesday, and Wednesday. The delta from Sunday to Monday is three days: Sunday to Monday, Monday to Tuesday, and Tuesday to Wednesday. INTCK counts the delta, not the span, but the difference between the delta and the span is always one.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 40.

Date and Datetime Intervals

The intervals that you need to use with SAS datetime values are SAS datetime intervals. Datetime intervals are formed by adding the prefix "DT" to any date interval. For example, MONTH is a SAS date interval, and DTMONTH is a SAS datetime interval. Similarly, YEAR is a SAS date interval, and DTYEAR is a SAS datetime interval.

To ensure correct results with interval functions, use date intervals with date values and datetime intervals with datetime values. SAS does not return an error message if you use a date value with a datetime interval, but the results are incorrect.

The following example uses the DTDAY datetime interval and returns the number of days between August 1, 2021, and February 1, 2022:

```
data _null_;
   days=intck('dtday', '01aug2021:00:10:48'dt, '01feb2022:00:10:48'dt);
   put days=;
run;
```

SAS writes the following output to the log:

```
days=184
```

These examples show several ways in which INTCK can be used with macro variables.

This example shows datetime literals in the DATA step.

```
data _null_;
   x=intck('dtmonth','01jan2021:12:34:56'dt,'15mar2021:00:00:00'dt);
   put x=;
run;
```

Here are the results from the code:

```
x=2
```

This example shows datetime literals in the DATA step but using macro variables that are used in double quotation marks:

```
%let from=01jan2021:12:34:56;
%let to=15mar2021:00:00:00;

data _null_;
   x = intck('dtmonth',"&from."dt,"&to."dt);
   put x=;
run;
```

Here are the results from the code:

```
x=2
```

This example shows datetime literals in the DATA step but using macro variables that are used in single quotation marks with the *dt* indicator:

```
%let from='01jan2021:12:34:56'dt;
%let to='15mar2021:00:00:00'dt;

data _null_;
   x = intck('dtmonth',&from.,&to.);
   put x=;
run;
```

Here are the results from the code:

```
x=2
```

This example shows datetime literals in the DATA step but using macro variables that are used in double quotation marks with the *dt* indicator:

```
%let from="01jan2021:12:34:56"dt;
%let to="15mar2021:00:00:00"dt;

data _null_;
   x = intck('dtmonth',&from.,&to.);
   put x=;
run;
```

Here are the results from the code:

```
x=2
```

This example shows datetime literals in the DATA step using numeric values for the macro variables:

```
data _null;
   from = input('01jan2021:12:34:56',datetime19.);
   to = input('15mar2021:00:00:00',datetime19.);
   call symputx('from',from);
   call symputx('to',to);
run;

data _null_;
   x = intck('dtmonth',&from,&to);
   put x=;
run;
```

Here are the results from the code:

```
x=2
```

```
%let x=%sysfunc(intck(dtmonth,&from,&to));
%put x=&x;
```

```
x=2
```

This example demonstrates using datetime literals with %SYSFUNC

```
%let from='01jan2021:12:34:56'dt;
%let to='15mar2021:00:00:00'dt;
%let x=%sysfunc(intck(dtmonth,&from,&to));
%put x=&x;
```

Here are the results from the code:

```
x=2
```

Custom Time Intervals

A custom time interval is defined by a SAS data set. The data set must contain the *begin* variable; it can also contain the *end* and *season* variables. Each observation represents one interval with the *begin* variable containing the start of the interval, and the *end* variable, if present, containing the end of the interval. The intervals must be listed in ascending order. There cannot be gaps between intervals, and intervals cannot overlap.

You must properly define the *END* variable to avoid gaps. Gaps in the data can cause errors in processing. If the END variable is not specified, the only requirement to avoid gaps and overlaps is that the observations be sorted in ascending order with respect to the BEGIN variable. If the END variable is specified, it must be properly defined to avoid gaps and overlaps.

Overlap
: The value of the END variable for observation $n(END_n)$ exceeds $BEGIN_k$ for $k>n$.

Gap
: $BEGIN\ n+1 - END_n > 1$

Ascending
BEGIN$_n$ ≤ *END*$_n$ < *BEGIN*$_{n+1}$

The SAS system option INTERVALDS= is used to define custom intervals and associate interval data sets with new interval names. The following example shows how to specify the INTERVALDS= system option:

```
options intervalds=(interval=libref.dataset-name);
```

Arguments

interval
specifies the name of an interval. The value of *interval* is the data set that is named in *libref.dataset-name*.

libref.dataset-name
specifies the libref and data set name of the file that contains user-supplied holidays.

For more information, see "Custom Time Intervals" on page 43.

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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench* .

Examples

Example 1: Interval Examples Using INTCK

```
data one;
   qtr=intck('qtr', '10jan2021'd, '01jul2021'd);
   put qtr=;

   year=intck('year', '31dec2020'd, '01jan2021'd);
   put year=;

   year=intck('year', '01jan2021'd, '31dec2021'd);
   put year=;

   semi=intck('semiyear', '01jan2018'd, '01jan2021'd);
   put semi=;

   weekvar=intck('week2.2', '07jan2021'd, '01apr2021'd);
   put weekvar=;

   wdvar=intck('weekday7w', '01jan2013'd, '01feb2013'd);
   put wdvar=;

   y='year';
```

```
   date1='01sep2003'd;
   date2='01sep2013'd;
   newyears=intck(y, date1, date2);
   put newyears=;

   y=trim('year ');
   date1='1sep2005'd + 300;
   date2='1sep2013'd - 300;
   newyears=intck(y, date1, date2);
   put newyears=;
run;
```

These statements produce these results:

```
qtr=2
year=1
year=0
semi=6
weekvar=6
wdvar=27
newyears=10
newyears=6
```

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 is returned 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 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.

Example 2: An Example That Compares Methods

The following example shows different values for *method*:

```
data a;
   interval='month';
   start='14FEB2013'd;
   end='13MAR2013'd;
   months_default=intck(interval, start, end);
   months_discrete=intck(interval, start, end, 'd');
   months_continuous=intck(interval, start, end, 'c');
   output;

   end='14MAR2013'd;
   months_default=intck(interval, start, end);
```

```
   months_discrete=intck(interval, start, end, 'd');
   months_continuous=intck(interval, start, end, 'c');
   output;

   start='31JAN2013'd;
   end='01FEB2013'd;
   months_default=intck(interval, start, end);
   months_discrete=intck(interval, start, end, 'd');
   months_continuous=intck(interval, start, end, 'c');
   output;
   format start end date9.;
run;

proc print data=a;
run;
```

These statements produce these results:

Output 3.39 *Comparisons among Methods*

The SAS System

Obs	interval	start	end	months_default	months_discrete	months_continuous
1	month	14FEB2013	13MAR2013	1	1	0
2	month	14FEB2013	14MAR2013	1	1	1
3	month	31JAN2013	01FEB2013	1	1	0

Example 3

This example uses INTCK to calculate age:

```
data actual_dates;
input raw_date date9.;
format raw_date date9.;
datalines;
21JAN2019
03FEB2019
12MAR2019
29APR2019
02MAY2019
06JUN2019
11JUL2019
29AUG2019
25SEP2019
11NOV2019
16DEC2019
05JAN2020
;

data baby_ages(rename=(raw_date=birthday));
     set actual_dates;
```

```
      *today = today();
      today = '19DEC2019'D;
      months_old = INTCK('MONTH',raw_date,today,'Cont');
      if months_old < 0 then
         months_old = -(INTCK('MONTH',today,raw_date,'Cont'));
      days_old = today - INTNX('MONTH',raw_date,months_old,'Same');
      format today DATE.;
run;
proc print data=baby_ages noobs;
      var birthday today months_old days_old;
run;
```

Here is the output for the code:

birthday	today	months_old	days_old
21JAN2019	19DEC19	10	28
03FEB2019	19DEC19	10	16
12MAR2019	19DEC19	9	7
29APR2019	19DEC19	7	20
02MAY2019	19DEC19	7	17
06JUN2019	19DEC19	6	13
11JUL2019	19DEC19	5	8
29AUG2019	19DEC19	3	20
25SEP2019	19DEC19	2	24
11NOV2019	19DEC19	1	8
16DEC2019	19DEC19	0	3
05JAN2020	19DEC19	0	-17

See Also

Functions:

- “INTNX Function” on page 1004

System Options:

- “INTERVALDS= System Option” in *SAS System Options for SAS Viya Workbench*

INTCYCLE Function

Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.

Category: Date and Time

Syntax

INTCYCLE(*interval* <<*multiple*.<*shift-index*>>>, <*seasonality*>)

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The general form of an interval name is as follows:

interval<multiple.shift-index>

The three parts of the interval name are listed below:

interval

specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

Optional Argument

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.

Example In the following example, the function
`INTCYCLE('MONTH', 3);`
has a *seasonality* argument and returns the value QTR. The function
`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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for information about multipliers and shift indexes. See "Commonly Used Time Intervals" on page 41 for information about how intervals are calculated.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 40.

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" on page 43.

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 FORCAST, see the *SAS/ETS User's Guide*.

Example

```
data
one;


cycle_year=intcycle('year');

   put
cycle_year=;




cycle_quarter=intcycle('qtr');

   put
cycle_quarter=;



   cycle_3=intcycle('month',
3);

   put
cycle_3=;




cycle_month=intcycle('month');

   put
cycle_month=;



cycle_weekday=intcycle('weekday');

   put
cycle_weekday=;



   cycle_weekday2=intcycle('weekday',
5);
```

```
   put
cycle_weekday2=;



cycle_day=intcycle('day');

   put
cycle_day=;



   cycle_day2=intcycle('day',
10);

   put
cycle_day2=;



var1='second';


cycle_second=intcycle(var1);

   put
cycle_second=;

run;
```

These statements produce these results:

```
cycle_year=YEAR
cycle_quarter=YEAR
cycle_3=QTR
cycle_month=YEAR
cycle_weekday=WEEK
cycle_weekday2=WEEK
cycle_day=WEEK
cycle_day2=TENDAY
cycle_second=DTMINUTE
```

See Also

Functions:

- “INTCINDEX Function” on page 965
- “INTINDEX Function” on page 993
- “INTSEAS Function” on page 1013

Other References:

- *SAS/ETS User's Guide*

INTFIT Function

Returns a time interval that is aligned between two dates.

Category: Date and Time

Syntax

INTFIT(*argument-1*, *argument-2*, '*type*')

Required Arguments

argument
specifies a SAS expression that represents a SAS date or datetime value, or an observation.

Tip Observation numbers are more likely to be used as arguments if date or datetime values are not available.

'type'
specifies whether the arguments are SAS date values, datetime values, or observations.

The following values for *type* are valid:

d specifies that *argument-1* and *argument-2* are date values.

dt specifies that *argument-1* and *argument-2* are datetime values.

obs specifies that *argument-1* and *argument-2* are observations.

Details

The INTFIT function returns the most likely time interval based on two dates, datetime values, or observations 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" on page 1004.

If the arguments that are used with INTFIT are observations, you can determine the cycle of an occurrence by using observation numbers. In the following example, the first two arguments of INTFIT are observation 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 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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Examples

Example 1: Finding Intervals That Are Aligned between Two Dates

The following example shows the intervals that are aligned between two dates. The *type* argument in this example identifies the input as date values.

```
data a;
   length interval $20;
   date1='01jan21'd;
   do i=1 to 25;
      date2=intnx('day', date1, i);
      interval=intfit(date1, date2, 'd');
      output;
   end;
   format date1 date2 date.;
run;
proc print data=a noobs;
run;
```

Output 3.40 *Interval Output from the INTFIT Function*

Obs	interval	date1	i	date2
1	DAY	01JAN21	1	02JAN21
2	DAY2.2	01JAN21	2	03JAN21
3	WEEKDAY	01JAN21	3	04JAN21
4	DAY4.2	01JAN21	4	05JAN21
5	DAY5.2	01JAN21	5	06JAN21
6	DAY6.4	01JAN21	6	07JAN21
7	WEEK.6	01JAN21	7	08JAN21
8	DAY8.2	01JAN21	8	09JAN21
9	DAY9.7	01JAN21	9	10JAN21
10	TENDAY	01JAN21	10	11JAN21
11	DAY11.7	01JAN21	11	12JAN21
12	DAY12.10	01JAN21	12	13JAN21
13	DAY13.13	01JAN21	13	14JAN21
14	WEEK2.13	01JAN21	14	15JAN21
15	SEMIMONTH	01JAN21	15	16JAN21
16	DAY16.10	01JAN21	16	17JAN21
17	DAY17.12	01JAN21	17	18JAN21
18	DAY18.16	01JAN21	18	19JAN21
19	DAY19.14	01JAN21	19	20JAN21
20	TENDAY2	01JAN21	20	21JAN21
21	WEEK3.6	01JAN21	21	22JAN21
22	DAY22.18	01JAN21	22	23JAN21
23	DAY23.18	01JAN21	23	24JAN21
24	DAY24.10	01JAN21	24	25JAN21
25	DAY25.7	01JAN21	25	26JAN21

The output shows that if the increment value is one day, then the result of the INTFIT function is DAY. If the increment value is two days, then the result of the INTFIT function is DAY2. If the increment value is three days, then the result is DAY3.2, with a shift index of 3. (If the two input dates are a Friday and a Monday, then the result is WEEKDAY.) If the increment value is seven days, then the result is WEEK.

Example 2: Finding Intervals That Are Aligned between Two Dates When the Dates Are Identified as Observations

The following example shows the intervals that are aligned between two dates. The *type* argument in this example identifies the input as observations.

```
data a;
   length interval $20;
   date1='01jan21'd;
   do i=1 to 25;
      date2=intnx('day', date1, i);
```

```
        interval=intfit(date1, date2, 'obs');
        output;
    end;
    format date1 date2 date.;
run;
proc print data=a;
run;
```

Output 3.41 *Interval Output from the INTFIT Function When Dates Are Identified as Observations*

Obs	interval	date1	i	date2
1	OBS	01JAN21	1	02JAN21
2	OBS2.2	01JAN21	2	03JAN21
3	OBS3	01JAN21	3	04JAN21
4	OBS4.2	01JAN21	4	05JAN21
5	OBS5.2	01JAN21	5	06JAN21
6	OBS6.4	01JAN21	6	07JAN21
7	OBS7	01JAN21	7	08JAN21
8	OBS8.2	01JAN21	8	09JAN21
9	OBS9.7	01JAN21	9	10JAN21
10	OBS10.2	01JAN21	10	11JAN21
11	OBS11.7	01JAN21	11	12JAN21
12	OBS12.10	01JAN21	12	13JAN21
13	OBS13.13	01JAN21	13	14JAN21
14	OBS14.8	01JAN21	14	15JAN21
15	OBS15.7	01JAN21	15	16JAN21
16	OBS16.10	01JAN21	16	17JAN21
17	OBS17.12	01JAN21	17	18JAN21
18	OBS18.16	01JAN21	18	19JAN21
19	OBS19.14	01JAN21	19	20JAN21
20	OBS20.2	01JAN21	20	21JAN21
21	OBS21	01JAN21	21	22JAN21
22	OBS22.18	01JAN21	22	23JAN21
23	OBS23.18	01JAN21	23	24JAN21
24	OBS24.10	01JAN21	24	25JAN21
25	OBS25.7	01JAN21	25	26JAN21

See Also

Functions:

- “INTCK Function” on page 968
- “INTNX Function” on page 1004

INTFMT Function

Returns a recommended SAS format when a date, time, or datetime interval is specified.

Category: Date and Time

Syntax

INTFMT(*interval* <<*multiple*.<.*shift-index*>>>, '*size*')

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name such as DTYEARV, WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench* .

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>

The three parts of the interval name are as follows:

interval

specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

'size'
specifies either LONG or SHORT. When a format includes a year value, LONG or L specifies a format that uses a four-digit year. SHORT or S specifies a format that uses a two-digit year.

Details

The INTFMT function returns a recommended format depending on a date, time, or datetime interval for displaying the time ID values that are associated with a time series of a given interval. The valid values of SIZE (LONG, L, SHORT, or S) specify whether to use a two-digit or a four-digit year when the format refers to a SAS date value. For more information about working with date and time intervals, see "Date and Time Intervals" on page 40.

The INTFMT 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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Examples

Example 1

```
data
one;

 fmt1=intfmt('qtr',
's');

 put
fmt1=;
```

```
 fmt2=intfmt('qtr',
'l');

 put
fmt2=;



 fmt3=intfmt('month',
'l');

 put
fmt3=;



 fmt4=intfmt('week',
'short');

 put
fmt4=;



 fmt5=intfmt('week3.2',
'l');

 put
fmt5=;



 fmt6=intfmt('day',
'long');

 put
fmt6=;




var1='month2';

 fmt7=intfmt(var1,
'long');

 put
fmt7=;

run;
```

These statements produce these results:

```
fmt1=YYQC4.
fmt2=YYQC6.
fmt3=MONYY7.
fmt4=WEEKDATX15.
fmt5=WEEKDATX17.
fmt6=DATE9.
fmt7=MONYY7.
```

Example 2

You can also display date and datetime values as strings by using the format that is identified by the INTFMT function. In the following example, INTFIT identifies the intervals of the sashelp.citiwk and sashelp.air data sets. Then INTFMT identifies formats that are based on the intervals. The formats convert the first SAS date value of each data set to a string. The START variable displays the date of the first observation of each data set. This method assumes that the interval of the data set can be identified by examining the first two observations. You can use the TIMESERIES procedure to prepare the data sets for input. More than two observations might be required to identify the difference between a DAY interval and a WEEKDAY interval. This example would need to be modified if the DATE variable contained SAS datetime values.

```
data a(keep=date0 date interval fmt start);
   length start interval fmt $32;
   format date0 date date.;
   set sashelp.air(obs=2) sashelp.citiwk(obs=2);
   date = lag(date);
   if (mod(_n_,2) eq 1) then delete;
   if (mod(_n_,2) eq 0) then interval = intfit( date0, date, 'D' );
   if (mod(_n_,2) eq 0) then fmt = intfmt( interval, 'l' );
   start = putn( date0, fmt );
run;
proc print;
run;
```

SAS creates the following output:

Output 3.42 *Displaying DATE and DATETIME Values as Strings*

Obs	START	INTERVAL	FMT	date0	date
1	JAN1949	MONTH	MONYY7.	01JAN49	01FEB49
2	Sun, 22 Dec 1985	WEEK	WEEKDATX17.	22DEC85	29DEC85

INTGET Function

Returns a time interval based on three date or datetime values.

Category: Date and Time

Syntax

INTGET(*date-1*, *date-2*, *date-3*)

Required Argument

date
specifies a SAS date or datetime value.

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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench* .

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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench* .

Example

```
data
one;

   interval1=intget('01jan00'd, '01jan01'd,
'01may01'd);

   put
interval1=;



   interval2=intget('29feb80'd, '28feb82'd,
'29feb84'd);

   put
interval2=;



   interval3=intget('01feb80'd, '16feb80'd,
'01mar80'd);

   put
interval3=;



   interval4=intget('2jan09'd, '2feb10'd,
'2mar11'd);

   put
interval4=;



   interval5=intget('10feb80'd, '19feb80'd,
'28feb80'd);

   put
interval5=;
```

```
   interval6=intget('01apr2006:00:01:02'dt, '01apr2006:00:02:02'dt,

'01apr2006:00:03:02'dt);
   put
interval6=;

run;
```

These statements produce these results:

```
interval1=MONTH4
interval2=YEAR2.2
interval3=SEMIMONTH
interval4=MONTH13.4
interval5=DAY9.2
interval6=MINUTE
```

See Also

Functions:

- “INTFIT Function” on page 983
- “INTNX Function” on page 1004

INTINDEX Function

Returns the seasonal index when a date, time, or datetime interval and value are specified.

Category: Date and Time

Syntax

INTINDEX(*interval* <<*multiple*.<*shift-index*>>>, *date-value*, <*seasonality*>)

Required Arguments

interval
specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in “About Date and Time Intervals” in *SAS Formats and Informats for SAS Viya Workbench*.

TIP If *interval* is a character constant, then enclose the value in quotation marks.

TIP Valid values for *interval* depend on whether *date-value* is a date is a date, time, or datetime value. For more information, see "Commonly Used Time Intervals" on page 41.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is listed below:

interval<multiple.shift-index>

The three parts of the interval name are as follows:

interval

specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

date-value
specifies a date, time, or datetime value that represents a time period of the given interval.

Optional Argument

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.

Example In this example, the following functions produce the same result.

```
INTINDEX('MONTH', date, 3);
INTINDEX('MONTH', date, 'QTR');
```

Seasonality in the first example is a number (the number of months), and in the second example *seasonality* is a cycle (QTR).

Details

INTINDEX Function Intervals

The INTINDEX function returns the seasonal index when you supply an interval and an appropriate date, time, or datetime value. The seasonal index is a number that represents the position of the date, time, or datetime 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.

```
intindex('month', '01DEC2012'd);
```

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.

```
intindex('qtr', '01JAN2013'd);
intindex('qtr', '31MAR2013'd);
```

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.

```
intindex('day', '07DEC2012'd);
```

How *Interval* and *Date-Time-Value* Are Related

To correctly identify the seasonal index, the interval should agree with the date, time, or datetime 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.

```
intindex('day', '07DEC2012'd);
```

The DAY interval requires a SAS date value.

This example returns a missing value because the QTR intervals expects the date to be a SAS data value rather than a datetime value.

```
intindex('qtr', '01JAN2013:00:00:00'dt);
```

This example returns a value of 12. The DTMONTH interval requires a datetime value.

```
intindex('dtmonth', '01DEC2013:00:00:00'dt);
```

For more information about working with date and time intervals, see "Date and Time Intervals" on page 40.

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" on page 43.

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 datetime 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 a value of 5 because April 4, 2013 is a Thursday, the fifth day of the week.

```
index=intindex('day', '04APR2013'd);
```

Using the same date, the INTCINDEX function returns a value of 14 because April 4, 2013 is the fourteenth week of the year.

```
cycle_index=intcindex('day', '04APR2013'd);
```

In this example, the INTINDEX function returns the minute of the hour.

```
index = intindex('minute', '01Sep2012:06:05:04'dt);
```

Using the same date and time, the INTCINDEX function returns the hour of the day.

```
cycle_index = intcindex('minute', '01Sep2012:06:05:04'dt);
```

In the example `intseas('interval');`, INTSEAS returns the maximum number that could be returned by `intindex('interval', date);`.

Examples

Example 1: Examples of Using INTINDEX with Two Arguments

```
data
one;

   interval1=intindex('qtr',
'14AUG2013'd);

   put
interval1=;



   interval2=intindex('dtqtr',
'23DEC2013:15:09:19'dt);

   put
interval2=;



   interval3=intindex('hour',
'09:05:15't);

   put
interval3=;



   interval4=intindex('month',
'26FEB2013'd);

   put
interval4=;



   interval5=intindex('dtmonth',
'28MAY2013:05:15:00'dt);

   put
interval5=;
```

```
   interval6=intindex('week',
'09SEP2013'd);

   put
interval6=;



   interval7=intindex('tenday',
'16APR2013'd);

   put
interval7=;

run;
```

These statements produce these results:

```
interval1=3
interval2=4
interval3=10
interval4=2
interval5=5
interval6=37
interval7=11
```

Example 2: Examples of Using INTINDEX with Three Arguments

This example uses three arguments in the INTINDEX function. The example also uses PROC FCMP.

```
data a;
   do i = 0 to 12;
      begin = INTNX('QTR.2','01NOV2018'D,i);
      season = INTINDEX('QTR.2',begin,'YEAR.11');
      output;
   end;
   format begin DATE.;
run;
proc print;
run;
```

The SAS System

Obs	i	begin	season
1	0	01NOV18	1
2	1	01FEB19	2
3	2	01MAY19	3
4	3	01AUG19	4
5	4	01NOV19	1
6	5	01FEB20	2
7	6	01MAY20	3
8	7	01AUG20	4
9	8	01NOV20	1
10	9	01FEB21	2
11	10	01MAY21	3
12	11	01AUG21	4
13	12	01NOV21	1

```
proc fcmp outlib=work.myfncs.fmtfncs;
function custqtr(date) $ 6;
qtr = INTINDEX('QTR.2',date,'YEAR.11');
year = YEAR(INTNX('YEAR.11',date,0));
value = trim(left(put(year,4.)))||':'||trim(left(put(qtr,1.)));
return(value);
endsub;

options cmplib=work.myfncs;

proc format;
   value custqtr(default=6) other=[custqtr()];
run;

data a;
   do i = 0 to 12;
      begin = INTNX('QTR.2','01NOV2018'D,i);
      season = INTINDEX('QTR.2',begin,'YEAR.11');
      output;
   end;
   format begin CUSTQTR.;
run;
proc print; run;
```

The SAS System

Obs	i	begin	season
1	0	2018:1	1
2	1	2018:2	2
3	2	2018:3	3
4	3	2018:4	4
5	4	2019:1	1
6	5	2019:2	2
7	6	2019:3	3
8	7	2019:4	4
9	8	2020:1	1
10	9	2020:2	2
11	10	2020:3	3
12	11	2020:4	4
13	12	2021:1	1

Example 3: Example of Seasonality

SAS uses a default seasonal cycle. For example, the assumption is that monthly data is yearly seasonal. However, monthly data could also have a seasonal cycle of semiyearly. This example shows that to use a third argument, *seasonality*, enables you to specify the seasonality rather than using the default. It also shows how to handle leap years:

```
data weekly;
     *do year=2007 to 2012;
     year=2004;
        NewYear=holiday('NEWYEAR', year);
        do i=-5 to 5;
           date=intnx('week', NewYear, i);
           output;
        end;
     *end;
     format date date.;
            format NewYear date.;
run;

/* The standard leap week is the first week of year. */
/* An alternative method uses a third argument:leap week is week 53. */
title "Using a Third Argument to Control Weekly Seasonality";
data LeapWeekExample;
     set weekly;
     StandardIndex=intindex('week', date);
```

```
     IndexWithLeap=intindex('week', date, 53);
run;
proc print;
run;

/* Using a number and an interval can be equivalent for the third
argument. */
title "Using the Third Argument as a Number or Cycle";
data Equiv3rdArg;
     set sashelp.air(obs=12);
     defaultSeasonal=intindex('MONTH', date);
     SeasonalArg12=intindex('MONTH', date, 12);
     SeasonalArgYear=intindex('MONTH', date, 'YEAR');
     format date date.;
run;
proc print;
run;


/* Use the third argument for non-standard seasonality. */
title "Using the Third Argument for Non-Standard Seasonality";
data NonStandardSeasonal;
     set sashelp.air(obs=24);
     /* Standard Index - MONTH is Yearly Seasonal */
     StandardIndex=intindex('MONTH', date);
     SemiYrIndex=intindex('MONTH', date, 'SEMIYR');
     Index6=intindex('MONTH', date, 6);
     format date date.;
run;

proc print;
run;
```

Output 3.43 *Output from the Seasonality Example*

Using a Third Argument to Control Weekly Seasonality

Obs	year	NewYear	i	date	StandardIndex	IndexWithLeap
1	2004	01JAN04	-5	23NOV03	48	48
2	2004	01JAN04	-4	30NOV03	49	49
3	2004	01JAN04	-3	07DEC03	50	50
4	2004	01JAN04	-2	14DEC03	51	51
5	2004	01JAN04	-1	21DEC03	52	52
6	2004	01JAN04	0	28DEC03	1	53
7	2004	01JAN04	1	04JAN04	1	1
8	2004	01JAN04	2	11JAN04	2	2
9	2004	01JAN04	3	18JAN04	3	3
10	2004	01JAN04	4	25JAN04	4	4
11	2004	01JAN04	5	01FEB04	5	5

Using the Third Argument as a Number or Cycle

Obs	DATE	AIR	defaultSeasonal	SeasonalArg12	SeasonalArgYear
1	01JAN49	112	1	1	1
2	01FEB49	118	2	2	2
3	01MAR49	132	3	3	3
4	01APR49	129	4	4	4
5	01MAY49	121	5	5	5
6	01JUN49	135	6	6	6
7	01JUL49	148	7	7	7
8	01AUG49	148	8	8	8
9	01SEP49	136	9	9	9
10	01OCT49	119	10	10	10
11	01NOV49	104	11	11	11
12	01DEC49	118	12	12	12

Using the Third Argument for Non-Standard Seasonality

Obs	DATE	AIR	StandardIndex	SemiYrIndex	Index6
1	01JAN49	112	1	1	1
2	01FEB49	118	2	2	2
3	01MAR49	132	3	3	3
4	01APR49	129	4	4	4
5	01MAY49	121	5	5	5
6	01JUN49	135	6	6	6
7	01JUL49	148	7	1	1
8	01AUG49	148	8	2	2
9	01SEP49	136	9	3	3
10	01OCT49	119	10	4	4
11	01NOV49	104	11	5	5
12	01DEC49	118	12	6	6
13	01JAN50	115	1	1	1
14	01FEB50	126	2	2	2
15	01MAR50	141	3	3	3
16	01APR50	135	4	4	4
17	01MAY50	125	5	5	5
18	01JUN50	149	6	6	6
19	01JUL50	170	7	1	1
20	01AUG50	170	8	2	2
21	01SEP50	158	9	3	3
22	01OCT50	133	10	4	4
23	01NOV50	114	11	5	5
24	01DEC50	140	12	6	6

See Also

Functions:

- “INTCINDEX Function” on page 965
- “INTCYCLE Function” on page 978
- “INTSEAS Function” on page 1013

Other References:

- *SAS/ETS User's Guide*

INTNX Function

Increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value.

Category: Date and Time

Syntax

INTNX(*interval* <*multiple*><*.shift-index*>, *start-from*, *increment* <, '*alignment*'>)

INTNX(*custom-interval*, *start-from*, *increment* <, '*alignment*'>)

Required Arguments

interval

specifies a character constant, variable, or expression that contains a time interval such as WEEK, SEMIYEAR, QTR, or HOUR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in the "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

> **TIP** The type of interval (date, datetime, or time) must match the type of value in *start-from*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The general form of an interval name is as follows:

interval<multiple.shift-index>

The three parts of the interval name are listed below:

interval

specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

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, MONTH type intervals shift by MONTH subperiods by default. Thus, 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. The interval name MONTH2.2, for example, specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

start-from
specifies a SAS expression that represents a SAS date, time, or datetime value that identifies a starting point.

increment
specifies a negative, positive, or zero integer that represents the number of date, time, or datetime intervals. *Increment* is the number of intervals to shift the value of *start-from*.

Optional Arguments

'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 1007 for more information.

Default BEGINNING

See "Aligning SAS Date Output within Its Intervals" on page 1007 for more information.

custom-interval
specifies an interval that you define.

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 SAS date value to a calendar date, use any valid SAS 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 March 17, 2021.

```
x=intnx('week', '17mar21'd, 6);
put x date9.;
```

INTNX returns the value 25APR2021.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 40.

Date and Datetime Intervals

The intervals that you need to use with SAS datetime values are SAS datetime intervals. Datetime intervals are formed by adding the prefix "DT" to any date interval. For example, MONTH is a SAS date interval, and DTMONTH is a SAS datetime interval. Similarly, YEAR is a SAS date interval, and DTYEAR is a SAS datetime interval.

To ensure correct results with interval functions, use date intervals with date values and datetime intervals with datetime values. SAS does not return an error

message if you use a date value with a datetime interval, but the results are incorrect:

```
data _null_;
      /* The following statement creates expected results. */
   date1=intnx('dtday', '01aug21:00:10:48'dt, 1);
      /* The following two statements create unexpected results. */
   date2=intnx('dtday', '01aug21'd, 1);
   date3=intnx('dtday', '01aug21:00:10:48'd, 1);
   put 'Correct Datetime Value   ' date1= datetime19. /
       'Incorrect Datetime Value ' date2= datetime19. /
       'Incorrect Datetime Value ' date3= datetime19.;
run;
```

SAS writes the following output to the log:

```
Correct Datetime Value date1=02AUG2021:00:00:00
Incorrect Datetime Value date2=02JAN1960:00:00:00
Incorrect Datetime Value date3=02JAN1960:00:00:00
```

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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

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. When you use SAME alignment with YEAR, for example, 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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench* .

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:

```
intnx('week', '15mar2000'd, 1, 'same');   returns 22MAR2000
intnx('dtweek','15mar2000:8:45'dt, 1, 'same');   returns
22MAR2000:08:45:00
intnx('year', '15mar2000'd, 5, 'same');   returns 15MAR2005
```

Adjusting Dates

The INTNX function automatically adjusts for the date if the date in the interval that is incremented does not exist. For example:

```
intnx('month', '15mar2000'd, 5, 'same');  returns 15AUG2000
intnx('year', '29feb2000'd, 2, 'same');   returns 28FEB2002
intnx('month', '31aug2001'd, 1, 'same');  returns 30SEP2001
intnx('year', '01mar1999'd, 1, 'same');   returns 01MAR2000 (the first
day of the 3rd month of the year)
```

In the example `intnx('year', '29feb2000'd, 2);`, the INTNX function returns the value 01JAN2002, which is the beginning of the year two years from the starting date (2000).

In the example `intnx('year', '29feb2000'd, 2, 'same');`, the INTNX function returns the value 28FEB2002. In this case, the starting date begins in the year 2000, the year is two years later (2002), the month is the same (February), and the date is the 28th, because that is the closest date to the 29th in February 2002.

Custom Intervals

A custom interval is defined by a SAS data set. The data set must contain the *begin* variable, and it can also contain the *end* and *season* variables. Each observation represents one interval with the *begin* variable containing the start of the interval, and the *end* variable, if present, containing the end of the interval. The intervals must be listed in ascending order. You cannot have gaps between intervals, and intervals cannot overlap.

The SAS system option INTERVALDS= is used to define custom intervals and associate interval data sets with new interval names. The following example shows how to specify the INTERVALDS= system option:

```
options intervalds=(interval=libref.dataset-name);
```

Argument

interval
specifies the name of an interval. The value of *interval* is the data set that is named in *libref.dataset-name*.

libref.dataset-name
specifies the libref and data set name of the file that contains user-supplied holidays.

For more information, see “Custom Time Intervals” on page 43.

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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Examples

Example 1

```
data one;
   yr=intnx('year', '05feb94'd, 3);
   put yr / yr date7.;

   x=intnx('month', '05jan95'd, 0);
   put x / x date7.;

  next=intnx('semiyear', '01jan97'd, 1);
   put next / next date7.;

   past=intnx('month2', '01aug96'd, -1);
   put past / past date7.;

   sm=intnx('semimonth2.2', '01apr97'd, 4);
   put sm / sm date7.;

   y='month';
   date='1jun1990'd;
   nextmon=intnx(y, date, 1);
   put nextmon / nextmon date7.;

   x1='month';
   x2=trim(x1);
   date='1jun1990'd - 100;
  nextmonth=intnx(x2, date, 1);
   put nextmonth / nextmonth date7.;

/* The following examples show the results of advancing a date by
using the optional alignment argument. */

   date1=intnx('month', '01jan95'd, 5, 'beginning');
   put date1 / date1 date7.;

   date2=intnx('month', '01jan95'd, 5, 'middle');
   put date2 / date2 date7.;

   date3=intnx('month', '01jan95'd, 5, 'end');
run;
```

These statements produce these results:

```
13515
01JAN97
12784
01JAN95
13696
01JUL97
13270
01MAY96
13711
16JUL97
11139
01JUL90
11017
01MAR90
12935
01JUN95
12949
15JUN95
```

Example 2: Example of Using Custom Intervals

The following example uses the *custom-interval* form of the INTNX function to increment a date, time, or datetime value by a given time interval.

```
options intervalds=(weekdaycust=dstest);
data dstest;
   format begin end date9.;
   begin='01jan2008'd; end='01jan2008'd; output;
   begin='02jan2008'd; end='02jan2008'd; output;
   begin='03jan2008'd; end='03jan2008'd; output;
   begin='04jan2008'd; end='06jan2008'd; output;
   begin='07jan2008'd; end='07jan2008'd; output;
   begin='08jan2008'd; end='08jan2008'd; output;
   begin='09jan2008'd; end='09jan2008'd; output;
   begin='10jan2008'd; end='10jan2008'd; output;
   begin='11jan2008'd; end='13jan2008'd; output;
   begin='14jan2008'd; end='14jan2008'd; output;
   begin='15jan2008'd; end='15jan2008'd; output;
run;

data _null_;
   format start date9. endcustom date9.;
   start='01jan2008'd;
   do i=0 to 9;
      endcustom=intnx('weekdaycust', start, i);
      put endcustom;
   end;
run;
```

These statements produce these results:

```
01JAN2008
02JAN2008
03JAN2008
04JAN2008
07JAN2008
08JAN2008
09JAN2008
10JAN2008
11JAN2008
14JAN2008
```

See Also

Functions:

- "INTCK Function" on page 968
- "INTSHIFT Function" on page 1016

System Options:

- "INTERVALDS= System Option" in *SAS System Options for SAS Viya Workbench*

INTRR Function

Returns the internal rate of return as a fraction.

Category: Financial

Syntax

INTRR(*frequency*, *c0*, *c1*, ..., *cn*)

Required Arguments

frequency
is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *frequency* > 0

Tip The case *frequency* = 0 is a flag to allow continuous compounding.

c0, c1, ..., cn
are numeric, the optional cash payments.

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 *frequency* > 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 "NETPV Function" on page 1134.) It is given by

$$r = \begin{cases} \frac{1}{x^{freq}} - 1 & freq > 0 \\ -log_{\varepsilon}(x) & freq = 0 \end{cases}$$

where *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 *frequency* > 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 *frequency* to 3.

If *frequency* 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 *frequency* to 0. The computed internal rate of return will be 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 expressed with these statements:

```
data
one;
```

```
   rate=intrr(1, -400, 100, 200,
300);

   put
rate=;

run;
```

These statements produce this result:

```
rate=0.1943770996
```

See Also

Functions:

- "IRR Function" on page 1026

INTSEAS Function

Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.

Category: Date and Time

Syntax

INTSEAS(*interval* <*multiple*.<*shift-index*>> <, *seasonality*>)

Required Argument

interval
specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>
The three parts of the interval name are as follows:

interval
specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

Optional Argument

seasonality
specifies a number or a cycle.

This argument enables you to have more flexibility in working with dates and time cycles. If there is a 53-week year, you can easily determine the seasonality by using 53 as the value for *seasonality*, as the following example shows: `INTSEAS('WEEK', 53);`. By default, `INTSEAS('WEEK');` equals 52.

Example The function
```
INTSEAS('interval', seasonality);
```
returns a number when you specify a numeric value for *seasonality*. The function
```
INTSEAS('MONTH', 'QTR');
```
returns a value of 3 when you specify the QTR cycle.

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" on page 40.

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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

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

```
data one;
   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=;
run;
```

These statements produce these results:

```
cycle_years=1
cycle_smiyears=2
cycle_quarters=4
cycle_number=3
cycle_months=12
cycle_smimonths=24
cycle_tendays=36
cycle_weeks=52
cycle_wkdays=5
cycle_hours=24
cycle_minutes=60
cycle_month2=6
```

See Also

Functions:

- “INTCYCLE Function” on page 978
- “INTINDEX Function” on page 993

Other References:

- *SAS/ETS User's Guide*

INTSHIFT Function

Returns the shift interval that corresponds to the base interval.

Category: Date and Time

Syntax

INTSHIFT(*interval* <<*multiple*.<*shift-index*>>>)

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains a time interval such as WEEK, SEMIYEAR, QTR, or HOUR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The general form of an interval name is as follows:

interval<multiple.shift-index>

The three parts of the interval name are as follows:

interval

specifies the name of the basic interval type. For 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. For example, the interval YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For 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.

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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

Details

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 "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Example

```
data one;
   shift1=intshift('year');
   put shift1=;
   shift2=intshift('dtyear');
   put shift2=;
   shift3=intshift('minute');
   put shift3=;
   interval='weekdays';
   shift4 = intshift(interval);
   put shift4=;
   shift5=intshift('weekday5.4');
   put shift5=;
   shift6=intshift('qtr');
   put shift6=;
   shift7=intshift('dttenday');
   put shift7=;
run;
```

These statements produce these results:

```
shift1=MONTH
shift2=DTMONTH
shift3=DTMINUTE
shift4=WEEKDAY
shift5=WEEKDAY
shift6=MONTH
shift7=DTTENDAY
```

INTTEST Function

Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.

Category: Date and Time

Syntax

INTTEST(*interval* <<*multiple*.<*shift-index*>>>)

Required Argument

interval

specifies a character constant, variable, or expression that contains an interval name, such as WEEK, MONTH, or QTR. *interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>

Here are the three parts of the interval name:

interval

specifies the name of the basic interval type. For 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. For example, YEAR2 consists of two-year, or biennial, periods.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point. For example, YEAR.3 specifies yearly periods that are shifted to start on the first of March of each calendar year and to end in February of the following year.

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 is invalid 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.

See "Incrementing Dates and Times By Using Multipliers and By Shifting Intervals" on page 40 for more information.

Details

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, see "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

The INTTEST function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "About Date and Time Intervals" in *SAS Formats and Informats for SAS Viya Workbench*.

Example

If *interval* is valid, 1 is returned, and if *interval* is invalid, 0 is returned.

```
data one;
   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=;
run;
```

These statements produce these results:

```
test1=1
test2=1
test3=1
test4=0
test5=1
```

INTZ Function

Returns the integer portion of the argument, using zero fuzzing.

Category: Rounding and Truncation

Syntax

INTZ(*argument*)

Required Argument

argument
is a numeric constant, variable, or expression.

Details

The following rules apply:

- If the value of the argument is an exact integer, INTZ returns that integer.
- If the argument is positive and not an integer, INTZ returns the largest integer that is less than the argument.
- If the argument is negative and not an integer, INTZ returns the smallest integer 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 an integer, the INT function fuzzes the result to be equal to that integer. The INTZ function does not fuzz the result. Therefore, with the INTZ function you might get unexpected results.

Example

This example demonstrates the INT and INTZ functions.

```
data _null_;
   one      = 1;
   int_one  = int(one);
   intz_one = intz(one);

   one_minus_delta      = 1 - 1.e-12;
   int_one_minus_delta  = int(one_minus_delta);
   intz_one_minus_delta = intz(one_minus_delta);

   put 'INFO: ' one= int_one= intz_one=;
   put 'INFO: ' one_minus_delta= int_one_minus_delta=
intz_one_minus_delta=;
run;
```

These statements produce these results:

```
INFO: one=1 int_one=1 intz_one=1
INFO: one_minus_delta=1 int_one_minus_delta=1 intz_one_minus_delta=0
```

See Also

Functions:

- “CEIL Function” on page 424
- “CEILZ Function” on page 426
- “FLOOR Function” on page 710
- “FLOORZ Function” on page 712
- “INT Function” on page 963
- “ROUND Function” on page 1348
- “ROUNDZ Function” on page 1359

IORCMSG Function

Returns a formatted error message for _IORC_.

Category: SAS File I/O

Syntax

IORCMSG()

Details

If the IORCMSG function returns a value to a variable that has not yet been assigned a length, then by default the variable is assigned a length of 200.

The IORCMSG function returns the formatted error message that is associated with the current value of the automatic variable _IORC_. The _IORC_ variable is created when you use the MODIFY statement, or when you use the SET statement with the KEY= option. The value of the _IORC_ variable is internal and is meant to be read in conjunction with the SYSRC autocall macro. If you try to set _IORC_ to a specific value, you might get unexpected results.

Example

In the following program, observations are either rewritten or added to the updated master file that contains bank accounts and current bank balance. The program queries the _IORC_ variable and returns a formatted error message if the _IORC_ value is unexpected.

```
libname bank 'SAS-library';
data bank.master(index=(AccountNum));
   infile 'external-file-1';
   format balance dollar8.;
   input @ 1 AccountNum $ 1-3 @ 5 balance 5-9;
run;
data bank.trans(index=(AccountNum));
   infile 'external-file-2';
   format deposit dollar8.;
   input @ 1 AccountNum $ 1-3 @ 5 deposit 5-9;
run;
data bank.master;
   set bank.trans;
   modify bank.master key=AccountNum;
   if (_IORC_ EQ %sysrc(_SOK)) then
      do;
         balance=balance+deposit;
         replace;
      end;
else
   if (_IORC_= %sysrc(_DSENOM)) then
      do;
         balance=deposit;
         output;
         _error_=0;
      end;
else
   do;
      errmsg=IORCMSG();
      put 'Unknown error condition:'
      errmsg;
   end;
run;
```

IPMT Function

Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.

Category: Financial

Syntax

IPMT(*rate*, *period*, *number-of-periods*, *principal-amount*, <*future-amount*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

period
specifies the payment period for which the interest payment is computed.

Requirement *Period* must be a positive integer value that is less than or equal to the value of *number-of-periods*.

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

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.

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.

Example

```
data
one;



 /*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 with these statements:
*/
```

```
   InterestPaid1=IPMT(0.1/12, 1, 36,
8000);



 /*If the same loan has beginning-of-period payments, then the
interest payment
can be computed with these
statements:*/



   InterestPaid2=IPMT(0.1/12, 1, 36, 8000, 0,
1);

   InterestPaid3=IPMT(0.1, 3, 3,
8000);

   InterestPaid4=IPMT(0.09/12, 359, 360, 125000, 0,
1);



   put
_all_;



run;
```

These statements produce these results:

```
InterestPaid1=66.666666667 InterestPaid2=0 InterestPaid3=292.44712991
InterestPaid4=14.807573663
_ERROR_=0 _N_=1
```

IQR Function

Returns the interquartile range.

Category: Descriptive Statistics

Syntax

IQR(*value-1* <, *value-2*..., *value-n*>)

Required Argument

value
specifies a numeric constant, variable, or expression for which the interquartile range is to be computed.

Details

If all arguments have missing values, the result is a missing value. Otherwise, the result is the interquartile range of the nonmissing values. The formula for the interquartile range is the same as the one that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide: Statistical Procedures.

Example

```
data
one;

   iqr=iqr(2, 4, 1, 3,
999999);

   put
iqr=;

run;
```

These statements produce this result:

```
iqr=2
```

See Also

Functions:

- “MAD Function” on page 1093
- “PCTL Function” on page 1190

IRR Function

Returns the internal rate of return as a percentage.

Category: Financial

Syntax

IRR(*frequency , c1, c2*<, ... , *cn*>)

Required Arguments

frequency
is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *frequency* > 0.

Tip The case *frequency* = 0 is a flag to allow continuous compounding.

c1, c2, ..., cn
are numeric, the optional cash payments.

Requirement A minimum of two cash payment values are required.

Details

The IRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c1*, ..., *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument *frequency* > 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 the expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return as a percentage can be expressed with these statements:

```
data
one;

   rate=irr(1, -400, 100, 200,
300);
```

```
   put
rate=;

run;
```

These statements produce this result:

```
rate=19.437709963
```

See Also

Functions:

- “INTRR Function” on page 1011

JBESSEL Function

Returns the value of the Bessel function.

Category: Mathematical

Syntax

JBESSEL(*nu*, *x*)

Required Arguments

nu

specifies a numeric constant, variable, or expression.

Range *nu* ≥ 0

x

specifies a numeric constant, variable, or expression.

Range *x* ≥ 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

```
data
one;

   x=jbessel(2,
2);

   put
x=;

run;
```

These statements produce this result:

```
x=0.3528340286
```

JSONPP Function

Creates a readable copy of JSON input.

Syntax

JSONPP('*input file*','*output file*')

Required Arguments

input file
specifies the JSON input file. The *input file* can be a physical name or a fileref.

output file
specifies the readable JSON file. The *output file* can be a physical name or a fileref. If the *output file* is LOG, then the output is written to the SAS log.

Details

JSONPP stands for JSON Prettyprint.

Example

This example reads the disk file test.json and creates a disk file test.json.pp.

```
data _null_;
```

```
   rc = jsonpp('test.json', 'test.json.pp');
run;
```

This example reads JSON input from a url fileref and pretty prints to disk fileref. The filerefs are created in the FILENAME statements.

```
filename in url "http://example.com/~username/snip.json";
filename out 'pp.txt';
data _null_;
   rc = jsonpp('in','out');
run;
```

JULDATE Function

Returns the Julian date from a SAS date value.

Category: Date and Time

See: "About SAS Date, Time, and Datetime Values" in *SAS Formats and Informats for SAS Viya Workbench*

Syntax

JULDATE(*date*)

Required Argument

date
specifies a SAS date value.

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). Because leading zeros are dropped from the result, the year portion of a Julian date might be omitted (for years ending in 00), or it might 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 example, if YEARCUTOFF=1940, JULDATE would return 97001 for January 1, 1997, and return 1878365 for December 31, 1878.

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

```
data one;
   julian1=juldate('31dec99'd);
   julian2=juldate('01jan2099'd);
   put julian1= julian2=;
run;
```

These statements produce these results:

```
julian1=99365 julian2=2099001
```

See Also

Functions:

- "DATEJUL Function" on page 518
- "JULDATE7 Function" on page 1031

JULDATE7 Function

Returns a seven-digit Julian date from a SAS date value.

Category: Date and Time

See: "About SAS Date, Time, and Datetime Values" in *SAS Formats and Informats for SAS Viya Workbench*

Syntax

JULDATE7(*date*)

Required Argument

date
specifies a SAS date value.

Details

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.

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

```
data one;
   julian1=juldate7('31dec96'd);
   julian2=juldate7('01jan2099'd);
   put julian1= julian2=;
run;
```

These statements produce these results:

```
julian1=1996366 julian2=2099001
```

See Also

Functions:

- "JULDATE Function" on page 1030

KURTOSIS Function

Returns the kurtosis.

Category: Descriptive Statistics

Syntax

KURTOSIS(*argument-1*, *argument-2*, *argument-3*, *argument-4* <, ..., *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

At least four nonmissing arguments are required. Otherwise, the function returns a missing value. If all nonmissing arguments have equal values, the kurtosis is mathematically undefined. The KURTOSIS function returns a missing value and sets _ERROR_ equal to 1.

The argument list can consist of a variable list, which is preceded by OF.

Example

```
data
one;

   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_;

run;
```

These statements produce these results:

```
x1=0.928 x2=-3.3 x3=1.5 x4=-4.483379501 x5=-5.065692754 _ERROR_=0 _N_=1
```

LAG Function

Returns values from a queue.

Category: Special

Note: The VARCHAR type is not supported for arguments in the LAG function.

Syntax

LAG <*n*> (*argument*)

Required Argument

argument
specifies a numeric or character constant, variable, or expression.

Optional Argument

n
specifies the number of lagged values.

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 a value stored previously in that queue. 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.

If the argument of LAG*n* is an array name, a separate queue is maintained for each variable in 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.

```
data one;
   input x @@;
   y=lag1(x);
   z=lag2(x);
   datalines;
1 2 3 4 5 6
;
proc print data=one;
   title 'LAG Output';
run;
```

Output 3.44 *Output from Generating Two Lagged Values*

LAG Output

Obs	x	y	z
1	1	.	.
2	2	1	.
3	3	2	1
4	4	3	2
5	5	4	3
6	6	5	4

LAG1 returns one missing value and the values of X (lagged once). LAG2 returns two missing values and the values of X (lagged twice).

Example 2: Generating Multiple Lagged Values in BY Groups

The following example shows how to generate up to three lagged values within each BY group.

```
/
*************************************************************************
****/
/* This program generates up to three lagged values.  By increasing
the    */
/* size of the array and the number of assignment statements that
use      */
/* the LAGn functions, you can generate as many lagged values as
needed.   */
/
*************************************************************************
****/
/* Create starting data. */
data old;
  input start end;
datalines;
1 1
1 2
1 3
1 4
1 5
1 6
1 7
2 1
2 2
3 1
3 2
3 3
3 4
3 5
;
data new(drop=i count);
  set old;
  by start;
  /* Create and assign values to three new variables.  Use
ENDLAG1-      */
  /* ENDLAG3 to store lagged values of END, from the most recent to
the  */
  /* third preceding
value.                                              */
  array x(*) endlag1-endlag3;
  endlag1=lag1(end);
  endlag2=lag2(end);
  endlag3=lag3(end);
  /* Reset COUNT at the start of each new BY-Group */
  if first.start then count=1;
  /* On each iteration, set to missing array elements   */
  /* that have not yet received a lagged value for the  */
  /* current BY-Group.  Increase count by 1.            */
  do i=count to dim(x);
    x(i)=.;
  end;
  count + 1;
run;
proc print;
run;
```

Output 3.45 *Output from Generating Three Lagged Values*

The SAS System

Obs	start	end	endlag1	endlag2	endlag3
1	1	1	.	.	.
2	1	2	1	.	.
3	1	3	2	1	.
4	1	4	3	2	1
5	1	5	4	3	2
6	1	6	5	4	3
7	1	7	6	5	4
8	2	1	.	.	.
9	2	2	1	.	.
10	3	1	.	.	.
11	3	2	1	.	.
12	3	3	2	1	.
13	3	4	3	2	1
14	3	5	4	3	2

Example 3: Computing the Moving Average of a Variable through the Entire Data Set

The following example computes the moving average of a variable through the entire data set.

```
data x;
do x=1 to 10;
  output;
  end;
run;
/* Compute the moving average of the entire data set. */
data avg;
retain s 0;
set x;
s=s+x;
a=s/_n_;
run;
proc print;
run;
```

Output 3.46 *Output from Computing the Moving Average of a Variable*

The SAS System

Obs	s	x	a
1	1	1	1.0
2	3	2	1.5
3	6	3	2.0
4	10	4	2.5
5	15	5	3.0
6	21	6	3.5
7	28	7	4.0
8	36	8	4.5
9	45	9	5.0
10	55	10	5.5

Example 4: Computing the Moving Average of a Variable of the Last n Observations

The following example computes the moving average of a variable of the last n observations.

```
data x;
do x=1 to 10;
  output;
  end;
run;
%let n=5;
data avg (drop=s);
retain s;
set x;
s=sum (s, x, -lag&n(x)) ;
a=s / min(_n_, &n);
run;
proc print;
run;
```

Output 3.47 *Computing the Moving Average of a Variable of the last n observations.*

The SAS System

Obs	x	a
1	1	1.0
2	2	1.5
3	3	2.0
4	4	2.5
5	5	3.0
6	6	4.0
7	7	5.0
8	8	6.0
9	9	7.0
10	10	8.0

Example 5: Computing the Moving Average of a Variable of the Last n Observations within a BY Group

The following example computes the moving average of a variable of the last n observations within a BY group.

```
data x;
do x=1 to 10;
  output;
  end;
run;
data ds1;
do patient='A','B','C';
 do month=1 to 7;
  num=int(ranuni(0)*10);
  output;
 end;
end;
run;
proc sort;
by patient;
%let n = 4;
data ds2;
set ds1;
by patient;
retain num_sum 0;
if first.patient then do;
  count=0;
  num_sum=0;
end;
count+1;
```

```
last&n=lag&n(num);
if count gt &n then num_sum=sum(num_sum, num, -last&n);
else num_sum=sum(num_sum, num);
if count ge &n then mov_aver=num_sum/&n;
else mov_aver=.;
run;
proc print;
run;
```

Output 3.48 *Computing the Moving Average of a Variable of the last n observations within a BY group*

The SAS System

Obs	patient	month	num	num_sum	count	last4	mov_aver
1	A	1	9	9	1	.	.
2	A	2	0	9	2	.	.
3	A	3	1	10	3	.	.
4	A	4	6	16	4	.	4.00
5	A	5	3	10	5	9	2.50
6	A	6	9	19	6	0	4.75
7	A	7	5	23	7	1	5.75
8	B	1	7	7	1	6	.
9	B	2	8	15	2	3	.
10	B	3	8	23	3	9	.
11	B	4	1	24	4	5	6.00
12	B	5	0	17	5	7	4.25
13	B	6	6	15	6	8	3.75
14	B	7	4	11	7	8	2.75
15	C	1	5	5	1	1	.
16	C	2	7	12	2	0	.
17	C	3	1	13	3	6	.
18	C	4	8	21	4	4	5.25
19	C	5	0	16	5	5	4.00
20	C	6	0	9	6	7	2.25
21	C	7	6	14	7	1	3.50

Example 6: Generating a Fibonacci Sequence of Numbers

The following example 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_;
   put 'Fibonacci Sequence';
   n=1;
   f=1;
   put n= f=;
   do n=2 to 10;
      f=sum(f, lag(f));
      put n= f=;
   end;
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 7: Using Expressions for the LAG Function Argument

The following program uses an expression for the value of *argument* and 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.

```
data one;
   input X @@;
   Y=lag1(x+10);
   Z=lag2(x);
   datalines;
1 2 3 4 5 6
;
proc print;
   title 'Lag Output: Using an Expression';
run;
```

Output 3.49 *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

See Also

Functions:

- "DIF Function" on page 542

LARGEST Function

Returns the *k*th largest nonmissing value.

Category: Descriptive Statistics

Syntax

LARGEST(*k*, *value-1* <, *value-2* ..., *value-n*>)

Required Arguments

k
is a numeric constant, variable, or expression that specifies which value to return.

value
specifies the value of a numeric constant, variable, or expression to be processed.

Details

If *k* is missing, less than zero, or greater than the number of values, the result is a missing value and _ERROR_ is set to 1. Otherwise, if *k* is greater than the number of nonmissing values, the result is a missing value but _ERROR_ is not set to 1.

Example

```
data
one;


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
_all_;

run;
```

These statements produce these results:

```
k=4 largest1=789 largest2=456 largest3=123 largest4=. _ERROR_=0 _N_=1
```

See Also

Functions:

- “ORDINAL Function” on page 1187

- "PCTL Function" on page 1190
- "SMALLEST Function" on page 1395

LBOUND Function

Returns the lower bound of an array.

Category: Array

Syntax

LBOUND <*n*> (*array-name*)

LBOUND(*array-name*, *bound-n*)

Required Arguments

array-name
is the name of an array that was defined previously in the same DATA step.

bound-n
is a numeric constant, variable, or expression that specifies the dimension for which you want to know the lower bound. Use *bound-n* only if *n* is not specified.

Optional Argument

n
is an integer constant that specifies the dimension for which you want to know the lower bound. If no *n* value is specified, the LBOUND function returns the lower bound of the first dimension of the array.

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. Use LBOUND in array processing to avoid changing the lower bound of an iterative DO group each time you change the bounds of the array. LBOUND and HBOUND can be used together to return the values of the lower and upper bounds of an array dimension.

Examples

Example 1: One-Dimensional Array

In this example, LBOUND returns the lower bound of the dimension, a value of 2. SAS repeats the statements in the DO loop five times.

```
data
one;

   array big{2:6} weight sex height state
city;

     do i=lbound(big) to
hbound(big);

   put
i=;


end;

run;
```

These statements produce these results:

```
i=2
i=3
i=4
i=5
i=6
```

Example 2: Multidimensional Array

This example shows two ways of specifying the LBOUND function for multidimensional arrays. Both methods return the same value for LBOUND, as shown in the table that follows the SAS code example.

```
data
one;

 array mult{2:6, 4:13, 2} mult1-
mult100;


t1=LBOUND(MULT);


t2=LBOUND2(MULT);


t3=LBOUND3(MULT);

   put t1= t2=
t3=;
```

```
x1=LBOUND(MULT,1);

x2=LBOUND(MULT,2);

x3=LBOUND(MULT,3);

   put x1= x2=
x3=;

run;
```

These statements produce these results:

```
t1=2 t2=4 t3=1
x1=2 x2=4 x3=1
```

See Also

Functions:

- “DIM Function” on page 545
- “HBOUND Function” on page 907

Statements:

- “ARRAY Statement” in *SAS DATA Step Statements for SAS Viya Workbench*
- “Array Reference Statement” in *SAS DATA Step Statements for SAS Viya Workbench*

Other References:

- “Using Arrays” in *SAS Language Programmer’s Guide for SAS Viya Workbench*

LCM Function

Returns the least common multiple.

Category: Mathematical

Syntax

LCM(*x1*, *x2*, *x3*, ..., *xn*)

Required Argument

x
specifies a numeric constant, variable, or expression that has an integer value.

Details

The LCM (least common multiple) function returns the smallest multiple that is exactly divisible by every member of a set of numbers. For example, the least common multiple of 12 and 18 is 36.

If any of the arguments are missing, then the returned value is a missing value.

Example

The following example returns the smallest multiple that is exactly divisible by the integers 10 and 15.

```
data _null_;
   x=lcm(10, 15);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=30
```

See Also

Functions:

- "GCD Function" on page 754

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.

Category: Combinatorial

Syntax

LCOMB(*n*, *r*)

Required Arguments

n
is a nonnegative integer that represents the total number of elements from which the sample is chosen.

r
is a nonnegative integer that represents the number of chosen elements.

Restriction $r \le n$

Comparisons

The LCOMB function computes the logarithm of the COMB function.

Example

```
data _null_;
   x=lcomb(5000,500);
   y=lcomb(100,10);
   put x= y=;
run;
```

These statements produce these results:

```
x=1621.4411361 y=30.482323362
```

See Also

Functions:

- "COMB Function" on page 445

LEFT Function

Left-aligns a character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: DBCS equivalent function is KLEFT . See "DBCS Compatibility" on page 1049.

Syntax

LEFT(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

The Basics

In a DATA step, if the LEFT 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.

LEFT returns an argument with leading blanks moved to the end of the value. The argument's length does not change.

DBCS Compatibility

The LEFT function left-aligns a character string. You can use the LEFT function in most cases. If an application can be executed in an ASCII environment, or if the application does not manipulate character strings, then using the LEFT function rather than the KLEFT function.

Example

```
data
one;

   a=' DUE
DATE';


b=left(a);
```

```
   put
b=;

run;
```

These statements produce this result:

```
b=DUE DATE
```

See Also

Functions:

- “COMPRESS Function” on page 467
- “RIGHT Function” on page 1346
- “STRIP Function” on page 1416
- “TRIM Function” on page 1470

LENGTH Function

Returns the length of bytes of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tips: DBCS equivalent function is KLENGTH .

The LENGTH function returns a length in bytes. The KLENGTH function returns a length in a character-based unit.

Syntax

LENGTH(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTH function returns an integer that represents the position of the rightmost non-blank character in *string*. If the value of *string* is blank, LENGTH returns a value of 1. If *string* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTH returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

- The LENGTH and LENGTHN functions return the same value for non-blank character strings. LENGTH returns a value of 1 for blank character strings, whereas LENGTHN returns a value of 0.
- 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.
- The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string.

Example

```
data
one;


len=length('ABCDEF');

   len2=length('
');

   put len=
len2=;

run;
```

These statements produce these results:

```
len=6 len2=1
```

See Also

Functions:

- “LENGTHC Function” on page 1052
- “LENGTHM Function” on page 1054
- “LENGTHN Function” on page 1056

LENGTHC Function

Returns the length of a character string, including trailing blanks.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: The LENGTHC function returns a length in bytes.

Syntax

LENGTHC(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTHC function returns the number of characters, both blanks and non-blanks, in *string*. If *string* is a numeric constant, variable or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTHC returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTH and LENGTHN functions return the length of a character string, excluding trailing blanks. LENGTHC always returns a value that is greater than or equal to the value of LENGTHN.
- 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.

Example

```
data
one;

   x=lengthc('variable with trailing blanks
');

   length fixed
$35;

   fixed='variable with trailing blanks
';


y=lengthc(fixed);

   put x=
y=;

run;
```

These statements produce these results:

```
x=32 y=35
```

See Also

Functions:

- “LENGTH Function” on page 1050
- “LENGTHM Function” on page 1054
- “LENGTHN Function” on page 1056

LENGTHM Function

Returns the amount of memory, in bytes, that is allocated for a character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

LENGTHM(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTHM function returns an integer that represents the amount of memory in bytes that is allocated for *string*. If *string* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTHM returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

The LENGTHM function returns the amount of memory in bytes that is allocated for a character string, whereas the LENGTH, LENGTHC, and LENGTHN functions return the length of a character string. LENGTHM always returns a value that is greater than or equal to the values that are returned by LENGTH, LENGTHC, and LENGTHN.

Examples

Example 1: Determining the Amount of Allocated Memory for a Character Expression

This example determines the amount of memory (in bytes) that is allocated for a buffer that stores intermediate results in a character expression. Because SAS does not know how long the value of the expression CAT(x, y) is, SAS allocates memory for values up to 32,767 bytes long.

```
data _null_;
   x='x';
   y='y';
   lc=lengthc(cat(x, y));
   lm=lengthm(cat(x, y));
   put lc= lm=;
run;
```

These statements produce these results:

```
lc=2 lm=32767
```

Example 2: Determining the Amount of Allocated Memory for a Variable from an External File

This example determines the amount of memory (in bytes) that is allocated to a variable that is entered into a SAS file from an external file.

```
data _null_;
   file 'test.txt';
   put 'trailing blanks   ';
run;
data test;
   infile 'test.txt';
   input;
   x=lengthm(_infile_);
   put x;
run;
```

These statements produce this result:

```
256
```

See Also

Functions:

- “LENGTH Function” on page 1050
- “LENGTHC Function” on page 1052

- "LENGTHN Function" on page 1056

LENGTHN Function

Returns the length of bytes in a character string, excluding trailing blanks.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: The LENGTHN function returns a length in bytes.

Syntax

LENGTHN(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTHN function returns the number of bytes that represent the position of the rightmost non-blank character in *string*. For example, the character string in the following example is (ABCDEF). Each character in the string has one byte, so the total number of bytes is 6. Counting from left to right, the right-most character is 6. If the value of *string* is blank, LENGTHN returns a value of 0. If *string* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTHN returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

- The LENGTHN and LENGTH functions return the same value for non-blank character strings. LENGTHN returns a value of 0 for blank character strings, whereas LENGTH returns a value of 1.

- The LENGTHN 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. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHC.
- The LENGTHN function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHM.

Example

```
data
one;


len=lengthn('ABCDEF');

   len2=lengthn('
');

   put len=
len2=;

run;
```

These statements produce these results:

```
len=6 len2=0
```

See Also

Functions:

- “LENGTH Function” on page 1050
- “LENGTHC Function” on page 1052
- “LENGTHM Function” on page 1054

LEXCOMB Function

Generates all distinct combinations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

Restriction: The LEXCOMB function cannot be executed when you use the %SYSFUNC macro.

Syntax

LEXCOMB(*count*, *k*, *variable-1*, ..., *variable-n*)

Required Arguments

count
specifies an integer variable that is assigned values from 1 to the number of combinations in a loop.

k
is a constant, variable, or expression between 1 and *n*, inclusive, that specifies the number of items in each combination.

variable
specifies either all numeric variables, or all character variables that have the same length. The values of these variables are permuted.

Requirement: Initialize these variables before you execute the LEXCOMB function.

Tip: After executing the LEXCOMB function, the first *k* variables contain the values in one combination.

Details

The Basics

Use the LEXCOMB function in a loop where the first argument to LEXCOMB takes each integral value from 1 to the number of distinct combinations of the nonmissing values of the variables. In each execution of LEXCOMB within this loop, *k* should have the same value.

Number of Combinations

When all of the variables have nonmissing, unequal values, then the number of combinations is COMB(*n*,*k*). If the number of variables that have missing values is m, and all the nonmissing values are unequal, then LEXCOMB produces COMB(*n*-*m*,*k*) combinations because the missing values are omitted from the combinations.

When some of the variables have equal values, the exact number of combinations is difficult to compute, but COMB(*n*,*k*) provides an upper bound. You do not need to compute the exact number of combinations, provided that your program leaves the loop when LEXCOMB returns a value that is less than zero.

LEXCOMB Processing

On the first execution of the LEXCOMB function, the following actions occur:

- The argument types and lengths are checked for consistency.

- The m missing values are assigned to the last m arguments.
- The *n-m* nonmissing values are assigned in ascending order to the first *n-m* arguments following *count*.
- LEXCOMB returns 1.

On subsequent executions, up to and including the last combination, the following actions occur:

- The next distinct combination of the nonmissing values is generated in lexicographic order.
- If *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j=i*+1, then LEXCOMB returns *j*.

If you execute the LEXCOMB function after generating all the distinct combinations, then LEXCOMB returns –1.

If you execute the LEXCOMB function with the first argument out of sequence, then the results are not useful. In particular, if you initialize the variables and then immediately execute the LEXCOMB function with a first argument of *j*, you do not get the jth combination (except when *j* is 1). To get the jth combination, you must execute the LEXCOMB function *j* times. The first argument takes values from 1 through *j* in that exact order.

Comparisons

The LEXCOMB function generates all distinct combinations of the nonmissing values of n variables taken *k* at a time in lexicographic order. The ALLCOMB function generates all combinations of the values of k variables taken *k* at a time in a minimum change order.

Examples

Example 1: Generating Distinct Combinations in Lexicographic Order

The following example uses the LEXCOMB function to generate distinct combinations in lexicographic order.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   n=dim(x);
   k=3;
   ncomb=comb(n, k);
   do j=1 to ncomb+1;
      rc=lexcomb(j, k, of x[*]);
      put j 5. +3 x1-x3 +3 rc=;
      if rc<0 then leave;
   end;
run;
```

SAS writes the following output to the log:

```
  1   ant bee cat    rc=1
  2   ant bee dog    rc=3
  3   ant bee ewe    rc=3
  4   ant cat dog    rc=2
  5   ant cat ewe    rc=3
  6   ant dog ewe    rc=2
  7   bee cat dog    rc=1
  8   bee cat ewe    rc=3
  9   bee dog ewe    rc=2
 10   cat dog ewe    rc=1
 11   cat dog ewe    rc=-1
```

Example 2: Generating Distinct Combinations in Lexicographic Order: Another Example

The following is another example of using the LEXCOMB function.

```
data _null_;
   array x[5] $3 ('X' 'Y' 'Z' 'Z' 'Y');
   n=dim(x);
   k=3;
   ncomb=comb(n, k);
   do j=1 to ncomb+1;
      rc=lexcomb(j, k, of x[*]);
      put j 5. +3 x1-x3 +3 rc=;
      if rc<0 then leave;
   end;
run;
```

SAS writes the following output to the log:

```
  1   X Y Y    rc=1
  2   X Y Z    rc=3
  3   X Z Z    rc=2
  4   Y Y Z    rc=1
  5   Y Z Z    rc=2
  6   Y Z Z    rc=-1
```

See Also

Functions:

- “ALLCOMB Function” on page 142

CALL Routines:

- “CALL LEXCOMB Routine” on page 255

LEXCOMBI Function

Generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order.

Category: Combinatorial

Restriction: The LEXCOMBI function cannot be executed when you use the %SYSFUNC macro.

Syntax

LEXCOMBI(*n*, *k*, *index-1*, ..., *k*)

Required Arguments

n
is a numeric constant, variable, or expression that specifies the total number of objects.

K
is a numeric constant, variable, or expression that specifies the number of objects in each combination.

index
is a numeric variable that contains indices of the objects in the combination that is returned. Indices are integers between 1 and *n* inclusive.

Tip If *index-1* is missing or zero, then the LEXCOMBI function initializes the indices to `index-1=1` through `index-k=k`. Otherwise, LEXCOMBI creates a new combination by removing one index from the combination and adding another index.

Details

Before the first execution of the LEXCOMBI function, complete one of the following tasks:

- Set *index-1* equal to zero or to a missing value.
- Initialize *index-1* through *index-k* to distinct integers between 1 and *n* inclusive.

The number of combinations of *n* objects taken *k* at a time can be computed as COMB(*n*,*k*). To generate all combinations of *n* objects taken *k* at a time, execute the LEXCOMBI function in a loop that executes COMB(*n*,*k*) times.

In the LEXCOMBI function, the returned value indicates which, if any, indices changed. If *index-1* through *index-i* did not change, but *index-j* did change, where*j*=*i* +1, then LEXCOMBI returns*i*. If LEXCOMBI is called after the last combinations in lexicographic order have been generated, then LEXCOMBI returns –1.

Comparisons

The LEXCOMBI function generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order. The ALLCOMBI function generates all combinations of the indices of *n* objects taken *k* at a time in a minimum change order.

Example

The following example uses the LEXCOMBI function to generate combinations of indices in lexicographic order.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   array c[3] $3;
   array i[3];
   n=dim(x);
   k=dim(i);
   i[1]=0;
   ncomb=comb(n, k);
   do j=1 to ncomb+1;
      rc=lexcombi(n, k, of i[*]);
      do h=1 to k;
         c[h]=x[i[h]];
      end;
      put @4 j= @10 'i= ' i[*] +3 'c= ' c[*] +3 rc=;
   end;
run;
```

SAS writes the following output to the log:

```
j=1   i= 1 2 3    c= ant bee cat    rc=1
j=2   i= 1 2 4    c= ant bee dog    rc=3
j=3   i= 1 2 5    c= ant bee ewe    rc=3
j=4   i= 1 3 4    c= ant cat dog    rc=2
j=5   i= 1 3 5    c= ant cat ewe    rc=3
j=6   i= 1 4 5    c= ant dog ewe    rc=2
j=7   i= 2 3 4    c= bee cat dog    rc=1
j=8   i= 2 3 5    c= bee cat ewe    rc=3
j=9   i= 2 4 5    c= bee dog ewe    rc=2
j=10  i= 3 4 5    c= cat dog ewe    rc=1
j=11  i= 3 4 5    c= cat dog ewe    rc=-1
```

See Also

CALL Routines:

- “CALL ALLCOMBI Routine” on page 219
- “CALL LEXCOMBI Routine” on page 259

LEXPERK Function

Generates all distinct permutations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

Restriction: The LEXPERK function cannot be executed when you use the %SYSFUNC macro.

Syntax

LEXPERK(*count*, *k*, *variable-1*, ..., *variable-n*)

Required Arguments

count
specifies an integer variable that ranges from 1 to the number of permutations.

k
is a numeric constant, variable, or expression with an integer value between 1 and *n* inclusive.

variable
specifies either all numeric variables, or all character variables that have the same length. The values of these variables are permuted.

Requirement Initialize these variables before you execute the LEXPERK function.

Tip After executing LEXPERK, the first *k* variables contain the values in one permutation.

Details

The Basics

Use the LEXPERK function in a loop where the first argument to LEXPERK takes each integral value from 1 to the number of distinct permutations of *k* nonmissing values of the variables. In each execution of LEXPERK within this loop, *k* should have the same value.

Number of Permutations

When all of the variables have nonmissing, unequal values, the number of permutations is PERM(*n*,*k*). If the number of variables that have missing values is *m*, and all the nonmissing values are unequal, the LEXPERK function produces

PERM(*n*-*m*,*k*) permutations because the missing values are omitted from the permutations. When some of the variables have equal values, the exact number of permutations is difficult to compute, but PERM(*n*,*k*) provides an upper bound. You do not need to compute the exact number of permutations, provided you exit the loop when the LEXPERK function returns a value that is less than zero.

LEXPERK Processing

On the first execution of the LEXPERK function, the following actions occur:

- The argument types and lengths are checked for consistency.
- The *m* missing values are assigned to the last *m* arguments.
- The *n*-*m* nonmissing values are assigned in ascending order to the first *n*-*m* arguments following *count*.
- LEXPERK returns 1.

On subsequent executions, up to and including the last permutation, the following actions occur:

- The next distinct permutation of *k* nonmissing values is generated in lexicographic order.
- If *variable-1* through *variable-i* did not change, but *variable-i* did change, where *j*=*i*+1, then LEXPERK returns *j*.

If you execute the LEXPERK function after generating all the distinct permutations, then LEXPERK returns –1.

If you execute the LEXPERK function with the first argument out of sequence, then the results are not useful. In particular, if you initialize the variables and then immediately execute the LEXPERK function with a first argument of *j*, you do not get the jth permutation (except when *j* is 1). To get the jth permutation, you must execute the LEXPERK function *j* times. The first argument takes values from 1 through *j* in that exact order.

Comparisons

The LEXPERK function generates all distinct permutations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order. The LEXPERM function generates all distinct permutations of the nonmissing values of *n* variables in lexicographic order. The ALLPERM function generates all permutations of the values of several variables in a minimal change order.

Example

Here is an example of the LEXPERK function.

```
data _null_;
   array x[5] $3 ('X' 'Y' 'Z' 'Z' 'Y');
   n=dim(x);
```

```
   k=3;
   nperm=perm(n, k);
   do j=1 to nperm+1;
      rc=lexperk(j, k, of x[*]);
      put j 5. +3 x1-x3 +3 rc=;
      if rc<0 then leave;
   end;
run;
```

SAS writes the following output to the log:

```
 1   X Y Y    rc=1
 2   X Y Z    rc=3
 3   X Z Y    rc=2
 4   X Z Z    rc=3
 5   Y X Y    rc=1
 6   Y X Z    rc=3
 7   Y Y X    rc=2
 8   Y Y Z    rc=3
 9   Y Z X    rc=2
10   Y Z Y    rc=3
11   Y Z Z    rc=3
12   Z X Y    rc=1
13   Z X Z    rc=3
14   Z Y X    rc=2
15   Z Y Y    rc=3
16   Z Y Z    rc=3
17   Z Z X    rc=2
18   Z Z Y    rc=3
19   Z Z Y    rc=-1
```

See Also

Functions:

- “ALLPERM Function” on page 144
- “LEXPERM Function” on page 1065

CALL Routines:

- “CALL RANPERK Routine” on page 312
- “CALL RANPERM Routine” on page 314

LEXPERM Function

Generates all distinct permutations of the nonmissing values of several variables in lexicographic order.

Category: Combinatorial

Syntax

LEXPERM(*count*, *variable-1* <, ..., *variable-N*>)

Required Arguments

count
specifies an integer variable that ranges from 1 to the number of permutations.

variable
specifies either all numeric variables, or all character variables that have the same length. The values of these variables are permuted by LEXPERM.

Requirement Initialize these variables before you execute the LEXPERM function.

Details

Determine the Number of Distinct Permutations

The following variables are defined for use in the equation that follows:

N
specifies the number of variables that are being permuted, that is, the number of arguments minus one.

M
specifies the number of missing values among the variables that are being permuted.

d
specifies the number of distinct nonmissing values among the arguments.

N_i
for i=1, through i=d, N_i specifies the number of instances of the i*th* distinct value.

The number of distinct permutations of nonmissing values of the arguments is expressed as follows:

$$P = \frac{(N_1 + N_2 + ... + N_d)!}{N_1!N_2!...N_d!} < = N!$$

Note: The LEXPERM function cannot be executed with the %SYSFUNC macro.

LEXPERM Processing

Use the LEXPERM function in a loop where the argument *count* takes each integral value from 1 to P. You do not need to compute P provided you exit the loop when LEXPERM returns a value that is less than zero.

For 1=*count*<P, the following actions occur:

- The argument types and lengths are checked for consistency.
- The M missing values are assigned to the last M arguments.
- The N-M nonmissing values are assigned in ascending order to the first N-M arguments following *count*.
- LEXPERM returns 1.

For 1<*count*<=P, the following actions occur:

- The next distinct permutation of the nonmissing values is generated in lexicographic order.
- If *variable-1* through *variable-I* did not change, but *variable-J* did change, where J=I+1, then LEXPERM returns J.

For *count*>P, LEXPERM returns –1.

If the LEXPERM function is executed with the first argument out of sequence, the results might not be useful. In particular, if you initialize the variables and then immediately execute LEXPERM with a first argument of K, you do not get the K*th* permutation (except when K is 1). To get the K*th* permutation, you must execute LEXPERM K times. The first argument takes values from 1 through K in that exact order.

Comparisons

SAS provides three functions or CALL routines for generating all permutations:

- ALLPERM generates all *possible* permutations of the values, *missing or nonmissing*, of several variables. Each permutation is formed from the previous permutation by interchanging two consecutive values.
- LEXPERM generates all *distinct* permutations of the *nonmissing* values of several variables. The permutations are generated in lexicographic order.
- LEXPERK generates all *distinct* permutations of K of the *nonmissing* values of N variables. The permutations are generated in lexicographic order.

ALLPERM is the fastest of these functions and CALL routines. LEXPERK is the slowest.

Example

Here is an example of the LEXPERM function.

```
data _null_;
   array x[6] $1 ('X' 'Y' 'Z' ' ' 'Z' 'Y');
   nfact=fact(dim(x));
   put +3 nfact=;
   do i=1 to nfact;
      rc=lexperm(i, of x[*]);
      put i 5. +2 rc= +2 x[*];
      if rc<0 then leave;
```

```
   end;
run;
```

SAS writes the following output to the log:

```
nfact=720
 1  rc=1   X Y Y Z Z
 2  rc=3   X Y Z Y Z
 3  rc=4   X Y Z Z Y
 4  rc=2   X Z Y Y Z
 5  rc=4   X Z Y Z Y
 6  rc=3   X Z Z Y Y
 7  rc=1   Y X Y Z Z
 8  rc=3   Y X Z Y Z
 9  rc=4   Y X Z Z Y
10  rc=2   Y Y X Z Z
11  rc=3   Y Y Z X Z
12  rc=4   Y Y Z Z X
13  rc=2   Y Z X Y Z
14  rc=4   Y Z X Z Y
15  rc=3   Y Z Y X Z
16  rc=4   Y Z Y Z X
17  rc=3   Y Z Z X Y
18  rc=4   Y Z Z Y X
19  rc=1   Z X Y Y Z
20  rc=4   Z X Y Z Y
21  rc=3   Z X Z Y Y
22  rc=2   Z Y X Y Z
23  rc=4   Z Y X Z Y
24  rc=3   Z Y Y X Z
25  rc=4   Z Y Y Z X
26  rc=3   Z Y Z X Y
27  rc=4   Z Y Z Y X
28  rc=2   Z Z X Y Y
29  rc=3   Z Z Y X Y
30  rc=4   Z Z Y Y X
31  rc=-1   Z Z Y Y X
```

See Also

Functions:

- “ALLPERM Function” on page 144

CALL Routines:

- “CALL ALLPERM Routine” on page 222
- “CALL RANPERK Routine” on page 312
- “CALL RANPERM Routine” on page 314

LFACT Function

Computes the logarithm of the FACT (factorial) function.

Category: Combinatorial

Syntax

LFACT(*n*)

Required Argument

n
is an integer 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

```
data
one;


x=lfact(5000);


y=lfact(100);

   put x=
y=;

run;
```

These statements produce these results:

```
x=37591.143509 y=363.73937556
```

See Also

Functions:

- “FACT Function” on page 587

LGAMMA Function

Returns the natural logarithm of the Gamma function.

Category: Mathematical

Syntax

LGAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

```
data
one;


x=lgamma(2);


y=lgamma(1.5);

   put x=
y=;

run;
```

These statements produce these results:

```
x=0 y=-0.120782238
```

LIBNAME Function

Assigns or clears a libref for a SAS library.

Category: SAS File I/O

Syntax

LIBNAME(*libref* <, *SAS-library* <, *engine* <, *options*>>>)

Required Argument

libref
specifies the libref that is assigned to a SAS library.

Tip The maximum length of *libref* is eight characters.

Optional Arguments

SAS-library
specifies the physical name of the SAS library that is associated with the libref. Specify this name as required by the host operating environment. This argument can be null.

engine
specifies the engine that is used to access SAS files opened in the data library. If you are specifying a SAS/SHARE server, then the value of engine should be REMOTE. This argument can be null.

options
specifies one or more valid options for the specified engine, delimited with blanks. This argument can be null.

Details

Basic Information about Return Codes

The LIBNAME function assigns or clears a libref from a SAS library. When you use the LIBNAME function with two or more arguments, SAS attempts to assign the libref. When you use one argument, SAS attempts to clear the libref. Return codes are generated depending on the value of the arguments that are used in the LIBNAME function and whether the libref is assigned.

When assigning a libref, the return code is 0 if the libref is successfully assigned. If the return code is nonzero and the SYSMSG function returns a warning message or a note, then the assignment was successful. If the SYSMSG function returns an error, then the assignment was unsuccessful.

If a library is already assigned, and you attempt to assign a different name to the library, the libref is assigned, the LIBNAME function returns a nonzero return code, and the SYSMSG function returns a note.

When LIBNAME Has One Argument

When LIBNAME has one argument, the following rules apply:

- If the libref is not assigned, a nonzero return code is returned and the SYSMSG function returns a warning message.
- If the libref is successfully assigned, a 0 return code is returned and the SYSMSG function returns a blank value.

When LIBNAME Has Two Arguments

When LIBNAME has two arguments, the following rules apply:

- If the second argument is null, all blanks, or zero length, SAS attempts to deassign the libref.
- If the second argument is not null, not all blanks, and not zero length, SAS attempts to assign the specified path (the second argument) to the libref.
- If the libref is not assigned, a nonzero return code is returned and the SYSMSG function returns an error message.
- If the libref is successfully assigned, a 0 return code is returned and the SYSMSG function returns a blank value.

When LIBNAME Has Three or Four Arguments

When LIBNAME has three or four arguments, the following rules apply:

- If the second argument is null, all blanks, or zero length, the results depend on your operating environment.
- If the second argument is null and the libref is not already assigned, then a nonzero return code is returned and the SYSMSG function returns an error message.
- If the second argument is null and the libref has already been assigned, then LIBNAME returns a value of 0 and the SYSMSG function returns a blank value.
- If at least one of the previous conditions is not met, then SAS attempts to assign the specified path (second argument) to the libref.
- If the engine is not a SAS engine (for example, ODBC), then the second argument, *SAS-library*, must be missing or empty and cannot contain " or TRIMN(). If the SAS-library argument does contain a value, then the libname is not assigned.

  These two examples use an ODBC engine to assign a libname:

```
rc3 = libname ('mylib3', , 'ODBC', 'DSN=mySQLServer_11');
/**Argument2, SAS-library is missing**/

rc3 = libname ('mylib3',, 'ODBC', 'DSN=mySQLServer_11');
/** Argument2, SAS-library is missing**/
```

  This example uses an ODBC engine but does not assign a libname because TRIMN is used:

```
rc3 = libname ('mylib3',TRIMN(&variable), 'ODBC', 'DSN=mySQLServer_11');
```

Note: In the DATA step, a character constant that consists of two consecutive quotation marks without any intervening spaces is interpreted as a single space, not

as a string with a length of 0. To specify a string with a length of 0, use the TRIMN Function on page 1472.

Operating Environment Information: Some systems allow a *SAS-library* value of ' ' (a space between the single quotation marks) to assign a libref to the current directory. Other systems clear the libref from the SAS library when the *SAS-library* value contains only blanks. The behavior of LIBNAME when a single space is specified for *SAS-library* is dependent on your operating environment. Under some operating environments, you can assign librefs by using system commands that are outside the SAS session.

Examples

Example 1: Assigning a Libref

This example attempts to assign the libref NEW to the SAS library MYLIB. If an error or warning occurs, the message is written to the SAS log. Note that in a macro statement, you do not enclose character strings in quotation marks.

```
%macro
test;

   %let mylib=c:\projects
\May2015;

   %if %sysfunc(libname(new,&mylib))
%then

   %put
%sysfunc(sysmsg());

   %else %put
success;

%mend
test;



%test
```

Example 2: Deassigning a Libref

This example deassigns the libref NEW that was previously associated with the data library MYLIB in the preceding example. If an error or warning occurs, the message is written to the SAS log. In a macro statement, you do not enclose character strings in quotation marks.

```
%macro
test;
```

```
   %if (%sysfunc(libname(new)))
%then

   %put
%sysfunc(sysmsg());

%mend
test;



%test
```

Example 3: Compressing a Library

This example assigns the libref NEW to the MYLIB data library and uses the COMPRESS option to compress the library. This example uses the default SAS engine. In a DATA step, you enclose character strings in quotation marks.

```
data test;
   rc=libname('new', 'MYLIB', , 'compress=yes');
run;
```

See Also

Functions:

- “LIBREF Function” on page 1074
- “SYSMSG Function” on page 1437

LIBREF Function

Verifies that a libref has been assigned.

Category: SAS File I/O

Syntax

LIBREF(*libref*)

Required Argument

libref
specifies the libref to be verified. In a DATA step, *libref* can be a character expression, a string enclosed in quotation marks, or a DATA step variable whose value contains the libref. In a macro, *libref* can be any expression.

Range 1 to 8 characters

Details

The LIBREF function returns 0 if the libref has been assigned, or returns a nonzero value if the libref has not been assigned.

Example

This example verifies a libref. If an error or warning occurs, the message is written to the log. Under some operating environments, the user can assign librefs by using system commands outside the SAS session.

```
%macro
test;

   %if %sysfunc(libref(sashelp))
%then

   %put
%sysfunc(sysmsg());

   %else %put libref
exists;

%mend
test;



%test
```

These statements produce this result:

```
libref exists
```

See Also

Functions:

- “LIBNAME Function” on page 1070

LOG Function

Returns the natural (base e) logarithm.

Category: Mathematical

Syntax

LOG(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

```
data
one;


x=log(1.0);


y=log(10.0);

   put x=
y=;

run;
```

These statements produce these results:

```
x=0 y=2.302585093
```

LOG10 Function

Returns the logarithm to the base 10.

Category: Mathematical

Syntax

LOG10(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

```
data
one;


x=log10(1.0);


y=log10(10.0);


z=log10(100.0);

   put x= y=
z=;

run;
```

These statements produce these results:

```
x=0 y=1 z=2
```

LOG1PX Function

Returns the log of 1 plus the argument.

Category: Mathematical

Syntax

LOG1PX(*x*)

Required Argument

x
specifies a numeric variable, constant, or expression.

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 example computes the log of 1 plus the value 0.5.

```
data _null_;
   x=log1px(0.5);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=0.4054651081
```

Example 2: Comparing the LOG1PX Function with the LOG Function

In the following example, the value of X is computed by using the LOG1PX function. The value of Y is computed by using the LOG function.

```
data _null_;
   x=log1px(1.e-5);
   put x= hex16.;
   y=log(1+1.e-5);
   put y= hex16.;
run;
```

SAS writes the following output to the log:

```
x=3EE4F8AEA9AE7317
y=3EE4F8AEA9AF0A25
```

See Also

Functions:

- “LOG Function” on page 1076

LOG2 Function

Returns the logarithm to the base 2.

Category: Mathematical

Syntax

LOG2(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

```
data
one;


x=log2(2.0);


y=log2(0.5);

   put x=
y=;

run;
```

These statements produce these results:

```
x=1 y=-1
```

LOGBETA Function

Returns the logarithm of the beta function.

Category: Mathematical

Syntax

LOGBETA(*a*, *b*)

Required Arguments

a
is the first shape parameter, where *a*>0.

b
is the second shape parameter, where *b*>0.

Details

The LOGBETA function is mathematically given by the equation
$log(\beta(a,b)) = log(\Gamma(a)) + log(\Gamma(b)) - log(\Gamma(a+b))$

where $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, LOGBETA returns a missing value.

Example

In these statements, the first shape parameter has a value of 5 and the second shape parameter has a value of 3.

```
data _null_;
   x=beta(5,3);
   y=logbeta(5,3);
   put x= y=;
run;
```

These statements produce these results:

```
x=0.0095238095 y=-4.65396035
```

See Also

Functions:

- "BETA Function" on page 194

LOGCDF Function

Returns the logarithm of a left cumulative distribution function.

Category: Probability

See: "CDF Function" on page 392

Syntax

LOGCDF(*'distribution'*, *quantile* <, *parameter-1*, ..., *parameter-k*>)

Required Arguments

'distribution'
is a character constant, variable, or expression that identifies the distribution.

Valid distributions are as follows:

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'

Distribution	Argument
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.

Optional Argument

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

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 "CDF Function" on page 392.

For more information about the distributions that are listed in the table, see "PDF Function" on page 1193.

See Also

Functions:

- "CDF Function" on page 392
- "LOGPDF Function" on page 1084
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "QUANTILE Function" on page 1293
- "SDF Function" on page 1376
- "SQUANTILE Function" on page 1404

LOGISTIC Function

Returns the logistic transformation of the argument.

Category: Mathematical

Syntax

LOGISTIC(*argument*)

Required Argument

argument
is a numeric variable, constant, or expression that specifies the value of a numeric random variable. When *argument* is missing, the LOGISTIC function returns a missing 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 value.

Example

```
data
one;


x=0.5;


z=logistic(x);

   put
z=;

run;
```

These statements produce this result:

```
z=0.6224593312
```

LOGPDF Function

Returns the logarithm of a probability density (mass) function.

Category: Probability

Alias: LOGPMF

See: "PDF Function" on page 1193

Syntax

LOGPDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Required Arguments

'distribution'
is a character constant, variable, or expression that identifies the distribution.

Valid distributions are as follows:

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'

Distribution	Argument
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.

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The LOGPDF function computes the logarithm of the probability density (mass) function from various continuous and discrete distributions. For more information, see "PDF Function" on page 1193.

For more information about the distributions that are listed in the table, see "PDF Function" on page 1193.

See Also

Functions:

- "CDF Function" on page 392
- "LOGCDF Function" on page 1081
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "QUANTILE Function" on page 1293
- "SDF Function" on page 1376
- "SQUANTILE Function" on page 1404

LOGSDF Function

Returns the logarithm of a survival function.

Category: Probability

See: "SDF Function" on page 1376

Syntax

LOGSDF(*'distribution'*, *quantile*, *parameter-1, ..., parameter-k*)

Required Arguments

'distribution'
is a character constant, variable, or expression that identifies the distribution.

Valid distributions are as follows:

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'

Distribution	Argument
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.

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

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 1376.

For more information about the distributions that are listed in the table, see "PDF Function" on page 1193.

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “PDF Function” on page 1193
- “QUANTILE Function” on page 1293
- “SDF Function” on page 1376
- “SQUANTILE Function” on page 1404

LOWCASE Function

Converts all uppercase single-width English alphabet letters in an argument to lowercase.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

LOWCASE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the LOWCASE 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.

The LOWCASE function copies the character argument, converts all uppercase single-width English alphabet letters to lowercase letters, and returns the altered value as a result.

The results of the LOWCASE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*) and indirectly on the ENCODING and LOCALE system options.

Example

```
data
one;


x='INTRODUCTION';


y=lowcase(x);

   put
y=;

run;
```

These statements produce this result:

```
y=introduction
```

See Also

Functions:

- "PROPCASE Function" on page 1259
- "UPCASE Function" on page 1485

LPERM Function

Computes the logarithm of the PERM function, which is the logarithm of the number of permutations of *n* objects, with the option of including *r* number of elements.

Category: Combinatorial

Syntax

LPERM(*n* <, *r*>)

Required Argument

n

is an integer that represents the total number of elements from which the sample is chosen.

Optional Argument

r

is an optional integer value that represents the number of chosen elements. If *r* is omitted, the function returns the factorial of *n*.

Restriction $r \leq n$

Details

The LPERM function computes the logarithm of the PERM function.

Example

```
data _null_;
   x=lperm(5000,500);
   y=lperm(100,10);
   put x= y=;
run;
```

These statements produce these results:

```
x=4232.7715946 y=45.586735935
```

See Also

Functions:

- "PERM Function" on page 1217

LPNORM Function

Returns the L*p* norm of the second argument and subsequent nonmissing arguments.

Category: Descriptive Statistics

Syntax

LPNORM(*p*, *value-1* <, *value-2* ..., *value-n*>)

Required Arguments

p
specifies a numeric constant, variable, or expression that is greater than or equal to 1, which is used as the power for computing the L_p norm.

value
specifies a numeric constant, variable, or expression.

Details

If all arguments have missing values, then the result is a missing value. Otherwise, the result is the L_p norm of the nonmissing values of the second and subsequent arguments.

In the following example, *p* is the value of the first argument, and $x1, x2, ..., xn$ are the values of the other nonmissing arguments.

$$LPNORM(p, x1, x2, ..., xn) = \left(abs(x1)^p + abs(x2)^p + ... + abs(xn)^p\right)^{1/p}$$

Examples

Example 1: Calculating the Lp Norm

The following example returns the L_p norm of the second and subsequent nonmissing arguments.

```
data _null_;
   x1=lpnorm(1, ., 3, 0, .q, -4);
   x2=lpnorm(2, ., 3, 0, .q, -4);
   x3=lpnorm(3, ., 3, 0, .q, -4);
   x999=lpnorm(999, ., 3, 0, .q, -4);
   put x1= / x2= / x3= / x999=;
run;
```

SAS writes the following output to the log:

```
x1=7
x2=5
x3=4.4979414453
x999=4
```

Example 2: Calculating the Lp Norm When You Use a Variable List

The following example uses a variable list and returns the L_p norm.

```
data _null_;
   x1=1;
   x2=3;
   x3=4;
   x4=3;
   x5=1;
   x=lpnorm(of x1-x5);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=11
```

See Also

Functions:

- "EUCLID Function" on page 580 (L2 norm)
- "MAX Function" on page 1099 (Linfinity norm)
- "SUMABS Function" on page 1430 (L1 norm)

MAD Function

Returns the median absolute deviation from the median.

Category: Descriptive Statistics

Syntax

MAD(*value-1* <, *value-2*..., *value-n*>)

Required Argument

value
specifies a numeric constant, variable, or expression of which the median absolute deviation from the median is to be computed.

Details

If all arguments have missing values, the result is a missing value. Otherwise, the result is the median absolute deviation from the median of the nonmissing 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

```
data _null_;
   mad=mad(2, 4, 1, 3, 5, 999999);
   put mad= ;
run;
```

These statements produce this result:

```
mad=1.5
```

See Also

Functions:

- “IQR Function” on page 1025
- “MEDIAN Function” on page 1104
- “PCTL Function” on page 1190

MARGRCLPRC Function

Calculates call prices for European options on stocks, based on the Margrabe model.

Category: Financial

Syntax

MARGRCLPRC(*X*$_1$, *t*, *X*$_2$, *sigma-1*, *sigma-2*, *rho12*)

Required 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.

t
is a nonmissing value that specifies the time to expiration, in years.

X₂
is a nonmissing, positive value that specifies the price of the second asset.

Requirement Specify X_2 and X_1 in the same units.

sigma-1
is a nonmissing, positive fraction that specifies the volatility of the first asset.

sigma-2
is a nonmissing, positive fraction that specifies the volatility of the second asset.

rho12
specifies the correlation between the first and second assets, $\rho_{x_1x_2}$.

Range between –1 and 1

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, in years.

σ_{x1}^2
specifies the variance of the first asset.

σ_{x2}^2
specifies the variance of the second asset.

σ_{x1}
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.

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" on page 11.

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

```
data
one;

   a=margrclprc(15, .5, 13, .06, .05,
1);

   put
a=;

   b=margrclprc(2, .25, 1, .3, .2,
1);

   put
b=;

run;
```

These statements produce these results:

```
a=2
b=1
```

See Also

Functions:

- "MARGRPTPRC Function" on page 1097

MARGRPTPRC Function

Calculates put prices for European options on stocks, based on the Margrabe model.

Category: Financial

Syntax

MARGRPTPRC(*X_1*, *t*, *X_2*, *sigma-1*, *sigma-2*, *rho12*)

Required 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.

t
is a nonmissing value that specifies the time to expiration, in years.

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.

sigma-1
is a nonmissing, positive fraction that specifies the volatility of the first asset.

sigma-2
is a nonmissing, positive fraction that specifies the volatility of the second asset.

rho12
specifies the correlation between the first and second assets, $\rho_{x_1 y_1}$.

Range between –1 and 1

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.

$$pd_1 = \frac{\left(\ln\left(\frac{N_1}{N_2}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$pd_2 = pd_1 - \sigma\sqrt{t}$$

$$\sigma^2 = \sigma_{x1}^2 + \sigma_{x2}^2 - 2\rho_{x1,\,x2}\sigma_{x1}\sigma_{x2}$$

The following arguments apply to the preceding equation:

t
is a nonmissing value that specifies the time to expiration, in years.

σ_{x1}^2
specifies the variance of the first asset.

σ_{x2}^2
specifies the variance of the second asset.

σ_{x1}
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 1094.

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 basic information about pricing, see "Using Pricing Functions" on page 11.

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

```
data
one;

   a=margrptprc(2, .25, 3, .06, .2,
1);

   put
a=;

   b=margrptprc(3, .25, 4, .05, .3,
1);

   put
b=;

run;
```

These statements produce these results:

```
a=1.0000000001
b=1.0015762491
```

See Also

Functions:

- “MARGRCLPRC Function” on page 1094

MAX Function

Returns the largest value.

Category: Descriptive Statistics

Syntax

MAX(*argument-1* <, *argument-2*, ..., *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one argument is required. If you use only one argument, then the value of that argument is returned. The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The MAX function returns a missing value (.) only if all arguments are missing.

The MAX operator (<>) returns a missing value only if both operands are missing. In this case, it returns the value of the operand that is higher in the sort order for missing values.

Example

```
data
one;

   x=max(8,
3);

   x1=max(2,
6, .);

   x2=max(2, -3, 1,
-1);

   x3=max(3, .,
-3);

   x4=max(of x1-
x3);


x5=max(73);

   put
_all_;

run;
```

These statements produce these results:

```
x=8 x1=6 x2=2 x3=3 x4=6 x5=73 _ERROR_=0 _N_=1
```

MD5 Function

Returns an MD5 message digest as a 16-byte binary string for a message consisting of a character string.

Categories: Binary Results
Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: The message can be a VARCHAR and can be any length. If the message is a character constant and contains invalid characters, write it as a hexadecimal constant.

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

MD5(*message*)

Required Argument

message
specifies a character constant, variable, or expression.

Tip Enclose a literal string of characters in quotation marks.

Details

Operating Environment Information: Results from the MD5 function depend on the character encoding of the message. For example, 'abc' is '616263'x in ASCII but '818283'x in EBCDIC, which results in different digests.

The MD5 function returns a binary value, so the result might contain unprintable characters. You can print the result in a readable form by using the $HEX32. format.

Example: Generating Results with the MD5 Function

This example generates results that are returned by the MD5 function:

```
data _null_;
   y=md5('abc');
   z=md5('access method');
   put y=/ y=$hex32.;
   put z=/ z=$hex32.;
run;
```

The output from this program contains unprintable characters.

MDY Function

Returns a SAS date value from month, day, and year values.

Category: Date and Time

Syntax

MDY(*month*, *day*, *year*)

Required Arguments

month
specifies a numeric constant, variable, or expression that represents an integer from 1 through 12.

day
specifies a numeric constant, variable, or expression that represents an integer from 1 through 31.

year
specifies a numeric constant, variable, or expression with a value of a two-digit or four-digit integer that represents the year. The YEARCUTOFF= system option defines the year value for two-digit dates.

Example

```
data null;
   birthday=mdy(8,27,10);
   put birthday=;
   put birthday=worddate.;
run;
data null;
```

```
    anniversary=mdy(7,11,21);
    put anniversary=;
    put anniversary=date9.;
run;
```

These statements produce these results:

```
birthday=18501
birthday=August 27, 2010
anniversary=22472
anniversary=11JUL2021
```

See Also

Functions:

- “DAY Function” on page 521
- “MONTH Function” on page 1120
- “YEAR Function” on page 1573

MEAN Function

Returns the arithmetic mean (average).

Category: Descriptive Statistics

Syntax

MEAN(*argument-1* <, ... *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one nonmissing argument is required. Otherwise, the function returns a missing value.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, and the MEDIAN function returns the median of the

nonmissing values, whereas the MEAN function returns the arithmetic mean (average).

Example

```
data
one;

   x1=mean(2, ., .,
6);

   x2=mean(1, 2, 3,
2);

   x3=mean(of x1-
x2);

 put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=4 x2=2 x3=3
```

See Also

Functions:

- “GEOMEAN Function” on page 758
- “GEOMEANZ Function” on page 759
- “HARMEAN Function” on page 889
- “HARMEANZ Function” on page 891
- “MEDIAN Function” on page 1104

MEDIAN Function

Returns the median value.

Category: Descriptive Statistics

Syntax

MEDIAN(*value–1* <, *value–2*, ..., *value-n*>)

Required Argument

value
is a numeric constant, variable, or expression.

Details

The MEDIAN function returns the median of the nonmissing values. If all arguments have missing values, the result is a missing 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 values, whereas the MEAN function returns the arithmetic mean (average).

Example

```
data _null_;
   x=median(2,4,1,3);
   y=median(5,8,0,3,4);
   put x=;
   put y=;
run;
```

These statements produce these results:

```
x=2.5
y=4
```

See Also

Functions:

- "MEAN Function" on page 1103

MIN Function

Returns the smallest value.

Category: Descriptive Statistics

Syntax

MIN(*argument-1* <, *argument-2, ..., argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one argument is required. If you use only one argument, then the value of that argument is returned. The argument list can consist of a variable list, which is preceded by OF.

Details

The MIN function returns a missing value (.) only if all arguments are missing.

The MIN operator (><) returns a missing value if either operand is missing. In this case, it returns the value of the operand that is lower in the sort order for missing values.

Example

```
data
one;

   x=min(7,
4);

   x1=min(2, .,
6);

   x2=min(2, -3, 1,
-1);

   x3=min(0,
4);
```

```
   x4=min(of x1-
x3);


x5=min(34);

   put
_all_;

run;
```

These statements produce these results:

```
x=4 x1=2 x2=-3 x3=0 x4=-3 x5=34 _ERROR_=0 _N_=1
```

MINUTE Function

Returns the minute from a SAS time or datetime value.

Category: Date and Time

Syntax

MINUTE(*time* | *datetime*)

Required Arguments

time
is a numeric constant, variable, or expression that specifies a SAS time value.

datetime
is a numeric constant, variable, or expression that specifies a SAS datetime value.

Details

The MINUTE function returns an integer that represents a specific minute of the hour. MINUTE always returns a positive number in the range of 0 through 59.

Example

```
data one;
   time='3:19:24't;
   m=minute(time);
```

```
   put m=;
run;
```

These statements produce this result:

```
m=19
```

See Also

Functions:

- “HOUR Function” on page 934
- “SECOND Function” on page 1381

MISSING Function

Returns a numeric result that indicates whether the argument contains a missing value.

Categories: Descriptive Statistics
Character
Missing Values

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

MISSING(*numeric-expression* | *character-expression*)

Required Arguments

numeric-expression
specifies a numeric constant, variable, or expression.

character-expression
specifies a character constant, variable, or expression.

Details

- The MISSING function checks a numeric or character expression for a 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.
- A numeric expression is considered missing if it evaluates to a numeric missing value: ., ._, .A, ..., .Z.
- A character expression is considered missing if it evaluates to a string that contains all blanks or has a length of zero.

Comparisons

The MISSING function can have only one argument. The CMISS function can have multiple arguments and returns a count of the missing values. The NMISS function requires numeric arguments and returns the number of missing values in the list of arguments.

Example

This example uses the MISSING function to check whether the input variables contain missing values.

```
data values;
   input @1 var1 3. @5 var2 3.;
   if missing(var1) then
      do;
         put 'Variable 1 is Missing.';
      end;
   else if missing(var2) then
      do;
         put 'Variable 2 is Missing.';
      end;
   datalines;
127
988 195
;
run;
```

SAS writes the following output to the log:

```
Variable 2 is Missing.
```

See Also

Functions:

- "CMISS Function" on page 436
- "NMISS Function" on page 1139

CALL Routines:

- "CALL MISSING Routine" on page 272

MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.

Category: Mathematical

Syntax

MOD(*argument-1*, *argument-2*)

Required Arguments

argument-1
is a numeric constant, variable, or expression that specifies the dividend.

argument-2
is a numeric constant, variable, or expression that specifies the divisor.

Restriction cannot be 0

Details

The MOD function returns the remainder from the division of *argument-1* by *argument-2*. 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.

  To determine the largest integer for which the computation is exact, execute the following DATA step:

```
data _null_;
   exactint=constant('exactint');
   put exactint=;
```

```
run;
```

Operating Environment Information: For information about the largest integer, see the SAS documentation for your operating environment.

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

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a 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 missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

```
data
one;

   x1=mod(10,
3);

   put x1=
9.4;

   xa=modz(10,
3);

   put xa=
9.4;

   x2=mod(.3,
-.1);

   put x2=
9.4;
```

```
   xb=modz(.3,
-.1);

   put xb=
9.4;


x3=mod(1.7, .1);

   put x3=
9.4;


xc=modz(1.7, .1);

   put xc=
9.4;


x4=mod(.9, .3);

   put x4=
24.20;


xd=modz(.9, .3);

   put xd=
24.20;

run;
```

These statements produce these results:

```
x1=1.0000
xa=1.0000
x2=0.0000
xb=0.1000
x3=0.0000
xc=0.0000
x4=0.000000000000000000
xd=0.000000000000011102
```

See Also

Functions:

- “INT Function” on page 963
- “INTZ Function” on page 1020
- “MODZ Function” on page 1118

MODEXIST Function

Determines whether a software image exists in the version of SAS that you have installed.

Category: External Routines

Operating environment: This function is supported under Linux.

Syntax

Linux:

MODEXIST('*product-name*' | '*pathname*')

Required Arguments

'product-name'
specifies a character constant, variable, or expression that is the name of the product image that you are checking.

'pathname'
specifies the pathname for the product image that you are checking.

Details

When you supply a product name, the MODEXIST function determines whether a software image exists in the version of SAS that you have installed. If an image exists, then MODEXIST returns a value of 1. If an image does not exist, then MODEXIST returns a value of 0.

When you supply a pathname, the MODEXIST function searches the directories that are listed in the *pathname* argument for an executable module. The name of the executable module is passed to MODEXIST. MODEXIST returns 1 if the module is found, and 0 if the module is not found.

Comparisons

The MODEXIST function determines whether a software image exists in the version of SAS that you have installed. The SYSPROD function determines whether a product is licensed.

Example

This example determines whether a product is licensed and the image is installed. The example returns a value of 1 if a SAS/GRAPH image is installed in your version of SAS, and returns a value of 0 if the image is not installed. The SYSPROD function determines whether the product is licensed.

```
data _null_;
   rc1=sysprod('graph');
   rc2=modexist('sasgplot');
   put rc1= rc2=;
run;
```

SAS writes the following output to the log:

```
rc1=1 rc2=1
```

MODULE Function

Calls a specific routine or module that resides in an external dynamic link library (DLL).

Category: External Routines

Note: The MODULEN and MODULEC functions return a number *num* or a character *char*, respectively.

CAUTION: **Be sure to use the correct arguments and attributes.** Using incorrect arguments or attributes for a DLL function can cause SAS, and possibly your operating system, to fail.

Syntax

CALL MODULE(<*control-string*>,*module*,*argument-1, ..., argument-n*);

MODULEN(<*control-string*>,*module*,*argument-1, ..., argument-n*);

MODULEC(<*control-string*> ,*module*,*argument-1, ..., argument-n*);

CALL MODULEI(<*control-string*> ,*module*,*argument-1, ..., argument-n*);

MODULEIN(<*control-string*> ,*module*,*argument-1, ..., argument-n*)

MODULEIC(<*control-string*> ,*module*,*argument-1, ..., argument-n*);

Required Arguments

module
is the name of the external module to use, specified as a DLL name and the routine name or ordinal value, separated by a comma. The module must reside in a dynamic link library (DLL) and it must be externally callable. For example, the value `'KERNEL32,GetProfileString'` specifies to load KERNEL32.DLL and

to invoke the GetProfileString routine. Note that while the DLL name is not case sensitive, the routine name is based on the restraints of the routine's implementation language, so the routine name is case sensitive.

If the DLL supports ordinal-value naming, you can provide the DLL name followed by a decimal number, such as `'XYZ,30'`.

You do not need to specify the DLL name if you specified the MODULE attribute for the routine in the SASCBTBL attribute table, as long as the routine name is unique (that is, no other routines have the same name in the attribute file).

You can specify *module* as a SAS character expression instead of as a constant; most often, though, you pass it as a constant.

argument
are the arguments to pass to the requested routine. Use the proper attributes for the arguments (numeric arguments for numeric attributes and character arguments for character attributes).

Optional Argument

control-string
is an optional control string whose first character must be an asterisk (*), followed by any combination of characters.

Here are valid characters:

I prints the hexadecimal representations of all arguments to the MODULE function and to the requested DLL routine before and after the DLL routine is called. You can use this option to help diagnose problems that are caused by incorrect arguments or attribute tables. If you specify the **I** option, the **E** option is implied.

E prints detailed error messages. Without the **E** option (or the **I** option, which supersedes it), the only error message that the MODULE function generates is "Invalid argument to function," which is usually not enough information to determine the cause of the error.

S*x* uses *x* as a separator character to separate field definitions. You can then specify *x* in the argument list as its own character argument to serve as a delimiter for a list of arguments that you want to group together as a single structure. Use this option only if you do not supply an entry in the SASCBTBL attribute table. If you do supply an entry for this module in the SASCBTBL attribute table, you should use the FDSTART option in the ARG statement in the table to separate structures.

H provides brief help information about the syntax of the MODULE routines, the attribute file format, and the suggested SAS formats and informats.

For example, the control string `'*IS/'` specifies that parameter lists be printed and that the string `'/'` is to be treated as a separator character in the argument list.

Details

The MODULE functions execute a routine *module* that resides in an external (outside SAS) dynamic link library with the specified arguments *arg-1* through *arg-n*.

The MODULE call routine does not return a value. The MODULEN and MODULEC functions return a number *num* or a character *char*, respectively. Which routine you use depends on the expected return value of the DLL function that you want to execute.

MODULEI, MODULEIC, and MODULEIN are special versions of the MODULE functions that permit vector and matrix arguments. Their return values are still scalar. You can invoke these functions only from PROC IML.

Other than this name difference, the syntax for all six routines is the same.

The MODULE function builds a parameter list by using the information in *arg-1* to *arg-n* and by using a routine description and argument attribute table that you define in a separate file. Before you invoke the MODULE routine, you must define the fileref of SASCBTBL to point to this external file. You can name the file whatever you want when you create it.

You can use SAS variables and formats as arguments to the MODULE function and ensure that these arguments are properly converted before being passed to the DLL routine.

CALL MODULEI, MODULEIN, and MODULEIC permit vector and matrix arguments; you can use them within the IML procedure. For more information, see the *SAS/IML Studio: User's Guide*.

MODULEC Function

Calls an external routine and returns a character value.

Category: External Routines

See: "CALL MODULE Routine" on page 274

Syntax

MODULEC(<*cntl-string*, > *module-name* <, *argument-1*, ..., *argument-n*>)

Details

For details about the MODULEC function, see "CALL MODULE Routine" on page 274.

See Also

Functions:

- “MODULEN Function” on page 1117

CALL Routines:

- “CALL MODULE Routine” on page 274

MODULEN Function

Calls an external routine and returns a numeric value.

Category: External Routines

See: “CALL MODULE Routine” on page 274

Syntax

MODULEN(<*cntl-string*, > *module-name* <, *argument-1, ..., argument-n*>)

Details

For details about the MODULEN function, see “CALL MODULE Routine” on page 274.

See Also

Functions:

- “MODULEC Function” on page 1116

CALL Routines:

- “CALL MODULE Routine” on page 274

MODZ Function

Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.

Category: Mathematical

Syntax

MODZ(*argument-1*, *argument-2*)

Required Arguments

argument-1
is a numeric constant, variable, or expression that specifies the dividend.

argument-2
is a nonzero numeric constant, variable, or expression that specifies the divisor.

Details

The MODZ function returns the remainder from the division of *argument-1* by *argument-2*. 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.

  To determine the largest integer for which the computation is exact, execute the following DATA step:

```
data _null_;
   exactint=constant('exactint');
   put exactint=;
run;
```

  Operating Environment Information: For information about the largest integer, see the SAS documentation for your operating environment.

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 extra computations, called 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 missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

```
data
one;

   x1=mod(10,
3);

   put x1=
9.4;

   xa=modz(10,
3);

   put xa=
9.4;

  x2=mod(.3,
-.1);

   put x2=
9.4;

   xb=modz(.3,
-.1);

   put xb=
9.4;


x3=mod(1.7, .1);

   put x3=
9.4;


xc=modz(1.7, .1);

   put xc=
9.4;
```

```
x4=mod(.9, .3);

   put x4=
24.20;


xd=modz(.9, .3);

   put xd=
24.20;

run;
```

These statements produce these results:

```
x1=1.0000
xa=1.0000
x2=0.0000
xb=0.1000
x3=0.0000
xc=0.0000
x4=0.000000000000000000
xd=0.000000000000011102
```

See Also

Functions:

- “INT Function” on page 963
- “INTZ Function” on page 1020
- “MOD Function” on page 1110

MONTH Function

Returns the month from a SAS date value.

Category: Date and Time

Syntax

MONTH(*date*)

Required Argument

date
specifies a numeric constant, variable, or expression that represents a SAS date value.

Details

The MONTH function returns a numeric value that represents the month from a SAS date value. Numeric values can range from 1 through 12.

Example

```
data one;
   date='25jan21'd;
   m=month(date);
   put m=;
run;
```

These statements produce this result:

```
m=1
```

See Also

Functions:

- “DAY Function” on page 521
- “YEAR Function” on page 1573

MOPEN Function

Opens a file by directory ID and member name, and returns either the file identifier or a 0.

Category:	External Files
Operating environment:	Open-modes are specific to the Linux OS.

Syntax

MOPEN(*directory-id*, *member-name* <, *open-mode* <, *record-length* <, *record-format*>>>)

Required Arguments

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

member-name
is a character constant, variable, or expression that specifies the member name in the directory.

Optional Arguments

open-mode
is a character constant, variable, or expression that specifies the type of access to the file.

Here are valid values:

A APPEND mode allows writing new records after the current end of the file.

I INPUT mode allows reading only (default).

O OUTPUT mode defaults to the OPEN mode specified in the operating environment option in the FILENAME statement or function. If no operating environment option is specified, it allows writing new records at the beginning of the file.

S Sequential input mode is used for pipes and other sequential devices such as hardware ports.

U UPDATE mode allows both reading and writing.

W Sequential Update mode is used for pipes and other sequential devices such as ports.

Default I

record-length
is a numeric variable, constant, or expression that specifies a new logical record length for the file. To use the existing record length for the file, specify a length of 0, or do not provide a value here.

record-format
is a character constant, variable, or expression that specifies a new record format for the file. To use the existing record format, do not specify a value here.

The following values are valid:

B specifies that data is to be interpreted as binary data.

D specifies the default record format.

E specifies the record format that you can edit.

F specifies that the file contains fixed-length records.

P specifies that the file contains printer carriage control in operating environment-dependent record format.

V specifies that the file contains variable-length records.

Note: If an argument is invalid, then MOPEN returns 0. You can obtain the text of the corresponding error message from the SYSMSG function. Invalid arguments do not produce a message in the SAS log and do not set the _ERROR_ automatic variable.

Note: If a folder contains a file that is not of type text, then MOPEN should only use binary-record format to read and download the member. If the internal format of the binary content is unknown, MOPEN should not be used to read and interpret the content.

Details

MOPEN returns the identifier for the file, or 0 if the file could not be opened. You can use a *file-id* that is returned by the MOPEN function as you would use a *file-id* returned by the FOPEN function.

CAUTION
Use OUTPUT mode with care. Opening an existing file for output might overwrite the current contents of the file without warning.

The member is identified by *directory-id* and *member-name* instead of by a fileref. You can also open a directory member by using FILENAME to assign a fileref to the member, followed by a call to FOPEN. However, when you use MOPEN, you do not have to use a separate fileref for each member.

If the file already exists, the output and Update modes default to the operating environment option (append or replace) specified with the FILENAME statement or function. For example,

```
%let rc=%sysfunc(filename(file, physical-name, , mod));
%let did=%sysfunc(dopen(&file));
%let fid=%sysfunc(mopen(&did, member-name, o, 0, d));
%let rc=%sysfunc(fput(&fid, This is a test.));
%let rc=%sysfunc(fwrite(&fid));
%let rc=%sysfunc(fclose(&fid));
```

If `'file'` already exists, FWRITE appends the new record instead of writing it at the beginning of the file. However, if no operating environment option is specified with the FILENAME function, the output mode implies that the record be replaced.

If the open fails, use SYSMSG to retrieve the message text.

Operating Environment Information: The term *directory* in this description refers to an aggregate grouping of files that are managed by the operating environment. Different host operating environments identify such groupings with different names, such as directory, subdirectory, folder, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment. Opening a directory member for output or append is not possible in some operating environments.

Example

This example assigns the fileref MYDIR to a directory. It opens the directory, determines the number of members, retrieves the name of the first member, and opens that member. The last three arguments to MOPEN are the defaults. Note that in a macro statement, you do not enclose character strings in quotation marks.

```
%macro
test;

%let
filrf=mydir;

%let rc=%sysfunc(filename(filrf, physical-
name));

%let did=
%sysfunc(dopen(&filrf));

%let
frstname=;

%let memcount=
%sysfunc(dnum(&did));



 %if (&memcount > 0) %then
%do;

  %let frstname= %sysfunc(dread(&did,
1));

  %let fid= %sysfunc(mopen(&did, &frstname, i, 0,
d));

  %let rc=
%sysfunc(fclose(&fid));


%end;
```

```
%mend
test;


%test
```

See Also

Functions:

- “DCLOSE Function” on page 522
- “DNUM Function” on page 555
- “DOPEN Function” on page 556
- “DREAD Function” on page 569
- “FCLOSE Function” on page 591
- “FILENAME Function” on page 611
- “FOPEN Function” on page 720
- “FPUT Function” on page 733
- “FWRITE Function” on page 744
- “SYSMSG Function” on page 1437

MORT Function

Returns amortization parameters.

Category: Financial

Syntax

MORT(*a*, *p*, *r*, *n*)

Required Arguments

a
is numeric, and specifies the initial amount.

p
is numeric, and specifies the periodic payment.

r
is numeric, and specifies the periodic interest rate that is expressed as a fraction.

n
is an integer, and specifies the number of compounding periods.

Range $n \geq 0$

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.

If Interest Rate is 0 (R=0), then the result will be the outstanding amount divided by number of payments.

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 these statements, an amount of $50,000 is borrowed for 30 years at an annual interest rate of 10% compounded monthly. These statements express the monthly payment:

```
data
one;

 payment=mort(50000, . , .10/12,
30*12);
```

```
 put
payment=;

run;
```

These statements produce this result:

```
payment=438.79
```

The second argument has been 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.

MSPLINT Function

Returns the ordinate of a monotonicity-preserving interpolating spline.

Category: Mathematical

Syntax

MSPLINT(*X*, *n*, X_1 <, X_2, ..., X_n>, Y_1 <, Y_2, ..., Y_n> <, D_1, D_n>)

Required Arguments

X
is a numeric constant, variable, or expression that specifies the abscissa for which the ordinate of the spline is to be computed.

n
is a numeric constant, variable, or expression that specifies the number of knots. *N* must be a positive integer.

X_1, ..., X_n
are numeric constants, variables, or expressions that specify the abscissas of the knots. These values must be nonmissing and listed in nondecreasing order. Otherwise, the result is undefined. MSPLINT does not check the order of the X_1 through X_n arguments.

Y_1, ..., Y_n
are numeric constants, variables, or expressions that specify the ordinates of the knots. The number of Y_1 through Y_n arguments must be the same as the number of X_1 throughX_n arguments.

Optional Argument

***D*, *D*$_n$**
are optional numeric constants, variables, or expressions that specify the derivatives of the spline at X_1 and X_n. These derivatives affect only abscissas that are less than X_2 or greater than X_{n-1}.

Details

The MSPLINT function returns the ordinate of a monotonicity-preserving cubic interpolating spline for a single abscissa, *X*.

An interpolating spline is a function that passes through each point that is specified by the ordered pairs (X_1, Y_1), (X_2, Y_2), ..., (X_n, Y_n). These points are called knots.

A spline preserves monotonicity if both of the following conditions are true:

- For any two or more consecutive knots with nondecreasing ordinates, all interpolated values within that interval are also nondecreasing.
- For any two or more consecutive knots with nonincreasing ordinates, all interpolated values within that interval are also nonincreasing.

However, if you specify values of D_1 or D_n with the wrong sign, monotonicity is not preserved for values that are less than X_2 or greater than X_{n-1}.

If the arguments D_1 and D_n are omitted or missing, then the following actions occur:

- For *n*=1, MSPLINT returns Y_1.
- For *n*=2, MSPLINT uses linear interpolation or extrapolation.

If the arguments D_1 and D_n have nonmissing values, or if *n* >= 3, then the following actions occur:

- If $X < X_1$ or $X > X_n$, MSPLINT uses linear extrapolation.
- If $X_1 <= X <= X_n$, MSPLINT uses cubic spline interpolation.

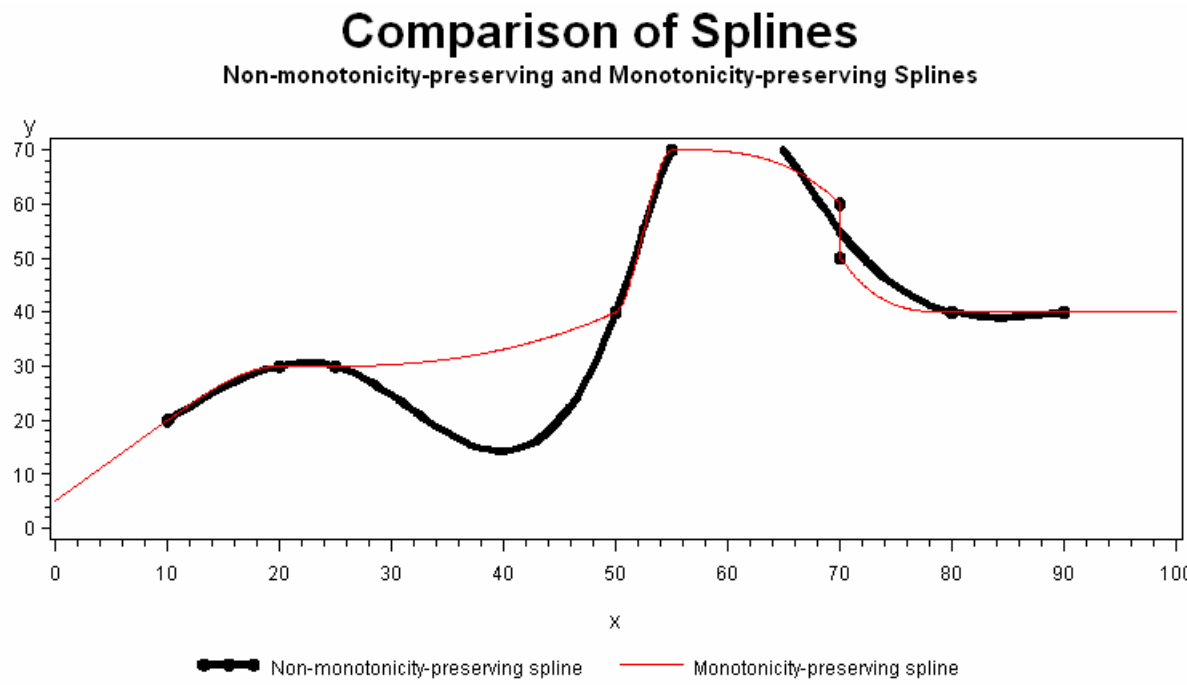
If two knots have equal abscissas but different ordinates, then the spline is discontinuous at that abscissa. If two knots have equal abscissas and equal ordinates, then the spline is continuous at that abscissa, but the first derivative is usually discontinuous at that abscissa. Otherwise, the spline is continuous and has a continuous first derivative.

If *X* is missing, or if any other arguments required to compute the result are missing, then MSPLINT returns a missing value. MSPLINT does not check all of the arguments for missing values. Because the arguments D_1 and D_n are optional, and they are not required to compute the result, if one or both are missing and no errors occur, then MSPLINT returns a nonmissing result.

Example

Here is an example of the MSPLINT function.

```
data msplint;
   do x=0 to 100 by .1;
      msplint=msplint(x, 9,
         10, 20, 25, 50, 55, 70, 70, 80, 90,
         20, 30, 30, 40, 70, 60, 50, 40, 40);
      output;
   end;
run;
data knots;
   input x y;
   datalines;
10 20
20 30
25 30
50 40
55 70
70 60
70 50
80 40
90 40
;
data plot;
   merge knots msplint;
   by x;
run;
title "Comparison of
Splines";

title2 "Non-monotonicity-preserving and Monotonicity-preserving

Splines";
legend1 value=('Non-monotonicity-preserving
spline'

               'Monotonicity-preserving spline')
label=none;

symbol1 value=dot  interpol=spline color=black
width=5;

symbol2 value=none interpol=join
color=red;

proc gplot
data=plot;

   plot y*x=1 msplint*x=2/overlay
legend=legend1;

run;
quit;
```



References

Fritsch, F. N., and J. Butland. 1984. "A Method for Constructing Local Monotone Piecewise Cubic Interpolants." *Siam Journal of Scientific and Statistical Computing* 5:2: 300-304.

MVALID Function

Checks the validity of a character string for use as a SAS member name.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

MVALID(*libname*, *string*, *member-type* <, *valid-member-name*>)

Required Arguments

libname
specifies a character constant, variable, or expression that associates a SAS library with a libref. Leading and trailing blanks are ignored.

string
specifies a character constant, variable, or expression that is checked to determine whether its value can be used as a SAS member name. Leading and trailing blanks are ignored.

member-type
specifies a character constant, variable, or expression that is the member type of the member name that you are using. Leading and trailing blanks are ignored. The value of *member-type* is not validated.

The following member types are available:

ACCESS	specifies access descriptor files that are created by SAS/ACCESS.
CATALOG	specifies SAS catalogs.
DATA	specifies SAS data sets.
FDB	specifies a financial database.
ITEMSTOR	specifies a SAS data set that consists of pieces of information that can be accessed independently. The SAS Registry is an example of an item store.
MDDB	specifies a multidimensional database.
PROGRAM	specifies stored compiled SAS programs.
VIEW	specifies SAS views.

Optional Argument

valid-member-name
specifies a character constant, variable, or expression. The values for *valid-member-name* can be uppercase or lowercase. Leading and trailing blanks are ignored.

Here are valid values:

COMPAT
COMPATIBLE
determines that *string* is a valid SAS member name when all three of the following conditions are true:

- The *string* argument begins with an English letter or an underscore.
- All subsequent characters are English letters, underscores, or digits.
- The length of *string* is 32 or fewer alphanumeric characters.

EXTEND
determines that *string* is a valid SAS member name when all of the following conditions are true:

- The length of *string* is 32 or fewer bytes.
- The *string* argument does not contain the characters / \ * ? " < > | : –

  Note: The SPD Engine also does not allow '$' as the first character. It also does not allow a period (.) in the member name.

- The *string* argument does not contain null bytes.
- The *string* argument does not begin with a blank or period (.).
- The *string* argument contains at least one character. A name that consists of all blanks is not valid.

Default VALIDMEMNAME= is set to COMPAT.

Note If no value is specified, the MVALUE function determines that *string* is a valid SAS member name based on the value of the VALIDMEMNAME= system option.

Details

The Basics

The MVALID function checks the value of *string* to determine whether it can be used as a SAS member name.

The MVALID function returns a value of 1 if *string* can be used as a SAS member name, and a value of 0 if *string* cannot be used as a SAS member name.

MVALID returns a missing value if one of the following conditions is true:

- The *libname* argument is not an assigned libref.
- The *member-type* argument is longer than nine characters.
- The *valid-member-name* argument does not have one of the following values: COMPATIBLE, COMPAT, or EXTEND, regardless of whether the value is uppercase or lowercase.

Requirements for Validation of a SAS Member Name

The *string* argument is evaluated to determine whether it is a valid SAS member name. An engine name with its associated library, as well as member type, affect the validation of *string*. Of the member types, only DATA, ITEMSTOR, and VIEW allow names with extended characters. When *string* is evaluated, the EXTEND value of the optional *valid-member-name* argument is taken into account. Not all engines support *valid-member-name* processing. For the engines that do not, *string* is validated based on the rules for that engine.

The following example shows you how to use the MVALID function to determine whether *string* is a valid SAS member name, based on engine name, DATA member type, and the EXTEND value for *valid-member-name*:

```
libname V9eng V9 'mypath';
data _null_;
   rc=MVALID('V9eng', 'my name', 'data', 'extend');
   put rc=;
run;
```

The following items apply to the preceding example:

- The example returns a value of 1, indicating that '*my name*' is a valid member name for the V9 engine when member type equals DATA and *valid-member-name* equals EXTEND.
- If you use the V6 engine in the example, the program returns a value of 0, indicating that '*my name*' is not valid when member type equals DATA and *valid-member-name* equals EXTEND. The V6 engine does not support *valid-member-name* processing.

In the following example, CATALOG is used instead of DATA for member type:

```
libname V9eng V9 'mypath';
data _null_;
   rc=MVALID('V9eng', 'my name', 'catalog', 'extend');
   put rc=;
run;
```

The following items apply to the preceding example:

- If you use CATALOG in the example instead of DATA, the program returns a value of 0, indicating that '*my name*' is not valid when member type equals CATALOG and *valid-member-name* equals EXTEND. The member type CATALOG does not support extended names, and therefore the EXTEND value for *valid-member-name* is not valid.
- If you use COMPAT in the example instead of EXTEND, the program returns a value of 0, indicating that '*my name*' is not valid when member type equals CATALOG and *valid-member-name* equals COMPAT. The COMPAT value of *valid-member-name* does not allow spaces in member names.

N Function

Returns the number of nonmissing numeric values.

Categories: Descriptive Statistics
Missing Values

Syntax

N(*argument-1* <, ... *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one argument is required. The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The N function counts nonmissing values, whereas the NMISS and the CMISS functions count missing values. N requires numeric arguments, whereas CMISS works with both numeric and character values.

Example

```
data
one;

   x1=n(1, 0,., 2,
5, .);

   x2=n(1,
2);

   x3=n(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=4 x2=2 x3=2
```

NETPV Function

Returns the net present value as a percent.

Category: Financial

Syntax

NETPV(*r*, *frequency*, *c0, c1, ..., cn*)

Required Arguments

r
is numeric, the interest rate over a specified base period of time expressed as a fraction.

Range *r* >= 0

frequency
is numeric, the number of payments during the base period of time that is specified with the rate *r*.

Range *frequency* > 0

Note The case *frequency* = 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.

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 *frequency*>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 *frequency*>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 *frequency* to 3 and set *r* to .04.

If *frequency* 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 *frequency* 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 six years and an annual discount rate of 10%, the net present value of the investment can be expressed with these statements:

```
data
one;

 value=netpv(.10, .5, -500, 200, 300,
400);

 put
value=;

run;
```

These statements produce this result:

```
value=95.982864829
```

See Also

Functions:

- "NPV Function" on page 1176

NLITERAL Function

Converts a character string that you specify to a SAS name literal.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

NLITERAL(*string*)

Required Argument

string
specifies a character constant, variable, or expression that is to be converted to a SAS name literal.

Restriction If the string is a valid SAS variable name, it is not changed.

Tip Enclose a literal string of characters in quotation marks.

Details

Length of Returned Variable

In a DATA step, if the NLITERAL function returns a value to a variable that has not previously been assigned a length, then the variable is given a length of 200 bytes.

The Basics

String is converted to a name literal, unless it qualifies under the default rules for a SAS variable name. These default rules are in effect when the SAS system option VALIDVARNAME=V7:

- It begins with an English letter or an underscore.
- All subsequent characters are English letters, underscores, or digits.
- The length is 32 or fewer alphanumeric characters.

String qualifies as a SAS variable name, when all of these rules are true.

The NLITERAL function encloses the value of *string* in single or double quotation marks, based on the contents of *string*.

Value in *string*	**Result**
an ampersand (&)	enclosed in single quotation marks
a percent sign (%)	enclosed in single quotation marks
more double quotation marks than single quotation marks	enclosed in single quotation marks
none of the above	enclosed in double quotation marks

If insufficient space is available for the resulting n-literal, NLITERAL returns a blank string, writes an error message, and sets _ERROR_ to 1.

Example

This example demonstrates multiple uses of NLITERAL.

```
data test;
   input string $32.;
   length result $ 67;
   result=nliteral(string);
   datalines;
abc_123
This and That
cats & dogs
Company's profits (%)
"Double Quotes"
'Single Quotes'
;
proc print;
title 'Strings Converted to N-Literals or Returned Unchanged';
run;
```

Output 3.50 *Output from Converting Strings to Name Literals with NLITERAL*

Strings Converted to N-Literals or Returned Unchanged

Obs	string	result
1	abc_123	abc_123
2	This and That	"This and That"N
3	cats & dogs	'cats & dogs'N
4	Company's profits (%)	'Company''s profits (%)'N
5	"Double Quotes"	'"Double Quotes"'N
6	'Single Quotes'	'''Single Quotes'''N

See Also

Functions:

- “COMPARE Function” on page 447
- “DEQUOTE Function” on page 532
- “NVALID Function” on page 1178

System Options:

- “VALIDVARNAME= System Option” in *SAS System Options for SAS Viya Workbench*

Other References:

- "Words and Names" in *SAS Language Programmer's Guide for SAS Viya Workbench*

NMISS Function

Returns the number of missing numeric values.

Categories: Descriptive Statistics
Missing Values

Syntax

NMISS(*argument-1* <,... *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one argument is required. The argument list can consist of a variable list, which is preceded by OF.

Details

The NMISS function returns the number of missing values, whereas the N function returns the number of nonmissing values. NMISS requires numeric values, whereas CMISS works with both numeric and character values. NMISS works with multiple numeric values, whereas MISSING works with only one value that can be either numeric or character.

Example

```
data
one;

 x1=nmiss(1, 0, ., 2,
5, .);

 x2=nmiss(1,
0);
```

```
 x3=nmiss(of x1-
x2);

 put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=2 x2=0 x3=0
```

NOMRATE Function

Returns the nominal annual interest rate.

Category: Financial

Syntax

NOMRATE(*compounding-interval*, *rate*)

Required Arguments

compounding-interval
is a SAS interval. This value represents how often the returned value is compounded.

rate
is numeric. *Rate* is the effective annual interest rate (expressed as a percentage) that is compounded at each interval.

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

```
data
one;

 /*If an effective rate is 10% when compounded monthly, the
corresponding nominal rate can
be
   expressed as follows
*/

 effective_rate1=NOMRATE('MONTH',
10);



 /*If an effective rate is 10% when compounded quarterly, the
corresponding nominal rate can
be
   expressed as follows
*/

 effective_rate2=NOMRATE('QUARTER',
10);

 put effective_rate1=
effective_rate2=;

run;
```

These statements produce these results:

```
effective_rate1=9.5689685147 effective_rate2=9.6454756338
```

NORMAL Function

Returns a random variate from a normal, or Gaussian, distribution.

Category: Random Number

Alias: RANNOR

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Normal') function.

See: "CALL STREAMINIT Routine" on page 352

NOTALNUM Function

Searches a character string for a non-alphanumeric character, and returns the first position at which the character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Syntax

NOTALNUM(*string* <, *start*>)

Required Argument

string
specifies a character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTALNUM function to search a string from left to right for non-alphanumeric characters.

```
data _null_;
   string='Next = Last + 1;';
   j=0;
   do until(j=0);
      j=notalnum(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
```

```
        end;
      end;
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=;
That's all
```

See Also

Functions:

- "ANYALNUM Function" on page 147

NOTALPHA Function

Searches a character string for a nonalphabetic character, and returns the first position at which the character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTALPHA(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTALPHA function to search a string from left to right for nonalphabetic characters.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notalpha(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=;
That's all
```

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;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec,hex2.);
   notalpha=notalpha(byte);
   output;
 end;
 proc print data=test;
 run;
```

See Also

Functions:

- "ANYALPHA Function" on page 150

NOTCNTRL Function

Searches a character string for a character that is not a control character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTCNTRL(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   notcntrl=notcntrl(byte);
   output;
 end;
 proc print data=test;
 run;
```

See Also

Functions:

- “ANYCNTRL Function” on page 152

NOTDIGIT Function

Searches a character string for any character that is not a digit, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTDIGIT(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTDIGIT function to search for a character that is not a digit.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notdigit(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=;
That's all
```

See Also

Functions:

- “ANYDIGIT Function” on page 154

NOTE Function

Returns an observation ID for the current observation of a SAS data set.

Category: SAS File I/O

Syntax

NOTE(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

Details

You can use the observation ID value to return to the current observation by using POINT. Observations can be marked by using NOTE and then returned to later by using POINT. Each observation ID is a unique numeric value.

To free the memory that is associated with an observation ID, use DROPNOTE.

Example

This example calls CUROBS to display the observation number, calls NOTE to mark the observation, and calls POINT to point to the observation that corresponds to NOTEID.

```
%macro
test;

%let dsid=
%sysfunc(open(sashelp.cars,i));

/* Go to observation 10 in data set
*/

%let rc=%sysfunc(fetchobs(&dsid,
10));

%if %sysfunc(abs(&rc))
%then

%put FETCHOBS
FAILED;

%else

%do;

/* Display observation number
*/

/* in the Log
*/

%let cur=
%sysfunc(curobs(&));

%put
CUROBS=&cur;

/* Mark observation 10 using NOTE
*/

%let noteid=
%sysfunc(note(&dsid));

/* Rewind pointer to beginning
*/

/* of data
*/

/* set using REWIND
*/
```

```
%let rc=
%sysfunc(rewind(&dsid));

/* FETCH first observation into DDV
*/

%let rc=
%sysfunc(fetch(&dsid));

/* Display first observation number
*/

%let cur=
%sysfunc(curobs(&dsid));

%put
CUROBS=&cur;

/* POINT to observation 10 marked
*/

/* earlier by NOTE
*/

%let rc=%sysfunc(point(&dsid,
&noteid));

/* FETCH observation into DDV
*/

%let rc=
%sysfunc(fetch(&dsid));

/* Display observation number 10
*/

/* marked by NOTE
*/

%let cur=
%sysfunc(curobs(&dsid));

%put
CUROBS=&cur;

%end;

%if (&dsid > 0)
%then

%let rc=
%sysfunc(close(&dsid));

%mend test;
%test
```

These statements produce these results:

```
CUROBS=10
CUROBS=1
CUROBS=10
```

See Also

Functions:

- “DROPNOTE Function” on page 570
- “OPEN Function” on page 1183
- “POINT Function” on page 1221
- “REWIND Function” on page 1344

NOTFIRST Function

Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTFIRST(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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, or if the second argument has a missing value, 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 example 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_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notfirst(string, j+1);
      if j=0 then put +3 "That's all";
```

```
         else do;
            c=substr(string, j, 1);
            put +3 j= c=;
         end;
      end;
   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=;
That's all
```

See Also

Functions:

- "ANYFIRST Function" on page 156

NOTGRAPH Function

Searches a character string for a non-graphical character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTGRAPH(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTGRAPH function to search a string for a non-graphical character.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notgraph(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=7 c=
j=11 c=
j=13 c=
That's all
```

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;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   notgraph=notgraph(byte);
   output;
 end;
 proc print data=test;
 run;
```

See Also

Functions:

- “ANYGRAPH Function” on page 158

NOTLOWER Function

Searches a character string for a character that is not a lowercase letter, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTLOWER(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTLOWER function to search a string for any character that is not a lowercase letter.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notlower(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=;
That's all
```

See Also

Functions:

- "ANYLOWER Function" on page 161

NOTNAME Function

Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTNAME(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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, or if the second argument has a missing value, 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 example 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_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notname(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=;
That's all
```

See Also

Functions:

- “ANYNAME Function” on page 163

NOTPRINT Function

Searches a character string for a nonprintable character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTPRINT(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and LOCALE system options.

The NOTPRINT function searches a string for the first occurrence of a nonprintable 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, or if the second argument has a missing value, 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 nonprintable 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;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   notprint=notprint(byte);
   output;
 end;
 proc print data=test;
 run;
```

These statements produce these results. Partial output is displayed.

Output 3.51 *Output for the NORPRINT Function*

The SAS System

Obs	dec	byte	hex	notprint
1	0		00	1
2	1	©	01	1
3	2	®	02	1
4	3		03	1
5	4	™	04	1
6	5		05	1
7	6		06	1
8	7		07	1
9	8		08	1
10	9		09	1
11	10		0A	1
12	11		0B	1
13	12		0C	1
14	13		0D	1
15	14		0E	1
16	15		0F	1
17	16		10	1
18	17		11	1
19	18		12	1
20	19		13	1
21	20		14	1
22	21		15	1
23	22		16	1

See Also

Functions:

- “ANYPRINT Function” on page 165

NOTPUNCT Function

Searches a character string for a character that is not a punctuation character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTPUNCT(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTPUNCT function to search a string for characters that are not punctuation characters.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notpunct(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=7 c=
j=9 c=n
j=11 c=
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
That's all
```

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;
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   notpunct=notpunct(byte);
   output;
 end;
 proc print data=test;
 run;
```

See Also

Functions:

- “ANYPUNCT Function” on page 168

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 position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTSPACE(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTSPACE function to search a string for a character that is not a whitespace character.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notspace(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=;
That's all
```

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;
```

```
do dec=0 to 255;
   byte=byte(dec);
   hex=put(dec, hex2.);
   notspace=notspace(byte);
   output;
 end;
 proc print data=test;
 run;
```

See Also

Functions:

- "ANYSPACE Function" on page 170

NOTUPPER Function

Searches a character string for a character that is not an uppercase letter, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTUPPER(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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 for SAS Viya Workbench*) and indirectly on the ENCODING and 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, or if the second argument has a missing value, 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 example uses the NOTUPPER function to search a string for any character that is not an uppercase letter.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notupper(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
```

```
        end;
      end;
   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=;
That's all
```

See Also

Functions:

- “ANYUPPER Function” on page 173

NOTXDIGIT Function

Searches a character string for a character that is not a hexadecimal character, and returns the first position at which that character is found.

Category: Character

Restriction: This function is assigned an I18N Level 1 status unless a VARCHAR variable is used, or if the function is threaded or runs in DS2. If these exceptions occur, then this function is assigned an I18N Level 2 status. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

NOTXDIGIT(*string* <, *start*>)

Required Argument

string
is the character constant, variable, or expression to search.

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

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, or if the second argument has a missing value, 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 example uses the NOTXDIGIT function to search a string for a character that is not a hexadecimal character.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
      j=notxdigit(string, j+1);
      if j=0 then put +3 "That's all";
      else do;
         c=substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
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=;
That's all
```

See Also

Functions:

- “ANYXDIGIT Function” on page 175

NPV Function

Returns the net present value with the rate expressed as a percentage.

Category: Financial

Syntax

NPV(*r*, *frequency*, *c0*, *c1*, ..., *cn*)

Required Arguments

r
is numeric, the interest rate over a specified base period of time expressed as a percentage.

frequency
is numeric, the number of payments during the base period of time specified with the rate *r*.

Range *frequency* > 0

Note The case *frequency* = 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.

Comparisons

The NPV function is identical to NETPV, except that the *r* argument is provided as a percentage. See "NETPV Function".

Example

For an initial investment of $500 that returns biannual payments of $200, $300, and $400 over the succeeding six years and an annual discount rate of 10%, the net present value of the investment can be expressed with these statements:

```
data
one;

 value=npv(.10, .5, -500, 200, 300,
400);

 put
value=;

run;
```

These statements produce this output.

```
value=396.01197086
```

NVALID Function

Checks the validity of a character string for use as a SAS variable name.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

NVALID(*string* <, *validvarname*>)

Required Argument

string
specifies a character constant, variable, or expression that is checked to determine whether its value can be used as a SAS variable name.

Note: Trailing blanks are ignored.

Tip Enclose a literal string of characters in quotation marks.

Optional Argument

validvarname
is a character constant, variable, or expression that specifies a value.

Here are valid values:

V7
determines that *string* is a valid SAS variable name when all three of the following are true:

- *string* begins with an English letter or an underscore
- All subsequent characters are English letters, underscores, or digits
- The length is 32 or fewer bytes

ANY
determines that *string* is a valid SAS variable name if it contains 32 or fewer bytes of any type, including blanks.

NLITERAL
determines that *string* is a valid SAS variable name if it is in the form of a SAS name literal ('name'N) or if it is a valid SAS variable name when VALIDVARNAME=V7.

See V7 above in this same list.

UPCASE
determines that *string* is a valid SAS variable name that is in uppercase.

Default If no value is specified, the NVALID function determines that *string* is a valid SAS variable name based on the value of the SAS system option VALIDVARNAME=.

Details

The NVALID function checks the value of *string* to determine whether it can be used as a SAS variable name.

The NVALID function returns a value of 1 or 0.

Condition	Returned Value
string can be used as a SAS variable name	1
string cannot be used as a SAS variable name	0

Example

This example determines the validity of specified strings as SAS variable names. The value that is returned by the NVALID function varies with the *validvarname* argument. The value of 1 is returned when the string is determined to be a valid SAS variable name under the rules for the specified *validvarname* argument. Otherwise, the value of 0 is returned.

```
options validvarname=v7 ls=64;
data string;
   input string $char40.;
   v7=nvalid(string, 'v7');
   any=nvalid(string, 'any');
   nliteral=nvalid(string, 'nliteral');
   default=nvalid(string);
   datalines;
Tooooooooooooooooooooooooo Long
OK
Very_Long_But_Still_OK_for_V7
1st_char_is_a_digit
Embedded blank
!@#$%^&*
"Very Loooong N-Literal with """N
'No closing quotation mark
;
```

```
proc print noobs;
title1 'NLITERAL and Validvarname Arguments Determine';
title2 'Invalid (0) and Valid (1) SAS Variable Names';
run;
```

Output 3.52 *Output from Determining the Validity of SAS Variable Names with NLITERAL*

NLITERAL and Validvarname Arguments Determine
Invalid (0) and Valid (1) SAS Variable Names

string	v7	any	nliteral	default
Tooooooooooooooooooooooooo Long	0	0	0	0
OK	1	1	1	1
Very_Long_But_Still_OK_for_V7	1	1	1	1
1st_char_is_a_digit	0	1	0	0
Embedded blank	0	1	0	0
!@#$%^&*	0	1	0	0
"Very Loooong N-Literal with """N	0	0	1	0
'No closing quotation mark	0	1	0	0

See Also

Functions:

- "COMPARE Function" on page 447
- "NLITERAL Function" on page 1136

System Options:

- "VALIDVARNAME= System Option" in *SAS System Options for SAS Viya Workbench*

Other References:

- "Words and Names" in *SAS Language Programmer's Guide for SAS Viya Workbench*

NWKDOM Function

Returns the date for the *n*th occurrence of a weekday for the specified month and year.

Category: Date and Time

Syntax

NWKDOM(*n*, *weekday*, *month*, *year*)

Required Arguments

n
specifies the numeric week of the month that contains the specified day.

Range 1–5

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

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

year
specifies a four-digit calendar year.

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 will be 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 will produce 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 example uses the NWKDOM function and returns the date for specific occurrences of a weekday for a specified month and year.

```
data _null_;
      /* Return the date of the third Monday in May 2021. */
   a=nwkdom(3, 2, 5, 2021);
      /* Return the date of the fourth Wednesday in November 2021. */
   b=nwkdom(4, 4, 11, 2021);
      /* Return the date of the fourth Saturday in November 2021. */
   c=nwkdom(4, 7, 11, 2021);
      /* Return the date of the first Sunday in January 2022. */
   d=nwkdom(1, 1, 1, 2022);
      /* Return the date of the second Tuesday in September 2021. */
   e=nwkdom(2, 3, 9, 2021);
      /* Return the date of the fifth Thursday in December 2021. */
   f=nwkdom(5, 5, 12, 2021);
   put a= weekdatx.;
   put b= weekdatx.;
   put c= weekdatx.;
   put d= weekdatx.;
   put e= weekdatx.;
   put f= weekdatx.;
run;
```

SAS writes the following output to the log:

```
a=Monday, 17 May 2021
b=Wednesday, 24 November 2021
c=Saturday, 27 November 2021
d=Sunday, 2 January 2022
e=Tuesday, 14 September 2021
f=Thursday, 30 December 2021
```

Example 2: Returning the Date of the Last Monday in May

The following example returns the date that corresponds to the last Monday in the month of May in the year 2021 .

```
data _null_;
      /* The last Monday in May. */
   x=nwkdom(5, 2, 5, 2021);
   put x date9.;
run;
```

SAS writes the following output to the log:

```
31MAY2021
```

See Also

Functions:

- “HOLIDAY Function” on page 911
- “INTNX Function” on page 1004
- “MONTH Function” on page 1120
- “WEEKDAY Function” on page 1569

OPEN Function

Opens a SAS data set.

Category: SAS File I/O

Note: The OPEN function supports integer variables: INT64, INT32, INT16, INT8.

Syntax

OPEN(<*data-set-name* <, *mode* <, *generation-number* <, *type*>>>>)

Optional Arguments

data-set-name
is a character constant, variable, or expression that specifies the name of the SAS data set or SAS SQL view to be opened.

The value of this character string should be in this form:

<*libref*.> *member-name*<(*data-set-options*)>

Default The default value for *data-set-name* is _LAST_.

Restriction If you specify the FIRSTOBS= and OBS= data set options, they are ignored. All other data set options are valid.

mode
is a character constant, variable, or expression that specifies the type of access to the data set.

Here are valid values:

I opens the data set in INPUT mode (default). Values can be read but not modified. **'I'** uses the strongest access mode available in the engine. That is, if the engine supports random access, OPEN defaults to random access. Otherwise, the file is opened in **'IN'** mode automatically. Files are opened with sequential access and a system-level warning is set.

IN opens the data set in INPUT mode. Observations are read sequentially, and you are allowed to revisit an observation.

IS opens the data set in INPUT mode. Observations are read sequentially, but you are not allowed to revisit an observation.

Default I

generation-number
specifies a consistently increasing number that identifies one of the historical versions in a generation group.

Tip The *generation-number* argument is ignored if *type*=F.

type
is a character constant:

Here are valid values:

D
specifies that the first argument, *data-set-name*, is a one-level or two-level data set name. The following example shows how the D *type* value can be used:

```
rc=open('lib.mydata', , , 'D');
```

Tip D is the default if there is no fourth argument.

F
specifies that the first argument, *data-set-name*, is a filename, a physical path to a file. The following examples show how the F *type* value can be used:

```
rc=open('c:\data\mydata.sas7bdat', , , 'F');
rc=open('c:\data\mydata', , , 'F');
```

Tip If you use the F value, then the third argument, *generation-number*, is ignored.

Notes If an argument is invalid, OPEN returns 0. You can obtain the text of the corresponding error message from the SYSMSG function. Invalid arguments do not produce a message in the SAS log and do not set the _ERROR_ automatic variable.

When using the F option, dataset options are not supported on the first argument.

Details

The OPEN function opens a SAS data set, DATA step, or a SAS SQL view and returns a unique numeric data set identifier, which is used in most other data set access functions. OPEN returns 0 if the data set could not be opened.

If you call the OPEN function from a macro, then the result of the call is valid only when the result is passed to functions in a macro. If you call the OPEN function from the DATA step, then the result is valid only when the result is passed to functions in the same DATA step.

By default, a SAS data set is opened with a control level of RECORD. For more information, see "CNTLLEV= Data Set Option" in *SAS Data Set Options for SAS Viya Workbench*. An open SAS data set should be closed when it is no longer needed. If you open a data set within a DATA step, it closes automatically when the DATA step ends.

OPEN defaults to the strongest access mode available in the engine. That is, if the engine supports random access, OPEN defaults to random access. Otherwise, data sets are opened with sequential access, and a system-level warning is set.

Example

```
/*This example opens the data set CARS in the library SASHELP using
INPUT mode.
Note that in a macro statement, you do not enclose character strings
in quotation marks. */


%macro
test;

 %let dsid=%sysfunc(open(sashelp.cars,
i));

  %if (&dsid=0)
%then

   %put
%sysfunc(sysmsg());
```

```
%else

   %put CARS data set has been
opened;

 %let rc=
%sysfunc(close(&dsid));

%mend
test;



%test



/*This example passes values from macro or DATA step variables to be
used on
data set options.
It opens the data set Sasuser.Houses, and uses the WHERE= data set
option to
apply a permanent WHERE clause. Note that in a macro statement,
you do not enclose character strings in quotation
marks.*/



%macro
test;

 %let choice =
style="RANCH";

 %let dsid=%sysfunc(open(sasuser.houses(where=(&choice)),
i));

 %let rc=
%sysfunc(close(&dsid));

%mend
test;



%test



/*This example shows how to check the returned value for errors and to
write
an error message from the SYSMSG
function.*/
```

```
data
_null_;

 d=open('bad',
'?');

  if not d then
do;


m=sysmsg();

    put
m;


abort;

  end;
run;
```

See Also

Functions:

- “CLOSE Function” on page 435
- “SYSMSG Function” on page 1437

ORDINAL Function

Returns the k*th* smallest of the missing and nonmissing values.

Category: Descriptive Statistics

Syntax

ORDINAL(*k*, *argument-1*, *argument-2* <, ...*argument-n*>)

Required Arguments

k
is a numeric constant, variable, or expression with an integer value that is less than or equal to the number of subsequent elements in the list of arguments.

argument
specifies a numeric constant, variable, or expression. At least two arguments are required. An argument can consist of a variable list, preceded by OF.

Details

The ORDINAL function returns the k*th* smallest value, either missing or nonmissing, among the second through the last arguments.

Comparisons

The ORDINAL function counts both missing and nonmissing values, whereas the SMALLEST function counts only nonmissing values.

Example

```
data
one;

   x1=ordinal(4, 1, 2, 3, -4, 5, 6,
7);

   put
x1=;

run;
```

These statements produce this result:

```
x1=3
```

PATHNAME Function

Returns the physical name of an external file or a SAS library, or returns a blank.

Categories: SAS File I/O
External Files

Operating environment: *fileref* or *libref* argument can also specify a Linux environment variable.

Syntax

PATHNAME((*fileref* | *libref*) <, *search-ref*>)

Required Arguments

fileref
is a character constant, variable, or expression that specifies the fileref that is assigned to an external file.

Operating Environment Information:

In a DATA step, *fileref* can be a character expression, a string enclosed in quotation marks, or a DATA step variable whose value contains the fileref. In macro code, *fileref* can be any expression that resolves to a macro variable. The value of *fileref* can be a Linux environment variable.

libref
is a character constant, variable, or expression that specifies the libref that is assigned to a SAS library.

Operating Environment Information:

In a DATA step, *libref* can be a character expression, a string enclosed in quotation marks, or a DATA step variable whose value contains the libref. In macro code, *libref* can be any expression. The value of *libref* can be a Linux environment variable.

Optional Argument

search-ref
is a character constant, variable, or expression that specifies whether to search for a fileref or a libref.

F specifies a search for a fileref.

L specifies a search for a libref.

Details

PATHNAME returns the physical name of an external file or SAS library, or blank if *fileref* or *libref* is invalid.

If the name of a fileref is identical to the name of a libref, you can use the *search-ref* argument to choose which reference you want to search. If you specify a value of F, SAS searches for a fileref. If you specify a value of L, SAS searches for a libref.

If you do not specify a *search-ref* argument, and the name of a fileref is identical to the name of a libref, PATHNAME searches first for a libref. If a libref does not exist, PATHNAME then searches for a fileref.

The default length of the target variable in the DATA step is 200 characters.

You can assign a fileref to an external file by using the FILENAME statement or the FILENAME function.

You can assign a libref to a SAS library using the LIBNAME statement or the LIBNAME function. Some operating environments enable you to assign a libref using system commands.

Example

This example uses the FILEREF function to verify that the fileref MYFILE is associated with an external file. Then it uses PATHNAME to retrieve the actual name of the external file:

```
data _null_;
   length fname $ 100;
   rc=fileref('myfile');
   if (rc=0) then
   do;
      fname=pathname('myfile');
      put fname=;
   end;
run;
```

See Also

Functions:

- “FEXIST Function” on page 605
- “FILEEXIST Function” on page 609
- “FILENAME Function” on page 611
- “FILEREF Function” on page 616

Statements:

- “FILENAME Statement” in *SAS Global Statements for SAS Viya Workbench*
- “LIBNAME Statement” in *SAS Global Statements for SAS Viya Workbench*

PCTL Function

Returns the percentile that corresponds to the percentage.

Category: Descriptive Statistics

Syntax

PCTL<*n*> (*percentage*, *value–1* <, *value–2*, ..., *value-n*>)

Required Arguments

percentage
is a numeric constant, variable, or expression that specifies the percentile to be computed.

Requirement is numeric where, 0 ≤ *percentage* ≤ 100.

value
is a numeric variable, constant, or expression.

Optional Argument

n
is a digit from 1 to 5 which specifies the definition of the percentile to be computed.

Default definition 5

Details

The PCTL function returns the percentile of the nonmissing values corresponding to the percentage. If *percentage* is missing, less than zero, or greater than 100, the PCTL function generates an error message.

Note: The formula that is used in the PCTL function is the same formula that used in PROC UNIVARIATE in Base SAS Procedures Guide: Statistical Procedures. For more information, see "SAS Elementary Statistics Procedures" in *Base SAS Procedures for SAS Viya Workbench*.

Example

```
data
one;


   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=;



run;
```

These statements produce these results:

```
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.

Category: Probability

Alias: PMF

Syntax

PDF(*distribution*, *quantile* <, *parameter-1, ... , parameter-k*>)

Required Arguments

distribution

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

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 constant, variable, or expression that specifies the value of the random variable.

Optional Argument

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 1195 for complete information about these parameters.

Details

Bernoulli Distribution

PDF ('BERNOULLI', *x*, *p*)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

Details

The PDF function for the Bernoulli distribution returns the probability density function of a Bernoulli distribution, 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.

Beta Distribution

PDF ('BETA', *x*, *a*, *b* <, *l*, *r*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

b

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $b > 0$

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range $r > l$

Details

The PDF function for the beta distribution returns the probability density function of a beta distribution, 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 $\varepsilon \le \frac{x-l}{r-l} \le 1 - 2\varepsilon$.

Binomial Distribution

PDF ('BINOMIAL', *m*, *p*, *n*)

Arguments

m

is an integer random variable that counts the number of successes.

Range $m = 0, 1, \ldots$

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

n

is an integer parameter that counts the number of independent Bernoulli trials.

Range $n = 0, 1, \ldots$

Details

The PDF function for the binomial distribution returns the probability density function of a binomial distribution, with the parameters *p* and *n*, which 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.

Cauchy Distribution

PDF ('CAUCHY', *x* <, *θ*, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

Details

The PDF function for the Cauchy distribution returns the probability density function of a Cauchy distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*.

$$PDF('CAUCHY', x, \theta, \lambda) = \frac{1}{\pi}\left(\frac{\lambda}{\lambda^2 + (x-\theta)^2}\right)$$

Chi-Square Distribution

PDF ('CHISQUARE', *x*, *df* <, *nc*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

df

is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range $nc \geq 0$

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. The following equation describes the PDF function for the 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 the 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 the equation, $p_g(y,b)$ is the density from the gamma distribution, which is given by the following equation:

$$p_g(y, b) = \frac{1}{\Gamma(b)} e^{-y} y^{b-1}$$

Conway-Maxwell-Poisson Distribution

PDF('CONMAXPOI',*y*,*λ*,*v*)

Arguments

y
is a numeric constant, variable, or expression that specifies a nonnegative integer representing a count.

λ
is a numeric constant, variable, or expression that specifies a location parameter, similar to the Poisson mean parameter.

v
is a numeric constant, variable, or expression that specifies a dispersion parameter.

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 the following equation:

$$P\left(Y = y; \lambda, \nu\right) = \frac{1}{Z(\lambda, \nu)} \frac{\lambda^y}{(y!)^\nu} \qquad y = 0, 1, 2, \ldots$$

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 introduction of 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, enabling 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 the following 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 the variance of the Conway-Maxwell-Poisson distribution are defined by the following 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 its 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 the following 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:

$$D\left(Y\right) = \frac{V(Y)}{E(Y)}$$

Exponential Distribution

PDF ('EXPONENTIAL', *x* <, *λ*>)

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 scale parameter.

Default 1

Range λ > 0

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*.

$$PDF('EXPO', x, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

F Distribution

PDF ('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 the numerator degrees of freedom.

Range *ndf*> 0

ddf

is a numeric constant, variable, or expression that specifies the denominator degrees of freedom.

Range *ddf* > 0

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range *nc* ≥ 0

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 $\lambda = nc$. The following 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.

Gamma Distribution

PDF ('GAMMA', *x*, *a* <, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

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 > 0

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*.

$$PDF('GAMMA', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda^a \Gamma(a)} x^{a-1} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

Generalized Poisson Distribution

PDF ('GENPOISSON', *x*, *θ*, *η*)

Arguments

x

is a numeric constant, variable, or expression that specifies an integer random variable.

θ

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $<10^5$ and >0

η

is a numeric constant, variable, or expression that specifies a shape parameter.

Range ≥0 and <0.95

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:

$$f(x;\theta,\eta) = \theta(\theta+\eta x)^{x-1} e^{-\theta-\eta x}/x!, \quad x = 0, 1, 2, ..., \qquad \theta > 0,\ 0 \leq \eta < 1$$

Geometric Distribution

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* ≥ 0

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

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*.

$$PDF('GEOM', m, p) = \begin{cases} 0 & m < 0 \\ p(1-p)^m & m \ge 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Hypergeometric Distribution

PDF ('HYPER', *x*, *N*, *R*, *n* <, *o*>)

Arguments

x

is a numeric constant, variable, or expression that specifies an integer random variable.

N

is a numeric constant, variable, or expression that specifies an integer population size parameter.

Range $N = 1, 2, \ldots$

R

is a numeric constant, variable, or expression that specifies an integer number of items in the category of interest.

Range $R = 0, 1, \ldots, N$

n

is a numeric constant, variable, or expression that specifies an integer sample size parameter.

Range $n = 1, 2, \ldots, N$

o

is a numeric constant, variable, or expression that specifies an optional odds ratio parameter.

Range $o > 0$

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.

$$PDF\ ('HYPER', x, N, R, n, o) = \begin{cases} 0 & x < max(0, R+n-N) \\ \dfrac{\binom{R}{x}\binom{N-R}{n-x}o^x}{\sum_{j=max(0,R+n-N)}^{min(R,n)}\binom{R}{j}\binom{N-R}{n-j}o^j} & max(0, R+n-N) \le x \le min(R, n) \\ 0 & x > min(R, n) \end{cases}$$

Laplace Distribution

PDF ('LAPLACE', *x* <, *θ*, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

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*.

$$PDF('LAPLACE', x, \theta, \lambda) = \frac{1}{2\lambda} exp\left(- \frac{|x-\theta|}{\lambda} \right)$$

Logistic Distribution

PDF ('LOGISTIC', *x* <, *θ*, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

θ

is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

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*.

$$PDF('LOGISTIC', x, \theta, \lambda) = \frac{exp\left(-\frac{x-\theta}{\lambda}\right)}{\lambda\left(1+exp\left(-\frac{x-\theta}{\lambda}\right)\right)^2}$$

Lognormal Distribution

PDF ('LOGNORMAL', *x* <, *θ*, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

θ

is a numeric constant, variable, or expression that specifies a log scale parameter. exp(θ) is a scale parameter.

Default 0

λ

is a numeric constant, variable, or expression that specifies a shape parameter.

Default 1

Range λ > 0

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*.

$$PDF('LOGN', x, \theta, \lambda) = \begin{cases} 0 & x \le 0 \\ \frac{1}{x\lambda\sqrt{2\,\pi}}\ exp\left(-\frac{(log(x)-\theta)^2}{2\lambda^2}\right) & x > 0 \end{cases}$$

Negative Binomial Distribution

PDF ('NEGBINOMIAL', *m*, *p*, *n*)

Arguments

m

is a numeric constant, variable, or expression that specifies a positive integer random variable that counts the number of failures.

Range *m*= 0, 1, ...

p
is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

n
is a numeric constant, variable, or expression that specifies a value that counts the number of successes.

Range $n>0$

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.

Normal Distribution

PDF ('NORMAL', *x* <, *θ*, *λ*>)

Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

θ
is a numeric constant, variable, or expression that specifies a location parameter.

Default 0

λ
is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range λ > 0

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*.

$$PDF('NORMAL', x, \theta, \lambda) = \frac{1}{\lambda\sqrt{2\,\pi}}\ exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

Normal Mixture Distribution

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.

n

is a numeric constant, variable, or expression that specifies the integer number of mixtures.

Range *n* = 1, 2, ...

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, ...

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

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.

Pareto Distribution

PDF ('PARETO', *x*, *a* <, *k*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a numeric random variable.

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

k

is a numeric constant, variable, or expression that specifies a scale parameter.

Default 1

Range $k > 0$

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*.

$$PDF('PARETO', x, a, k) = \begin{cases} 0 & x < k \\ \frac{a}{k}\left(\frac{k}{x}\right)^{a+1} & x \geq k \end{cases}$$

Poisson Distribution

PDF ('POISSON', *n*, *m*)

Arguments

n

is a numeric constant, variable, or expression that specifies an integer random variable.

Range $n = 0, 1, \ldots$

m

is a numeric constant, variable, or expression that specifies a mean parameter.

Range $m > 0$

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.

T Distribution

PDF ('T', *t*, *df* <, *nc*>)

Arguments

t

is a numeric constant, variable, or expression that specifies a random variable.

df

is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

nc

is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

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 the following 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.

Tweedie Distribution

PDF ('TWEEDIE', *y*, *p* <, *μ*, *ϕ*>)

Arguments

y

is a numeric constant, variable, or expression that specifies a random variable.

Range *y* ≥0

Notes This argument is required.

When *y*>1, *y* is numeric. When *p*=1, *y* is an integer.

p

is a numeric constant, variable, or expression that specifies the power parameter.

Range *p* ≥1

Note This argument is required.

μ

is a numeric constant, variable, or expression that specifies the mean parameter.

Default 1

Range $\mu > 0$

ϕ

is a numeric constant, variable, or expression that specifies the dispersion parameter.

Default 1

Range $\phi > 0$

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 that case, 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.

Uniform Distribution

PDF ('UNIFORM', *x* <, *l*, *r*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

l

is a numeric constant, variable, or expression that specifies the left location parameter.

Default 0

r

is a numeric constant, variable, or expression that specifies the right location parameter.

Default 1

Range $r > l$

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.

$$PDF('UNIFORM', x, l, r) = \begin{cases} 0 & x < l \\ \frac{1}{r-l} & l \le x \le r \\ 0 & x > r \end{cases}$$

Wald (Inverse Gaussian) Distribution

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.

$$fx(x) = \left[\frac{\lambda}{2\pi x^3}\right]^{1/2} \exp\left\{-\frac{\lambda}{2\mu^2 x}(x-\mu)^2\right\}, \qquad x > 0$$

Weibull Distribution

PDF('WEIBULL', *x*, *a* <, *λ*>)

Arguments

x

is a numeric constant, variable, or expression that specifies a random variable.

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 $\lambda > 0$

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

```
data
one;

   a=pdf('BERN',
0, .25);

   b=pdf('BERN',
1, .25);

   c=pdf('BETA', 0.2, 3,
4);

   d=pdf('BINOM', 4, .5,
10);

   e=pdf('CAUCHY',
2);

   f=pdf('CHISQ', 11.264,
11);

   g=pdf('CONMAXPOI', .2,
2.3, .4);

   h=pdf('EXPO',
1);

   i=pdf('F', 3.32, 2,
3);

   j=pdf('GAMMA', 1,
3);
```

```
   k=pdf('GENPOISSON', 9,
1, .7);

   l=pdf('GEOMETRIC',
5, .3);

   m=pdf('HYPER', 2, 200, 50,
10);

   n=pdf('LAPLACE',
1);

   o=pdf('LOGISTIC',
1);

   p=pdf('LOGNORMAL',
1);

   q=pdf('NEGB', 1, .5,
2);

   r=pdf('NORMAL',
1.96);

   s=pdf('NORMALMIX',2.3, 3, .33, .33, .34, .5, 1.5, 2.5, .79, 1.6,
4.3);
   t=pdf('PARETO', 1,
1);

   u=pdf('POISSON', 2,
1);

   v=pdf('T', .9,
5);

   w=pdf('TWEEDIE', .8,
5);

   x=pdf('UNIFORM',
0.25);

   y=pdf('WALD', 1,
2);

   z=pdf('WEIBULL', 1,
2);

   put
_all_;

run;
```

These statements produce these results:

```
a=0.75 b=0.25 c=1.2288 d=0.205078125 e=0.0636619772 f=0.0816861868 g=0.0097732635
h=0.3678794412
i=0.0540269626 j=0.1839397206 k=0.0150130915 l=0.050421 m=0.2868505964
n=0.1839397206
o=0.1966119332 p=0.3989422804 q=0.25 r=0.0584409443 s=0.1165539644 t=1
u=0.1839397206
v=0.2419443436 w=0.7422908236 x=1 y=0.5641895835 z=0.7357588823
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “QUANTILE Function” on page 1293
- “SDF Function” on page 1376
- “SQUANTILE Function” on page 1404

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.

PEEKCLONG Function

Stores the contents of a memory address in a character variable on a 64-bit platform.

Categories: Binary Results
Special

Syntax

PEEKCLONG(*address* <, *length*>)

Required Argument

address
specifies a character constant, variable, or expression that contains the binary pointer address.

Optional Argument

length
is a numeric constant, variable, or expression that specifies the length of the character data.

Default 8

Range 1–32,767

Details

If you do not have access to the memory storage location that you are requesting, the PEEKCLONG function returns an `Invalid argument` error.

Comparisons

The PEEKCLONG function stores the contents of a memory address in a *character* variable.

The PEEKLONG function stores the contents of a memory address in a *numeric* variable. The function assumes that the input address refers to an integer in memory.

Example: Example for a 64-bit Platform

This example returns the pointer address for the character variable Z.

```
data _null_;
   x='ABCDE';
   y=addrlong(x);
   z=peekclong(y, 2);
   put z=;
run;
```

These statements produce this result:

```
z=AB
```

See Also

Functions:

- "PEEKLONG Function" on page 1216

PEEKLONG Function

Stores the contents of a memory address in a numeric variable on a 64-bit platform.

Category: Special

See: "PEEKCLONG Function" on page 1214

CAUTION: **The PEEKLONG functions can directly access memory addresses. Improper use of the PEEKLONG functions can cause SAS, and your operating system, to fail.** Use the PEEKLONG functions only to access information that is returned by one of the MODULE functions.

Syntax

PEEKLONG(*address* <, *length*>)

Required Argument

address
specifies a character constant, variable, or expression that contains the binary pointer address.

Optional Argument

length
is a numeric constant, variable, or expression that specifies the length of the character data.

Default 8 on 64-bit computers

Range 1–8 on 64-bit computers

Details

If you do not have access to the memory storage location that you are requesting, the PEEKLONG function returns an `Invalid argument` error.

Comparisons

The PEEKLONG function stores the contents of a memory address in a *numeric* variable. The function assumes that the input address refers to an integer in memory.

The PEEKCLONG function stores the contents of a memory address in a *character* variable. The function assumes that the input address refers to character data.

Usually, when you need to use one of the PEEKLONG functions, you use PEEKCLONG to access a character string.

Example: Example for a 64-bit Platform

This example returns the pointer address for the numeric variable Z.

```
data _null_;
   length y $4;
   y=put(1, IB4.);
   addry=addrlong(y);
   z=peeklong(addry, 4);
   put z=;
run;
```

SAS writes the following output to the log:

```
z=1
```

See Also

Functions:

- "PEEKCLONG Function" on page 1214

PERM Function

Computes the number of permutations of *n* items that are taken *r* at a time.

Category: Combinatorial

Syntax

PERM(*n* <, *r*>)

Required Argument

n

is an integer that represents the total number of elements from which the sample is chosen.

Optional Argument

r

is an integer value that represents the number of chosen elements. If *r* is omitted, the function returns the factorial of *n*.

Restriction $r \le n$

Details

The mathematical representation of the PERM function is given by the following equation:

$$PERM(n, r) = \frac{n!}{(n-r)!}$$

with $n \ge 0$, $r \ge 0$, 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 PERM function.

Example

```
data
one;

   x=perm(5,
1);


y=perm(5);

   z=perm(5,
2);

   put x= y=
z=;

run;
```

These statements produce these results:

```
x=5 y=120 z=20
```

See Also

Functions:

- "COMB Function" on page 445
- "FACT Function" on page 587
- "LPERM Function" on page 1090

PMT Function

Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.

Category: Financial

Syntax

PMT(*rate*, *number-of-periods*, *principal-amount*, <*future-amount*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

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.

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.

Example

```
data
one;



 /*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
*/


  Payment1=PMT(0.08/12, 10, 10000, 0,
0);

  Payment1=PMT(0.08/12, 10,
10000);

  put
Payment1=;



  /*If the same loan has beginning-of-period payments, then payment
can be computed as follows */


  Payment2=PMT(0.08/12, 10, 10000, 0,
1);

  put
Payment2=;



  /*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
*/



  Payment3=PMT(.01/12, 5,
5000);

  put
Payment3=;



  /*The payment for monthly periodic savings that accrue over 18 years
at a 6% nominal annual
    interest rate, and which accumulates $50,000 at the end of the 18
years, is computed as follows */
```

```
  Payment4=PMT(0.06/12, 216, 0, 50000,
0);

  put
Payment4=;



run;
```

These statements produce these results:

```
Payment1=1037.0320894
Payment2=1030.1643272
Payment3=1002.5013883
Payment4=-129.0811609
```

POINT Function

Locates an observation that is identified by the NOTE function.

Category: SAS File I/O

Syntax

POINT(*data-set-id*, *note-id*)

Required Arguments

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

note-id
is a numeric variable that specifies the identifier assigned to the observation by the NOTE function.

Details

POINT returns 0 if the operation was successful, ≠0 if it was not successful. POINT prepares the program to read from the SAS data set. The Data Set Data Vector is not updated until a read is done using FETCH or FETCHOBS.

Example

This example calls NOTE to obtain an observation ID for the most recently read observation of the SAS data set sashelp.cars. It calls POINT to point to that observation, and calls FETCH to return the observation marked by the pointer.

```
%macro
test(mydata,var);

 %let dsid=%sysfunc(open(&mydata,
i));

 %let rc=
%sysfunc(fetch(&dsid));

 %let noteid=
%sysfunc(note(&dsid));

 %let rc=%sysfunc(point(&dsid,
&noteid));

 %let rc=
%sysfunc(fetch(&dsid));

 %let name=
%sysfunc(varnum(&dsid,&var));

 %let val=
%sysfunc(getvarc(&dsid,&name));

  %put
&=val;

 %let rc=
%sysfunc(close(&dsid));

%mend
test;



%test(sashelp.cars,model)
```

These statements produce this result:

```
VAL=MDX
```

See Also

Functions:

- “DROPNOTE Function” on page 570

- “NOTE Function” on page 1151
- “OPEN Function” on page 1183

POISSON Function

Returns the probability from a Poisson distribution.

Category: Probability

See: “CDF Function” on page 392 , “PDF Function” on page 1193

Syntax

POISSON(*m*, *n*)

Required Arguments

m
is a numeric mean parameter.

Range $m \geq 0$

n
is an integer random variable.

Range $n \geq 0$

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

```
data
one;

   x=poisson(1,
2);
```

```
   put
x=;

run;
```

These statements produce this result:

```
x=0.9196986029
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PPMT Function

Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.

Category: Financial

Syntax

PPMT(*rate*, *period*, *number-of-periods*, *principal-amount*, <*future-amount*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

period
specifies the payment period for which the principal payment is computed. *Period* must be a positive integer value that is less than or equal to the value of *number-of-periods*.

number-of-periods
specifies the number of payment periods. *Number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

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.

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.

Example

```
data
one;


/*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
*/

   PrincipalPayment=PPMT(0.1/12, 1, 24,
2010);

   put
PrincipalPayment=;




/*The principal payment for a 3-year, $20,000 loan with beginning-of-
month payments
is computed as the following statement.
This computation returns a value of 64.321037613 as the principal
that was paid with the first payment.
*/
```

```
   PrincipalPayment2=PPMT(0.1/12, 1, 36, 2010, 0,
1);

   put
PrincipalPayment2=;

/* 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 the following
statement.
This computation returns value of -71.56222304 as the
principal that was paid with the first payment.
*/

   PrincipalPayment3=PPMT(0.1/12, 1, 36, 2010, 5000,
0);

   put PrincipalPayment3=;

run;
```

These statements produce these results:

```
PrincipalPayment=76.001301938
PrincipalPayment2=64.321037613
PrincipalPayment3=-71.56222304
```

PROBBETA Function

Returns the probability from a beta distribution.

Category: Probability

See: “CDF Function” on page 392, “PDF Function” on page 1193

Syntax

PROBBETA(*x*, *a*, *b*)

Required Arguments

x

is a numeric constant, variable, or expression that specifies the value of a random variable.

Range $0 \leq x \leq 1$

a

is a numeric constant, variable, or expression that specifies the value of this shape parameter.

Range $a > 0$

b

is a numeric constant, variable, or expression that specifies the value of this shape parameter.

Range $b > 0$

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

```
data
one;

   x=probbeta(.2, 3,
4);

   put
x=;

run;
```

These statements produce this result:

```
x=0.09888
```

See Also

Functions:

- “CDF Function” on page 392

- "LOGCDF Function" on page 1081
- "LOGPDF Function" on page 1084
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "SDF Function" on page 1376

PROBBNML Function

Returns the probability from a binomial distribution.

Category: Probability

See: "CDF Function" on page 392 , "PDF Function" on page 1193

Syntax

PROBBNML(*p*, *n*, *m*)

Required Arguments

p

is a numeric constant, variable, or expression that specifies a probability of success.

Range $0 \le p \le 1$

n

is a numeric constant, variable, or expression that specifies an integer number of independent Bernoulli trials.

Range $n > 0$

m

is a numeric constant, variable, or expression that specifies an integer number of successes.

Range $0 \le m \le n$

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success *p* and number of trials *n*, produces *m* or fewer successes. To compute the probability of exactly *m* successes, compute

the difference of two probabilities from the binomial distribution for *m* and *m*−1 successes.

Example

```
data
one;

   x=probbnml(0.5, 10,
4);

   put
x=;

run;
```

These statements produce this result:

```
x=0.376953125
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PROBBNRM Function

Returns a probability from a bivariate normal distribution.

Category: Probability

Syntax

PROBBNRM(*x*, *y*, *r*)

Required Arguments

x

is a numeric constant, variable, or expression that specifies the value of random variable *x*.

y

is a numeric constant, variable, or expression that specifies the value of random variable *y*.

r

is a numeric constant, variable, or expression that specifies the value of the correlation coefficient.

Range $-1 \le r \le 1$

Details

The PROBBNRM function returns the probability that an observation from a standardized bivariate normal distribution with mean 0, variance 1, and 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

```
data
one;

   x=probbnrm(.4,
-.3, .2);

   put
x=;

run;
```

These statements produce this result:

```
x=0.2783183345
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PROBCHI Function

Returns the probability from a chi-square distribution.

Category: Probability

See: “CDF Function” on page 392, “PDF Function” on page 1193

Syntax

PROBCHI(*x*, *df* <, *nc*>)

Required Arguments

x
is a numeric constant, variable, or expression that specifies the value of a random variable.

Range $x \geq 0$

df
is a numeric constant, variable, or expression that specifies the degrees of freedom parameter.

Range $df > 0$

Optional Argument

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range $nc \geq 0$

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

```
data
one;

   x=probchi(11.264,
11);

   put
x=;

run;
```

These statements produce this result:

```
x=0.5785813293
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PROBF Function

Returns the probability from an *F* distribution.

Category: Probability

See: “CDF Function” on page 392, “PDF Function” on page 1193

Syntax

PROBF(*x*, *ndf*, *ddf* <, *nc*>)

Required Arguments

x
is a numeric constant, variable, or expression that specifies the value of a random variable.

Range $x \geq 0$

ndf
is a numeric numerator that specifies the numerator degrees of freedom parameter.

Range $ndf > 0$

ddf
is a numeric denominator that specifies the denominator degrees of freedom parameter.

Range $ddf > 0$

Optional Argument

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

Range $nc \geq 0$

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

```
p=1-probf(x,ndf,ddf);
```

Example

```
data
one;

   x=probf(3.32, 2,
3);

   put
x=;

run;
```

These statements produce this result:

```
x=0.8263933602
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PROBGAM Function

Returns the probability from a gamma distribution.

Category: Probability

See: “CDF Function” on page 392, “PDF Function” on page 1193

Syntax

PROBGAM(*x*, *a*)

Required Arguments

x

is a numeric constant, variable, or expression that specifies the value of a random variable.

Range $x \geq 0$

a

is a numeric constant, variable, or expression that specifies the shape parameter.

Range $a > 0$

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

```
data
one;

   x=probgam(1,
3);

   put
x=;

run;
```

These statements produce this result:

```
x=0.0803013971
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193

- “SDF Function” on page 1376

PROBHYPR Function

Returns the probability from a hypergeometric distribution.

Category: Probability

See: “CDF Function” on page 392, “PDF Function” on page 1193

Syntax

PROBHYPR(*N*, *K*, *n*, *x* <, *r*>)

Required Arguments

N

is a numeric constant, variable, or expression that specifies an integer population size parameter.

Range $N \geq 1$

K

is a numeric constant, variable, or expression that specifies an integer number of items in the category of interest.

Range $0 \leq K \leq N$

n

is a numeric constant, variable, or expression that specifies an integer sample size parameter.

Range $0 \leq n \leq N$

x

is a numeric constant, variable, or expression that specifies an integer random variable.

Range $max(0, K + n-N) \leq x \leq min(K,n)$

Optional Argument

r

is a numeric constant, variable, or expression that specifies an optional numeric odds ratio parameter.

Range $r \geq 0$

Details

The PROBHYPR function returns the probability that an observation from an extended or noncentral 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 central hypergeometric distribution.

Example

```
data
one;

   x=probhypr(200, 50, 10,
2);

   put
x=;

run;
```

These statements produce this result:

```
x=0.5236734081
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PROBIT Function

Returns a quantile from the standard normal distribution.

Category: Quantile

Syntax

PROBIT(*p*)

Required Argument

p
is a numeric probability.

Range $0 < p < 1$

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

```
data
one;


x=probit(.025);


y=probit(1.e-7);

   put x=;
   put
y=;

run;
```

These statements produce these results:

```
x=-1.959963985
y=-5.199337582
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PROBMC Function

Returns a probability or a quantile from various distributions for multiple comparisons of means.

Category: Probability

Syntax

PROBMC(*distribution*, *q*, *prob*, *df*, *nparms* <, *parameters*>)

Required 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

q
is a numeric constant, variable, or expression that specifies the quantile from the distribution.

Restriction Either *q* or *prob* can be specified, but not both.

prob
is a numeric constant, variable, or expression that specifies the left probability from the distribution.

Restriction Either *prob* or *q* can be specified, but not both.

df
is a numeric constant, variable, or expression that specifies the degrees of freedom.

Note: A missing value is interpreted as an infinite value.

nparms
is a numeric constant, variable, or expression that specifies the number of treatments.

Note: For DUNNETT1 and DUNNETT2, the control group is not counted.

Optional Argument

parameters
is an optional set of *nparms* parameters that must be specified to handle 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.

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 the 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*.

$$d\mu_\nu(x) = \frac{\frac{\nu}{\nu 2}}{\Gamma\left(\frac{\nu}{2}\right)2^{\frac{\nu}{2}-1}} x^{\nu-1}\varepsilon^{-\frac{\nu x^2}{2}}dx$$

$$\phi(x) = \frac{1}{\sqrt{2\,\pi}}\varepsilon^{-\frac{x^2}{2}}$$

$$\Phi(x) = \int_{-\infty}^{x} \phi(u)du$$

Computing the Analysis of Means

Analysis of Means (ANOM) applies to data that is organized as *k* (Gaussian) samples; the ith sample is 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 $\mathbb{T}$ vector, with ν degrees of freedom, and an associated correlation matrix $\rho_{ij} = -\alpha_i\alpha_j$. This equation can be written as

$$\begin{aligned} prob &= r(|t_1| < h, |t_2| < h, ..., |t_k| < h) \\ &= \int_0^\infty \left\{ \int_0^\infty \prod_{j=0}^{j=k} g(sh, y, \alpha_j)\phi(y)dy \right\} d\mu_\nu(s) \end{aligned}$$

The following relationship applies to the preceding equation:

$$g(sh, y, \alpha_j) = \Phi\left(\frac{sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

$\Gamma(.)$, $\phi(.)$, and $\Phi(.)$, are the gamma function, the density, and the CDF from the standard normal distribution, respectively.

For $\nu = \infty$, the distribution reduces to:

$$r(|t_1| < h, |t_2| < h, ..., |t_k| < h) = \int_0^\infty \prod_{j=0}^{j=k} g(h, y, \alpha_j)\phi(y)dy$$

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:

$$r(|t_1| < h, |t_2| < h, ..., |t_n| < h) = \int_0^\infty f(h, y, \rho)^n \phi(y) dy$$

The following relationship applies to the preceding equation:

$$f(h, y, \rho) = \Phi\left(\frac{h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right) - \Phi\left(\frac{-h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right)$$

and $\rho = \frac{1}{n-1}$

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 that denote the alpha values from the first equation of this distribution. See "Computing the Analysis of Means" on page 1241.

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 λ_1, ..., λ_k, the value of *nparms* is set to *k*, and the value of *df* is set to *v*.

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + qx}{\sqrt{1-\lambda_i^2}}\right) dy\, du_\nu(x)$$

- 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*.

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y)[\Phi(y + \sqrt{2qx})]^k dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are λ_1, ..., λ_k, the value of *nparms* is set to *k*, and the value of *df* is set to missing.

$$prob = \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + q}{\sqrt{1-\lambda_i^2}}\right) dy$$

- 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.

$$prob = \int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2q})]^k dy$$

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*.

$$prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + qx}{\sqrt{1 - \lambda_i^2}} \right) - \Phi\left(\frac{\lambda_i y - qx}{\sqrt{1 - \lambda_i^2}} \right) \right] dy \, du_\nu(x)$$

- 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*.

$$prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2qx}) - \Phi(y - \sqrt{2qx})]^k dy \, du_\nu(x)$$

- 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.

$$prob = \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + q}{\sqrt{1 - \lambda_i^2}} \right) - \Phi\left(\frac{\lambda_i y - q}{\sqrt{1 - \lambda_i^2}} \right) \right] dy$$

- 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.

$$prob = \int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2q}) - \Phi(y - \sqrt{2q})]^k dy$$

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 1243.

The Studentized Range

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes.

- 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*.

$$prob = \int_0^\infty \int_{-\infty}^\infty k\phi(y)[\Phi(y) - \Phi(y - qx)]^{k-1} dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\sigma_1, \ldots, \sigma_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing.

$$prob = \int_{-\infty}^\infty \sum_{j=1}^k \left\{ \prod_{i=1}^k \left[\Phi\left(\frac{y}{\sigma_i}\right) - \Phi\left(\frac{y-q}{\sigma_i}\right) \right] \right\} \phi\left(\frac{y}{\sigma_j}\right) \frac{1}{\sigma_j} dy$$

- 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.

$$prob = \int_{-\infty}^\infty k\phi(y)[\Phi(y) - \Phi(y - q)]^{k-1} dy$$

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 $\sigma_1, \ldots, \sigma_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*.

$$prob = \int_0^\infty \prod_{i=1}^k \left[2\Phi\left(\frac{qx}{\sigma_i}\right) - 1 \right] d\mu_\nu(x)$$

- 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*.

$$prob = \int_0^{\infty} [2\Phi(qx) - 1]^k d\mu_\nu(x)$$

- 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.

$$prob = \prod_{i=1}^{k} \left[2\Phi\left(\frac{q}{\sigma_i}\right) - 1\right]$$

- 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.

$$prob = [2\Phi(q) - 1]^k$$

Williams' Test

PROBMC computes the probabilities or quantiles from the distribution defined in Williams (1971, 1972). (See "References" on page 1609 .) It 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 this equation

$$Y_k = \frac{X_1 + X_2 + \ldots + X_k}{k}$$

It is required to compute the distribution of this equation

$$(Y_k - Z)/S$$

Arguments

Y_k
is as defined previously.

Z
is an N(0,1) independent random variable.

S
is such that $\frac{1}{2}\nu S^2$ is a χ^2 variable with ν degrees of freedom.

As described in Williams (1971) (see "References" on page 1609), the full computation is extremely lengthy and is carried out in three stages.

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 as

$$U_i = \frac{X_1 + X_2 + ... + X_i}{i},\ i = 1, 2, ..., k$$

You can write a recursive expression for the probability of $Y_k > d$, with d as any real number.

$$\begin{aligned}\Pr(Y_k > d) &= \Pr(U_1 > d) \\ &\quad +\Pr(U_2 > d, U_1 < d) \\ &\quad +\Pr(U_3 > d, U_2 < d, U_1 < d) \\ &\quad +\ldots \\ &\quad +\Pr(U_k > d, U_{k-1} < d, \ldots, U_1 < d) \\ &\quad = \Pr(Y_{k-1} > d) + \Pr(X_k + (k-1)U_{k-1} > kd)\end{aligned}$$

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

This example shows how to compute probabilities.

```
data probs;
   array par{5};
      par{1}=.5;
      par{2}=.51;
      par{3}=.55;
      par{4}=.45;
      par{5}=.2;
   df=40;
   q=1;
   do test="dunnett1","dunnett2", "maxmod";
      prob=probmc(test, q, ., df, 5, of par1-par5);
      put test $10. df q e18.13 prob e18.13;
   end;
run;
```

These statements produce these results:

```
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

```
data _null_;
   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=;
run;
```

These statements produce these results:

```
q1=2.7895061016
q2=2.4549961967
q3=2.4549961967
q4=2.4532319994
```

Example 3: Comparing Means

This example shows how to compare group means to find where the significant differences lie. The data for this example is taken from a paper by Duncan (1955), and can also be found in Hochberg and Tamhane (1987). (See the References section at the end of this function.)

The following values are the group means:

- 49.6
- 71.2
- 67.6

- 61.5
- 71.3
- 58.1
- 61.0

For this data, the mean square error is s^2=79.64 (s=8.924) with v = 30.

```
data duncan;
   array tr{7}$;
   array mu{7};
   n=7;
   do i=1 to n;
      input tr{i} $1. mu{i};
   end;
   input df s alpha;
   prob=1-alpha;
      /* compute the interval */
   x=probmc("RANGE", ., prob, df, 7);
   w=x * s / sqrt(6);
      /* compare the means */
   do i=1 to n;
      do j=i + 1 to n;
         dmean = abs(mu{i} - mu{j});
         if dmean >= w then do;
            put tr{i} tr{j} dmean;
         end;
      end;
   end;
   datalines;
A 49.6
B 71.2
C 67.6
D 61.5
E 71.3
F 58.1
G 61.0
 30 8.924 .05
;
run;
```

These statements produce these results:

```
A B 21.6
A C 18
A E 21.7
```

Example 4

```
data _null_;
   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=;
run;
```

These statements produce these results:

```
q1=4.1022397989
q2=4.7888626338
```

Example 5: Computing Confidence Intervals

This example shows how to compute 95% one-sided and two-sided confidence intervals of Dunnett's test. This example and the data come from Dunnett (1955), and can also be found in Hochberg and Tamhane (1987). (See the References section at the end of this function.) The data are blood count measurements on three groups of animals. As shown in the following table, the third group serves as the control, and the first two groups were treated with different drugs. The numbers of animals in these three groups are unequal.

Treatment Group:	Drug A	Drug B	Control
	9.76	12.80	7.40
	8.80	9.68	8.50
	7.68	12.16	7.20
	9.36	9.20	8.24
		10.55	9.84
			8.32
Group Mean	8.90	10.88	8.25
n	4	5	6

The mean square error s^2=1.3805 (s=1.175) with v = 12.

```
data a;
   array drug{3}$;
   array count{3};
   array mu{3};
   array lambda{2};
   array delta{2};
   array left{2};
   array right{2};
      /* input the table */
   do i=1 to 3;
      input drug{i} count{i} mu{i};
   end;
      /* input the alpha level,     */
      /* the degrees of freedom,    */
      /* and the mean square error */
   input alpha df s;
```

```
      /* from the sample size, */
      /* compute the lambdas   */
   do i=1 to 2;
      lambda{i}=sqrt(count{i}/
        (count{i} + count{3}));
   end;
      /* run the one-sided Dunnett's test */
   test="dunnett1";
      x=probmc(test, ., 1 - alpha, df,
                2, of lambda1-lambda2);
      do i=1 to 2;
         delta{i}=x * s *
            sqrt(1/count{i} + 1/count{3});
         left{i}=mu{i} - mu{3} - delta{i};
      end;
   put test $10. x left{1} left{2};
      /* run the two-sided Dunnett's test */
   test="dunnett2";
      x=probmc(test, ., 1 - alpha, df,
                2, of lambda1-lambda2);
      do i=1 to 2;
         delta{i}=x * s *
            sqrt(1/count{i} + 1/count{3});
         left{i}=mu{i} - mu{3} - delta{i};
         right{i}=mu{i} - mu{3} + delta{i};
      end;
   put test $10. left{1} right{1};
   put test $10. left{2} right{2};
   datalines;
A 4 8.90
B 5 10.88
C 6 8.25
0.05 12 1.175
;
run;
```

These statements produce these results:

```
DUNNETT1  2.1210448226 -0.958726041 1.1208812046
DUNNETT2  -1.256408109 2.5564081095
DUNNETT2  0.8416306717 4.4183693283
```

Example 6: Computing Williams' Test

In the following example, 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

Treatment	Mean
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.

Here are the maximum likelihood estimates under the alternative hypothesis:

- $M_0=M_1=M_2 = X_{0,1,2}$=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 table.

```
data
one;

   prob=probmc("williams", 2.6, ., 42,
6);

   quant5=probmc("williams", ., .95, 42,
6);

   quant1=probmc("williams", ., .99, 42,
6);

   put prob=;
   put quant5=;
```

```
   put
quant1=;

run;
```

These statements produce these results:

```
prob=0.9924466872
quant5=1.806562536
quant1=2.490908273
```

See Also

Functions:

- "CDF Function" on page 392
- "LOGCDF Function" on page 1081
- "LOGPDF Function" on page 1084
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "SDF Function" on page 1376

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 Equi-correlated Multivariate Non-central t Distribution." *Communications in Statistics: Part B - Simulation and Computation* 10: 41–50.

Nelson, P. R. 1982. "Exact Critical Points for the Analysis of Means." *Communications in Statistics: Part A - Theory and Methods* 11: 699–709.

Nelson, P. R. 1982a. "An Approximation for the Complex Normal Probability Integral." *BIT* 22(1): 94–100.

Nelson, P. R. 1988. "Application of the Analysis of Means." *Proceedings of the SAS Global Forum 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.

Category: Probability

Returned data type: DOUBLE

Syntax

PROBMED(*n*, *x*)

Arguments

n

specifies the sample size.

Data type Double

x

specifies 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:

$$\Pr\left[X_{((n+1)/2)} \le x\right] = \mathrm{I}_{\Phi(x)}\left(\frac{n+1}{2}, \frac{n+1}{2}\right)$$

In the equation, $\mathrm{I}_{\Phi(x)}$ is the incomplete beta function, which is defined as follows:

$$\mathrm{I}_p(a,b) = \frac{\mathrm{I}}{\mathrm{B}(a,b)} \int_0^p t^{a-1}(1-t)^{b-1} dt$$

In the equation, $\mathrm{B}(\mathrm{a},\mathrm{b}) = \Gamma(\mathrm{a})\Gamma(\mathrm{b})/\Gamma(\mathrm{a}+\mathrm{b})$ is the beta function, and $\Gamma(\cdot)$ is the gamma function.

If *n* is even, the PROBMED function performs the following calculation:

$$\Pr\left[\frac{X_{(n/2)} + X_{((n/2)+1)}}{2} \leq x\right] =$$

$$\frac{2}{\mathrm{B}\left(\frac{n}{2}, \frac{n}{2}\right)} \int_{-\infty}^{x} \{[1 - \Phi(u)]^{n/2} - [1 - \Phi(2x - u)]^{n/2}\} [\Phi(u)]^{(n/2)-1} \phi(u) du$$

In this equation, $\mathrm{B}(n/2,\ n/2) = [\Gamma(n/2)]^2/\Gamma(n)$, and $\Phi(\cdot)$ and $\phi(\cdot)$ are the standard normal cumulative distribution function and density function, respectively.

Example

This example demonstrates the PROBMED functionality:

```
data _null_;
   b=probmed(5,-0.1);
   put b=;
run;
```

The preceding statements produce these results:

```
b=0.4256380897
```

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.

Category: Probability

See: "CDF Function" on page 392

Syntax

PROBNEGB(*p*, *n*, *m*)

Required Arguments

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

n

is a numeric constant, variable, or expression that specifies an integer number of successes.

Range $n \ge 1$

m

is a numeric constant, variable, or expression that specifies an integer number of failures.

Range $m \ge 0$

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*. This result is the probability of *m* failures occurring before the *n*th success.

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

```
data
one;

 x=probnegb(0.5, 2,
1);

 put
x=;

run;
```

These statements produce this result:

```
x=0.5
```

See Also

Functions:

- “CDF Function” on page 392

- "LOGCDF Function" on page 1081
- "LOGPDF Function" on page 1084
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "SDF Function" on page 1376

PROBNORM Function

Returns the probability from the standard normal distribution.

Category: Probability

See: "CDF Function" on page 392

Syntax

PROBNORM(*x*)

Required Argument

x
is a numeric constant, variable, or expression that specifies a random variable.

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

```
data
one;


x=probnorm(1.96);

   put
x=;
```

```
run;
```

These statements produce this result:

```
x=0.9750021049
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “SDF Function” on page 1376

PROBT Function

Returns the probability from a *t* distribution.

Category: Probability

See: “CDF Function” on page 392 , “PDF Function” on page 1193

Syntax

PROBT(*x*, *df* <, *nc*>)

Required Arguments

x
is a numeric constant, variable, or expression that specifies a random variable.

df
is a numeric constant, variable, or expression that specifies the degrees of freedom.

Range *df* > 0

Optional Argument

nc
is a numeric constant, variable, or expression that specifies an optional noncentrality parameter.

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

```
p=(1-probt(abs(x),df))*2;
```

Example

```
data
one;

   x=probt(0.9,
5);

   put
x=;

run;
```

These statements produce this result:

```
x=0.7953143998
```

See Also

Functions:

- "CDF Function" on page 392
- "LOGCDF Function" on page 1081
- "LOGPDF Function" on page 1084
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "SDF Function" on page 1376

PROPCASE Function

Converts all words in an argument to proper case.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

PROPCASE(*argument* <, *delimiters*>)

Required Argument

argument
specifies a character constant, variable, or expression.

Optional Argument

delimiter
specifies one or more delimiters that are enclosed in quotation marks. The default delimiters are blank, forward slash, hyphen, open parenthesis, period, and tab.

Tip If you use this argument, then the default delimiters, including the blank, are no longer in effect.

Details

Length of Returned Variable

In a DATA step, if the PROPCASE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument that is passed to PROPCASE.

The Basics

The PROPCASE function copies a character argument and converts all uppercase letters to lowercase letters. It then converts to uppercase the first character of a word that is preceded by a blank, forward slash, hyphen, open parenthesis, period, or tab. PROPCASE returns the value that is altered.

If you use the second argument, then the default delimiters are no longer in effect.

The results of the PROPCASE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*) and indirectly on the ENCODING and LOCALE system options.

Comparisons

The PROPCASE function is designed for Latin-based languages. It operates by capitalizing the first letter of each word, with words being defined by delimiters such as spaces.

The KPROPCASE function is designed for double-byte languages like Simplified Chinese, Traditional Chinese, Japanese, and Korean. These languages also require proper case, but not in the form of initial capital as in Latin languages. Instead, they involve other types of conversions, such as full-width and half-width transformations. Additionally, double-byte languages do not use delimiters to separate characters into words as do Latin languages.

Examples

Example 1: Changing the Case of Words

The following example shows how PROPCASE handles the case of words:

```
data _null_;
   input place $ 1-40;
   name=propcase(place);
   put name;
   datalines;
INTRODUCTION TO THE SCIENCE OF ASTRONOMY
VIRGIN ISLANDS (U.S.)
SAINT KITTS/NEVIS
WINSTON-SALEM, N.C.
;
run;
```

SAS writes the following output to the log:

```
Introduction To The Science Of Astronomy
Virgin Islands (U.S.)
Saint Kitts/Nevis
Winston-Salem, N.C.
```

Example 2: Using a Second Argument with PROPCASE

The following example uses a blank, a hyphen, and a single quotation mark as the second argument so that names such as O'Keeffe and Burne-Jones are written correctly.

```
data names;
   infile datalines dlm='#';
```

```
   input CommonName : $20. CapsName : $20.;
   PropcaseName=propcase(capsname, " -'");
   datalines;
Delacroix, Eugene# EUGENE DELACROIX
O'Keeffe, Georgia# GEORGIA O'KEEFFE
Rockwell, Norman# NORMAN ROCKWELL
Burne-Jones, Edward# EDWARD BURNE-JONES
;
proc print data=names noobs;
   title 'Names of Artists';
run;
```

Output 3.53 *Output Showing the Results of Using PROPCASE with a Second Argument*

Names of Artists

CommonName	CapsName	PropcaseName
Delacroix, Eugene	EUGENE DELACROIX	Eugene Delacroix
O'Keeffe, Georgia	GEORGIA O'KEEFFE	Georgia O'Keeffe
Rockwell, Norman	NORMAN ROCKWELL	Norman Rockwell
Burne-Jones, Edward	EDWARD BURNE-JONES	Edward Burne-Jones

See Also

Functions:

- “LOWCASE Function” on page 1089
- “UPCASE Function” on page 1485

PRXCHANGE Function

Performs a pattern-matching replacement.

Category: Character String Matching

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8).

Note: SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXCHANGE(*perl-regular-expression* | *regular-expression-id*, *times*, *source*)

Required Arguments

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

You can see lists of PRX metacharacters in "Tables of Perl Regular Expression (PRX) Metacharacters" on page 1597. For a complete list of metacharacters, see the Perl documentation.

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.

times
is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern.

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.

Details

The Basics

If you use *regular-expression-id*, the PRXCHANGE function searches the variable *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 variable *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.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 53.

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 DATA step, 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.

Quantifiers are used when a portion of a pattern needs to match a certain number (or numbers) of times. If there is not a quantifier the number of times to match is one.The maximun value for a quantifier is 2147483646 (which is 2^31) -1).

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" on page 1597. You can find a complete list of metacharacters on the Perl website.

Comparisons

The PRXCHANGE function is similar to the CALL PRXCHANGE routine. The difference is that the function returns the value of the pattern-matching replacement as a return argument instead of as one of its parameters.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Examples

Example 1: Changing the Order of First and Last Names By Using the DATA Step

The following example uses the DATA step to change the order of first and last names.

```
   /* Create a data set that contains a list of names. */
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
```

```
;
   /* Reverse last and first names with a DATA step. */
data names;
   set ReversedNames;
   name=prxchange('s/(\w+), (\w+)/$2 $1/', -1, name);
run;
proc print data=names;
run;
```

Output 3.54 *Results from the DATA Step*

The SAS System

Obs	name
1	Fred Jones
2	Kate Kavich
3	Ron Turley
4	Yolanda Dulix

Example 2: Changing the Order of First and Last Names By Using PROC SQL

The following example uses PROC SQL to change the order of first and last names.

```
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;
proc sql;
   create table names as
   select prxchange('s/(\w+), (\w+)/$2 $1/', -1, name) as name
   from ReversedNames;
quit;
proc print data=names;
run;
```

Output 3.55 *Results from PROC SQL*

The SAS System

Obs	name
1	Fred Jones
2	Kate Kavich
3	Ron Turley
4	Yolanda Dulix

Example 3: Matching Rows That Have the Same Name

The following example compares the names in two data sets, and writes those names that are common to both data sets.

```
data names;
   input name & $32.;
   datalines;
Ron Turley
Judy Donnelly
Kate Kavich
Tully Sanchez
;
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;
proc sql;
   create table NewNames as
   select a.name from names as a, ReversedNames as b
   where a.name=prxchange('s/(\w+), (\w+)/$2 $1/', -1, b.name);
quit;
proc print data=NewNames;
run;
```

Output 3.56 *Results from Matching Rows That Have the Same Names*

The SAS System

Obs	name
1	Ron Turley
2	Kate Kavich

Example 4: Changing Lowercase Text to Uppercase

The following example uses the \U, \L, and \E metacharacters to change the case of a string of text. Case modifications do not nest. In this example, note that "bear" does not convert to uppercase letters because the \E metacharacter ends all case modifications.

```
data _null_;
   length txt $32;
   txt=prxchange ('s/(big)(black)(bear)/\U$1\L$2\E$3/', 1,
'bigblackbear');
   put txt=;
run;
```

SAS writes the following output to the log:

```
txt=BIGblackbear
```

Example 5: Changing a Matched Pattern to a Fixed Value

This example locates a pattern in a variable and replaces the variable with a predefined value. The example uses a DATA step to find phone numbers and replace them with an informational message.

```
   /* Create data set that contains confidential information. */
data a;
   input text $80.;
   datalines;
The phone number for Ed is (801)443-9876 but not until tonight.
He can be reached at (910)998-8762 tomorrow for testing purposes.
;
run;
   /* Locate confidential phone numbers and replace them with message
*/
   /* indicating that they have been removed.
*/
data b;
   set a;
   text=prxchange('s/\([2-9]\d\d\) ?[2-9]\d\d-\d\d\d\d/*PHONE NUMBER
      REMOVED*/', -1, text);
run;
proc print data=b;
run;
```

Output 3.57 *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 1267
- “PRXPAREN Function” on page 1272
- “PRXPARSE Function” on page 1274
- “PRXPOSN Function” on page 1276

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXPOSN Routine” on page 289
- “CALL PRXSUBSTR Routine” on page 292

PRXMATCH Function

Searches for a pattern match and returns the position at which the pattern is found.

Category: Character String Matching

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8).

Note: SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXMATCH(*regular-expression-id | perl-regular-expression*, *source*)

Required Arguments

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.

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

You can see lists of PRX metacharacters in "Tables of Perl Regular Expression (PRX) Metacharacters" on page 1597. For a complete list of metacharacters, see the Perl documentation.

source
specifies a character constant, variable, or expression that you want to search.

Details

The Basics

If you use *regular-expression-id*, then the PRXMATCH function searches *source* with the *regular-expression-id* that is returned by PRXPARSE, and returns the position at which the string begins. If there is no match, PRXMATCH returns a zero.

If you use *perl-regular-expression*, PRXMATCH searches *source* with the *perl-regular-expression*, and you do not need to call PRXPARSE.

You can use PRXMATCH with a Perl regular expression in a WHERE clause and in PROC SQL. For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 53.

Quantifiers are used when a portion of a pattern needs to match a certain number (or numbers) of times. If there is not a quantifier the number of times to match is one.The maximun value for a quantifier is 2147483646 (which is 2^31) -1). See "Tables of Perl Regular Expression (PRX) Metacharacters".

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.

Note: The compile-once behavior occurs when you use PRXMATCH in a DATA step, in a WHERE clause, or in PROC SQL. For all other uses, the *perl-regular-expression* is recompiled for each call to PRXMATCH.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Examples

Example 1: Finding the Position of a Substring By Using PRXPARSE

The following example searches a string for a substring, and returns its position in the string.

```
/* For release 9.0: the following example makes a call to PRXPARSE. */
  /* For release 9.1, no call is required. */

   data _null_;
         /* 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=;
   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 example 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_;
         /* Use PRXMATCH to find the position of the pattern match. */
      position=prxmatch('/world/', 'Hello world!');
      put position=;
   run;
```

SAS writes the following output to the log:

```
position=7
```

Example 3: Finding the Position of a Substring in a String: A Complex Example

The following example uses several Perl regular expression functions and a CALL routine to find the position of a substring in a string.

```
   data _null_;
      if _N_=1 then
      do;
         retain PerlExpression;
         pattern="/(\d+):(\d\d)(?:\.(\d+))?/";
         PerlExpression=prxparse(pattern);
      end;

      array match[3] $ 8;
```

```
   input minsec $80.;
   position=prxmatch(PerlExpression, minsec);
   if position ^= 0 then
   do;
      do i=1 to prxparen(PerlExpression);
         call prxposn(PerlExpression, i, start, length);
         if start ^= 0 then
            match[i]=substr(minsec, start, length);
      end;
      put match[1] "minutes, " match[2] "seconds" @;
      if ^missing(match[3]) then
         put ", " match[3] "milliseconds";
   end;
   datalines;
14:56.456
45:32
;
run;
```

SAS writes the following output to the log:

```
14 minutes, 56 seconds, 456 milliseconds
45 minutes, 32 seconds
```

Example 4: Filtering a Data Set by ZIP Code

The following example uses a DATA step to search each observation in a data set for a nine-digit ZIP code, and writes those observations to the data set ZipPlus4.

```
data ZipCodes;
   input name: $16. zip:$10.;
   datalines;
Johnathan 32523-2343
Seth 85030
Kim 39204
Samuel 93849-3843
;
   /* Extract ZIP+4 ZIP codes with the DATA step. */
data ZipPlus4;
   set ZipCodes;
   where prxmatch('/\d{5}-\d{4}/', zip);
run;
proc print data=ZipPlus4;
run;
```

Output 3.58 *ZIP code Output from the DATA Step*

The SAS System

Obs	name	zip
1	Johnathan	32523-2343
2	Samuel	93849-3843

Example 5: Extracting a ZIP code By Using PROC SQL

The following example searches each observation in a data set for a nine-digit ZIP code, and writes those observations to the data set ZipPlus4.

```
data ZipCodes;
   input name: $16. zip:$10.;
   datalines;
Johnathan 32523-2343
Seth 85030
Kim 39204
Samuel 93849-3843
;
   /* Extract ZIP+4 ZIP codes with PROC SQL. */
proc sql;
   create table ZipPlus4 as
   select * from ZipCodes
   where prxmatch('/\d{5}-\d{4}/', zip);
run;
proc print data=ZipPlus4;
run;
```

Output 3.59 *ZIP code Output from PROC SQL*

The SAS System

Obs	name	zip
1	Johnathan	32523-2343
2	Samuel	93849-3843

See Also

Functions:

- “PRXCHANGE Function” on page 1261
- “PRXPAREN Function” on page 1272
- “PRXPARSE Function” on page 1274
- “PRXPOSN Function” on page 1276

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXPOSN Routine” on page 289

- "CALL PRXSUBSTR Routine" on page 292

PRXPAREN Function

Returns the last bracket match for which there is a match in a pattern.

Category: Character String Matching

Restrictions: Use with the PRXPARSE function.

This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8).

Note: SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXPAREN(*regular-expression-id*)

Required Argument

regular-expression-id
specifies a numeric variable with a value that is an identification number that is returned by the PRXPARSE function.

Details

The PRXPAREN function is useful in finding the largest capture-buffer number that can be passed to the CALL PRXPOSN routine, or in identifying which part of a pattern matched.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 53.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Example

The following example uses Perl regular expressions and writes the results to the SAS log.

```
data _null_;
   ExpressionID=prxparse('/(magazine)|(book)|(newspaper)/');
   position=prxmatch(ExpressionID, 'find book here');
   if position then paren=prxparen(ExpressionID);
   put 'Matched paren ' paren;
   position=prxmatch(ExpressionID, 'find magazine here');
   if position then paren=prxparen(ExpressionID);
   put 'Matched paren ' paren;
   position=prxmatch(ExpressionID, 'find newspaper here');
   if position then paren=prxparen(ExpressionID);
   put 'Matched paren ' paren;
run;
```

SAS writes the following output to the log:

```
Matched paren 2
   Matched paren 1
   Matched paren 3
```

See Also

Functions:

- “PRXCHANGE Function” on page 1261
- “PRXMATCH Function” on page 1267
- “PRXPARSE Function” on page 1274
- “PRXPOSN Function” on page 1276

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXPOSN Routine” on page 289
- “CALL PRXSUBSTR Routine” on page 292

PRXPARSE Function

Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.

Category: Character String Matching

Restrictions: Use with other Perl regular expressions.

This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8).

Notes: The PRXPARSE function returns a numeric pattern identifier in this pattern: *regular-expression-id*=PRXPARSE(*perl-regular-expression*)

SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXPARSE(*perl-regular-expression*)

Required Argument

perl-regular-expression
specifies a character value that is a Perl regular expression.

You can see lists of PRX metacharacters in "Tables of Perl Regular Expression (PRX) Metacharacters" on page 1597. For a complete list of metacharacters, see the Perl documentation.

Details

The Basics

The PRXPARSE function returns a pattern identifier number that is used by other Perl functions and CALL routines 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" on page 1597.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 53.

Quantifiers are used when a portion of a pattern needs to match a certain number (or numbers) of times. If there is not a quantifier the number of times to match is one.The maximun value for a quantifier is 2147483646 (which is 2^31) -1).

Here are the supported modifiers:

Table 3.62 *Supported Modifiers*

Modifier	Description
I	Use case-insensitive pattern matching.
M	Treat string as multiple lines.
S	Treat string as a single line.
X	Use extended regular expressions.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant or if it uses the /o option, the Perl regular expression is compiled only once. Successive calls to PRXPARSE do 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.

Note: If you have a Perl regular expression that is a constant, or if the regular expression uses the /o option, then calling PRXFREE to free the memory allocation results in the need to recompile the regular expression the next time it is called by PRXPARSE.The compile-once behavior occurs when you use PRXPARSE in a DATA step. For all other uses, the *perl-regular-expression* is recompiled for each call to PRXPARSE.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Example

The following example uses metacharacters and regular characters to construct a Perl regular expression. The example parses addresses and writes formatted results to the SAS log.

```
data _null_;
   if _N_=1 then
   do;
      retain patternID;
```

```
            /* The i modifier specifies a case insensitive search. */
         pattern="/ave|avenue|dr|drive|rd|road/i";
         patternID=prxparse(pattern);
      end;
      input street $80.;
      call prxsubstr(patternID, street, position, length);
      if position ^= 0 then
      do;
         match=substr(street, position, length);
         put match:$QUOTE. "found in " street:$QUOTE.;
      end;
      datalines;
153 First Street
6789 64th Ave
4 Moritz Road
7493 Wilkes Place
;
```

SAS writes the following output to the log:

```
"Ave" found in "6789 64th Ave"
"Road" found in "4 Moritz Road"
```

See Also

Functions:

- “PRXCHANGE Function” on page 1261
- “PRXMATCH Function” on page 1267
- “PRXPAREN Function” on page 1272
- “PRXPOSN Function” on page 1276

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXPOSN Routine” on page 289
- “CALL PRXSUBSTR Routine” on page 292

PRXPOSN Function

Returns a character string that contains the value for a capture buffer.

Category: Character String Matching

Restrictions: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8).

Use with the PRXPARSE function.

Note: SAS has adopted the International Components for Unicode (ICU). ICU enables SAS to apply regular expression matching to Unicode string data. For more information, see Regular Expressions.

Syntax

PRXPOSN(*regular-expression-id*, *capture-buffer*, *source*)

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned by the PRXPARSE function.

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.

source
specifies the text from which to extract capture buffers.

Note: The *source* string should be the same string that was used in the previous match.

Details

The PRXPOSN function uses the results of PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT to return a capture buffer. A match must be found by one of these functions for PRXPOSN to return meaningful information.

A capture buffer is part of a match, enclosed in parentheses, that is specified in a regular expression. This function simplifies using capture buffers by returning the text for the capture buffer directly, and by not requiring a call to SUBSTR as in the case of CALL PRXPOSN.

For more information about pattern matching, see “Pattern Matching Using Perl Regular Expressions (PRX) ” on page 53.

Comparisons

The PRXPOSN function is similar to the CALL PRXPOSN routine, except that it returns the capture buffer itself rather than the position and length of the capture buffer.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 100.

Examples

Example 1: Extracting First and Last Names

The following example uses PRXPOSN to extract first and last names from a data set.

```
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;
data FirstLastNames;
   length first last $ 16;
   keep first last;
   retain re;
   if _N_=1 then
      re=prxparse('/(\w+), (\w+)/');
   set ReversedNames;
   if prxmatch(re, name) then
      do;
         last=prxposn(re, 1, name);
         first=prxposn(re, 2, name);
      end;
run;
proc print data=FirstLastNames;
run;
```

Output 3.60 *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 example creates a data set that contains a list of names. Observations that have only a first name or only a last name are invalid. PRXPOSN extracts the valid names from the data set, and writes the names to the data set NEW.

```
data old;
   input name $60.;
   datalines;
Judith S Reaveley
Ralph F. Morgan
Jess Ennis
Carol Echols
Kelly Hansen Huff
Judith
Nick
Jones
;
data new;
   length first middle last $ 40;
   keep first middle last;
   re=prxparse('/(\S+)\s+([^\s]+\s+)?(\S+)/o');
   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;
run;
proc print data=new;
run;
```

Output 3.61 *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 1261
- “PRXMATCH Function” on page 1267
- “PRXPAREN Function” on page 1272
- “PRXPARSE Function” on page 1274

CALL Routines:

- “CALL PRXCHANGE Routine” on page 280
- “CALL PRXDEBUG Routine” on page 283
- “CALL PRXFREE Routine” on page 285
- “CALL PRXNEXT Routine” on page 287
- “CALL PRXPOSN Routine” on page 289
- “CALL PRXSUBSTR Routine” on page 292

PTRLONGADD Function

Returns the pointer address as a character variable on a 64-bit platform.

Categories: Binary Results
Special

Syntax

PTRLONGADD(*pointer*<, *amount*>)

Required Arguments

pointer
is a character constant, variable, or expression that specifies the pointer address.

amount
is a numeric constant, variable, or expression that specifies the amount to add to the address.

Tip *amount* can be a negative number.

Details

The PTRLONGADD function performs pointer arithmetic and returns a pointer address as a character string.

Example

The following example returns the pointer address for the variable Z.

```
data _null_;
   x='ABCDE';
   y=ptrlongadd(addrlong(x), 2);
   z=peekclong(y, 1);
   put z=;
run;
```

SAS writes the following output to the log:

```
z=C
```

PUT Function

Returns a value using a specified format.

Category: Special

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

PUT(*source*, *format*.)

Required Arguments

source
identifies the constant, variable, or expression whose value you want to reformat. The *source* argument can be character or numeric.

***format*.**
contains the SAS format that you want applied to the value that is specified in the source. This argument must be the name of a format with a period and optional width and decimal specifications, not a character constant, variable, or expression.

By default, if the source is numeric, the resulting string is right aligned, and if the source is character, the result is left aligned. To override the default alignment, you can add an alignment specification to a format. Here are valid values:

-L left aligns the value.

-C centers the value.

-R right aligns the value.

Restriction The *format*. must be of the same type as the source, either character or numeric. That is, if the source is character, the format name must begin with a dollar sign, but if the source is numeric, the format name must not begin with a dollar sign.

Note Alignment specifications are not supported for an in-database DATA step.

Details

If the PUT function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the width of the format.

Use the PUT function to convert a numeric value to a character value. The PUT function has no effect on which formats are used in PUT statements or which formats are assigned to variables in data sets. You cannot use the PUT function to directly change the type of variable in a data set from numeric to character. However, you can create a new character variable as the result of the PUT function. Then, if needed, use the DROP statement to drop the original numeric variable, followed by the RENAME statement to rename the new variable back to the original variable name.

Comparisons

The PUT statement and the PUT function are similar. The PUT function returns a value using a specified format. You must use an assignment statement to store the value in a variable. The PUT statement writes a value to an external destination (either the SAS log or a destination, that you specify).

Examples

Example 1: Converting Numeric Values to Character Value

In this example, the first statement converts the values of *cc*, a numeric variable, into the four-character hexadecimal format, and the second statement writes the same value that the PUT function returns.

```
data one;
   cc='abc';
   cchex=put(cc,hex4.);
   put cc= hex4.;

/* If you need to keep the original variable name of cc, but as a
character variable,
    then use the DROP and RENAME statements following the PUT
function.  */

   cchex=put(cc,hex4.);
   drop cc;
   rename cchex=cc;
   put cc=;
run;
```

Note: The new cchex variable is created as a character variable from the numeric value of the cc variable. The DROP statement prevents the numeric cc variable from being written to the data set, and the RENAME statement renames the new character cchex variable back to the name of cc.

These statements produce this result:

```
cc=6162
cc=abc
```

Example 2: Using PUT and INPUT Functions

In this example, the PUT function returns a numeric value as a character string. The value 122591 is assigned to the CHARDATE variable. The INPUT function returns the value of the character string as a SAS date value using a SAS date informat. The value 11681 is stored in the SASDATE variable.

```
data one;
   numdate=122591;
```

```
   chardate=put(numdate, z6.);
   sasdate=input(chardate, mmddyy6.);
   put sasdate=;
run;
```

These statements produce this result:

```
sasdate=11681
```

Example 3: Specifying Alignment

This example demonstrates alignment.

Note: Alignment specifications are not supported for an in-database DATA step.

```
data test;
length y $6;
   y='   abc';
   x=put(y,$6. -L);
   put 'y=' y $6.;
   put 'x=' x $6.;
run;
```

These statements produce this result:

```
y=   abc
x=abc
```

See Also

Functions:

- “INPUT Function” on page 956
- “INPUTC Function” on page 959
- “INPUTN Function” on page 962
- “PUTC Function” on page 1284
- “PUTN Function” on page 1288

Statements:

- “PUT Statement” in *SAS DATA Step Statements for SAS Viya Workbench*

PUTC Function

Enables you to specify a character format at run time.

Category: Special

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

PUTC(*value*, *format-specification* <, *w*>)

Required Arguments

value
specifies a character value to be formatted.

format-specification
is a character format that you want to apply to *value*.

Here are valid format forms:

- *format-name*
- *format-name.*
- *format-namew*.

Except for *format-name*, you can use –L, –R, and –C in *format-specification* to left-align, right-align, and center your output. For example, you can use `'upcase.-c'` as the value for the second argument, *format-specification*.

Optional Argument

w
is a numeric constant, variable, or expression that specifies a width to apply to the format.

Interaction If you specify a width here, it overrides any width specification in the format.

Details

If the PUTC function returns a value to a variable that has not yet been assigned a length, then the variable length is determined by the length of the first argument.

Comparisons

The PUTN function enables you to specify a numeric format at run time.

The PUT function is faster than PUTC because PUT enables you to specify a format at compile time rather than at run time.

Examples

Example 1: Working with the PUTC Function

The PROC FORMAT step creates a format, TYPEFMT., that formats the variable values 1, 2, and 3 with the name of one of the three other formats that this step creates. These three formats output responses of "positive," "negative," and "neutral" as different words, depending on the type of question. After PROC FORMAT creates the formats, the DATA step creates a SAS data set from raw data that consists of a number identifying the type of question and a response. After reading a record, the DATA step uses the value of TYPE to create a variable, RESPFMT. This variable contains the value of the appropriate format for the current type of question. The DATA step also creates another variable, WORD, whose value is the appropriate word for a response. The PUTC function assigns the value of WORD based on the type of question and the appropriate format.

```
proc format;
   value typefmt 1='$groupx'
                 2='$groupy'
                 3='$groupz';
   value $groupx 'positive'='agree'
                 'negative'='disagree'
                 'neutral'='notsure ';
   value $groupy 'positive'='accept'
                 'negative'='reject'
                 'neutral'='possible';
   value $groupz 'positive'='pass    '
                 'negative'='fail'
                 'neutral'='retest';
run;
data answers;
   length word $ 8;
   input type response $;
   respfmt=put(type, typefmt.);
   word=putc(response, respfmt);
   datalines;
1 positive
1 negative
1 neutral
2 positive
2 negative
2 neutral
3 positive
3 negative
3 neutral
;

proc print data=answers;
   title 'Using the Third Argument as a Number or Cycle';
run;
```

Output 3.62 *Output from the PUTC Function*

Using the Third Argument as a Number or Cycle

Obs	word	type	response	respfmt
1	agree	1	positive	$groupx
2	disagree	1	negative	$groupx
3	notsure	1	neutral	$groupx
4	accept	2	positive	$groupy
5	reject	2	negative	$groupy
6	possible	2	neutral	$groupy
7	pass	3	positive	$groupz
8	fail	3	negative	$groupz
9	retest	3	neutral	$groupz

The value of the variable WORD is `agree` for the first observation. The value of the variable WORD is `retest` for the last observation.

Example 2: Aligning Character Values

This example shows how to use a format and an alignment character to align the value of A.

```
data _null_;
   length a $20;
   a='experiment';
   y=putc(a,'upcase.-r',20);
   put '*' y $char20. '*';
   put '*' a $upcase20. '*';
run;
```

```
*          EXPERIMENT*
*EXPERIMENT          *
```

See Also

Functions:

- “INPUT Function” on page 956
- “INPUTC Function” on page 959
- “INPUTN Function” on page 962
- “PUT Function” on page 1281
- “PUTN Function” on page 1288

PUTN Function

Enables you to specify a numeric format at run time.

Category: Special

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

PUTN(*value*, *format-specification* <, *w* <, *d*>>)

Required Arguments

value
specifies a numeric value to be formatted.

format-specification
is the numeric format that you want to apply to *value*.

Here are valid format forms:

- *format-name*
- *format-name.*
- *format-namew*.
- *format-namew.d*

Except for *format-name*, you can use -L, -R, and -C in *format-specification* to left-align, right-align, and center your output. For example, you can use `'weekdate.-c'` as the value for the second argument, *format-specification*. Note that alignment specifications are not supported for an in-database DATA step.

Optional Arguments

w
is a numeric constant, variable, or expression that specifies a width to apply to the format.

Interaction If you specify a width here, it overrides any width specification in the format.

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 format.

Details

If the PUTN 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 bytes.

Comparisons

The PUTC function enables you to specify a character format at run time.

The PUT function is faster than PUTN because PUT enables you to specify a format at compile time rather than at run time.

Examples

Example 1: Working with Dates and Formats in the PUTN Function

The PROC FORMAT step creates a format, WRITFMT., that formats the variable values 1 and 2 with the name of a SAS date format. The DATA step creates a SAS data set from raw data that consists of a number and a key. After reading a record, the DATA step uses the value of KEY to create a variable, DATEFMT, that contains the value of the appropriate date format. The DATA step also creates a new variable, DATE, whose value is the formatted value of the date. PUTN assigns the value of DATE based on the value of NUMBER and the appropriate format.

```
proc format;
   value writfmt 1='date9.'
                 2='mmddyy10.';
run;
data dates;
   input number key;
   datefmt=put(key,writfmt.);
   date=putn(number,datefmt);
   datalines;
15756 1
14552 2
;

proc print data=dates;
   title 'Working with Dates and Formats';
run;
```

Output 3.63 *Output from Using the PUTN Function*

Working with Dates and Formats

Obs	number	key	datefmt	date
1	15756	1	date9.	20FEB2003
2	14552	2	mmddyy10.	11/04/1999

Example 2: Aligning Output from the PUTN Function

This example shows how to use a format and an alignment character to left-align a value.

```
data _null_;
   length y $30;
   y=putn(today(),'weekdate.-l',30);
   put '*' y $char30. '*';
run;
```

Example Code 3.4 *Alignment Results*

```
*Monday, November 19, 2012     *
```

See Also

Functions:

- “INPUT Function” on page 956
- “INPUTC Function” on page 959
- “INPUTN Function” on page 962
- “PUT Function” on page 1281
- “PUTC Function” on page 1284

PVP Function

Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.

Category: Financial

Syntax

PVP(*A*, *c*, *n*, *K*, k_0, *y*)

Required Arguments

A

specifies the par value.

Range: $A > 0$

c

specifies the nominal per-year coupon rate, expressed as a fraction.

Range: $0 \le c < 1$

n

specifies the number of coupons per year.

Range: $n > 0$ and is an integer

K

specifies the number of remaining coupons.

Range: $K > 0$ and is an integer

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of years.

Range: $0 < k_0 \le \frac{1}{n}$

y

specifies the nominal per-year yield-to-maturity, expressed as a fraction.

Range: $y > 0$

Details

The PVP function is based on the 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

```
data _null_;
   p=pvp(1000, .01, 4, 14, .33/2, .10);
   put p=;
run;
```

These statements produce this result:

```
p=743.168
```

QTR Function

Returns the quarter of the year from a SAS date value.

Category: Date and Time

Syntax

QTR(*date*)

Required Argument

date
specifies a numeric constant, variable, or expression that represents a SAS date value.

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.

Example

```
data new;
   do i= 0 to 3;
   x=intnx('qtr','20jan21'd,i);
   y=qtr(x);
   put x= date. y=;
  end;
run;
```

These statements produce these results:

```
x=01JAN21 y=1
x=01APR21 y=2
x=01JUL21 y=3
x=01OCT21 y=4
```

See Also

Functions:

- "YYQ Function" on page 1579

QUANTILE Function

Returns the quantile from a distribution when you specify the left probability (CDF).

Category: Quantile

See: "CDF Function" on page 392

Syntax

QUANTILE(*distribution*, *probability*, *parameter-1, ..., parameter-k*)

Required Arguments

distribution
is a character constant, variable, or expression that identifies the distribution.

Valid distributions are as follows:

Distribution	Argument
Bernoulli	`BERNOULLI`
Beta	`BETA`
Binomial	`BINOMIAL`
Cauchy	`CAUCHY`
Chi-Square	`CHISQUARE`
Conway-Maxwell-Poisson	`CONMAXPOI`

Distribution	Argument
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.

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 "Details" on page 394 in the CDF function.

The QUANTILE function for the Conway-Maxwell-Poisson distribution returns the largest integer 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 1198.

For more information about the distributions that are listed in the table, see "PDF Function" on page 1193.

Example

```
data
new;

   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', .2,
2.3, .4);
```

```
g=quantile('EXPO', .6);

   h=quantile('F', .8, 2,
3);

   i=quantile('GAMMA', .4,
3);

   j=quantile('GENPOISSON', .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 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=;
run;
```

These statements produce these results:

```
a=0
b=0.2009088789
c=5
d=1.9626105055
e=11.529833841
f=5
g=0.9162907319
h=2.8860266073
i=2.285076904
j=9
k=2
l=0.9162907319
m=0.8472978604
n=1
o=1
p=1.9599639845
q=1.1
r=1.0101010101
s=2
t=0.9195437802
u=1.2611198197
v=0.25
w=0.9526209927
x=0.9572307621
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084

- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "SDF Function" on page 1376
- "SQUANTILE Function" on page 1404

QUOTE Function

Adds double quotation marks to a character value.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

QUOTE(*argument-1*, *argument-2*)

Required Arguments

argument-1
specifies a character constant, variable, or expression.

argument-2
specifies a quoting character, which is a single or double quotation mark. Other characters are ignored and the double quotation mark is used. The double quotation mark is the default.

Details

Length of Returned Variable

In a DATA step, if the QUOTE 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.

The Basics

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 log.

Example

```
data _null_;
   x='A"B';
   a=quote(x);
   put a=;
run;
```

These statements produce this result:

```
a="A""B"
```

```
data _null_;
   x='A''B';
   b=quote(x);
   put b=;
run;
```

These statements produce this result:

```
b="A'B"
```

```
data _null_;
   x='Paul''s';
   c=quote(x);
   put c=;
run;
```

These statements produce this result:

```
c="Paul's"
```

```
data _null_;
   x='Paul''s Catering Service          ';
   d=quote(trim(x));
   put d=;
run;
```

These statements produce this result:

```
d="Paul's Catering Service"
```

```
data _null_;
```

```
      x='Catering Service Center          ';
      e=quote(trim(x));
      put e=;
run;
```

These statements produce this result:

```
e="Catering Service Center"
```

```
data _null_;
   x='Catering Service Center          ';
   f=quote(x);
   put f=;
run;
```

These statements produce this result:

```
f="Catering Service Center          "
```

```
data _null_;
   x='Catering Service Center          ';
   g=quote(x);
   put g=;
run;
```

These statements produce this result:

```
g="Catering Service Center          "
```

```
data _null_;
   x='Catering Service Center    ';
   h=quote(x);
   put h=;
run;
```

These statements produce this result:

```
h="Catering Service Center    "
```

```
data _null_;
   x=quote('abc');
   put x=;
run;
```

These statements produce this result:

```
x="abc"
```

```
data _null_;
   x=quote('abc',"'");
   put x=;
run;
```

Note: The second argument contains a single quotation mark. In order to be passed to the function in the DATA step, the argument is specified in the DATA step as double quotation mark, single quotation mark, or double quotation mark. The argument could have also been specified as four single quotation marks (that is, a quoted string that uses single quotation marks). The quoted value is an escaped single quotation mark represented as two single quotation marks.

These statements produce this result:

```
x='abc'
```

RANBIN Function

Returns a random variate from a binomial distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Binomial') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANBIN routine instead of the RANBIN function.

Syntax

RANBIN(*seed*, *n*, *p*)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Concepts" on page 20 for more information about seed values

n

is a numeric constant, variable, or expression with an integer value that specifies the number of independent Bernoulli trials parameter.

Range $n > 0$

p
is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 < p < 1$

Details

The RANBIN function returns a variate that is generated from a binomial distribution with mean *np* and variance *np*(1−*p*). If $n \leq 50$, $np \leq 5$, or $n(1-p) \leq 5$, an inverse transform method applied to a RANUNI uniform variate is used. If $n > 50$, $np > 5$, and $n(1-p) > 5$, the normal approximation to the binomial distribution is used. In that case, the Box-Muller transformation of RANUNI uniform variates is used.

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "RNGs, Seeds, and Random Numbers from Distributions" on page 23.

Comparisons

The CALL RANBIN routine, an alternative to the RANBIN function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 1304

CALL Routines:

- "CALL RANBIN Routine" on page 295
- "CALL STREAMINIT Routine" on page 352

RANCAU Function

Returns a random variate from a Cauchy distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Cauchy') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANCAU routine instead of the RANCAU function.

Syntax

RANCAU(*seed*)

Required Argument

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range *seed* < 2^{31} −1

See "Concepts" on page 20 for more information about seed values

Details

The RANCAU function returns a variate that is generated from a Cauchy distribution with location parameter 0 and scale parameter 1. An acceptance-rejection procedure applied to RANUNI uniform variates is used. If *u* and *v* are independent uniform (−1/2, 1/2) variables and $u^2 + v^2 \leq 1/4$, then *u*/*v* is a Cauchy variate. A Cauchy variate X with location parameter ALPHA and scale parameter BETA can be generated:

```
x=alpha+beta*rancau(seed);
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "RNGs, Seeds, and Random Numbers from Distributions" on page 23.

Comparisons

The CALL RANCAU routine, an alternative to the RANCAU function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 1304

CALL Routines:

- "CALL RANCAU Routine" on page 298
- "CALL STREAMINIT Routine" on page 352

RAND Function

Generates random numbers from a distribution that you specify.

Category: Random Number

Syntax

RAND(*distribution*, *parameter-1*, ..., *parameter-k*)

Required 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

Distribution	Function Call	Parameter Values
	`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)
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

Distribution	Function Call	Parameter Values
	`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
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

Distribution	Function Call	Parameter Values
	`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)

Note: Except for F and T, 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 appropriate for the specific distribution.

See "Details" on page 1307

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 (Matsumoto and Nishimura 1998) by default. Other random number generators (RNGs) can be specified by the STREAMINIT routine.

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, use the CALL STREAMINIT routine to specify a seed value for random number generation. Use the CALL STREAMINIT routine before any invocation of the RAND function. If you omit the call to the CALL STREAMINIT routine (or if you specify a nonpositive seed value in the CALL STREAMINIT routine), RAND obtains an initial seed from either the Intel RdRand instruction on machines that support the instruction or from the current date-time value on machines that do not support RdRand. For more

information, see CALL STREAMINIT."Example 1: Generate Random Integers" on page 355

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

Pseudo-random numbers are generated using numerical algorithms that transform uniform random variates. For some distributions, very small or very large parameter values can result in pseudo-random 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 pseudo-random 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 this 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 have the RAND function generate random numbers despite the possibility of a poor distribution, append a question mark to the end of the name of the distribution 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 this 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 constant, variable, or expression that specifies the 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 this 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 constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

b

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $b > 0$

Binomial Distribution

x=**RAND**('BINOMIAL', *p*, *n*)

Arguments

x

is an integer observation from the Binomial distribution with this 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 constant, variable, or expression that specifies the probability of success.

Range $0 \le p \le 1$

n
is an integer parameter that counts the number of independent Bernoulli trials.

Range $n = 1, 2, \ldots$

Cauchy Distribution

x=**RAND**('CAUCHY')

Arguments

x
is an observation from the Cauchy distribution with this 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 this 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 constant, variable, or expression that specifies the degrees of freedom.

Range $df > 0$

Erlang Distribution

x=**RAND**('ERLANG', *a*)

Arguments

x

is an observation from the Erlang distribution with this probability density function:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

Range $x > 0$

a

is a numeric constant, variable, or expression that specifies a shape parameter.

Range $a = 1, 2, \ldots$

Exponential Distribution

x=**RAND**('EXPONENTIAL', <*σ*>)

Arguments

x

is an observation from the Exponential distribution with this probability density function:

$$f(x) = \frac{1}{\sigma} \exp\left(-x/\sigma\right)$$

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 with this 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). To ensure

the accuracy of the random variates and to prevent numerical overflow, the ξ parameter is restricted to the range [-0.5, 5].

μ
is a numeric location parameter.

Default 0

σ
is a numeric scale parameter.

Default 1

Range σ > 0

ξ
is a numeric shape parameter.

Default 0

F Distribution

x=**RAND**('F', *n*, *d*)

Arguments

x
is an observation from the F distribution with this 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 constant, variable, or expression that specifies the numerator degrees of freedom.

Range *n* > 0

d
is a numeric constant, variable, or expression that specifies the denominator degrees of freedom.

Range *d* > 0

Gamma Distribution

x=**RAND**('GAMMA', *a*, <*λ*>)

Arguments

x

is an observation from the Gamma distribution with this probability density function:

$$f(x) = \frac{x^{a-1}}{\lambda^a \Gamma(a)} \exp(-x/\lambda)$$

Range $x > 0$

a

is a numeric shape parameter.

Range $a > 0$

λ

is a scale parameter.

Default 1

Range λ>0

Geometric Distribution

x=**RAND**('GEOMETRIC', *p*)

Arguments

x

is an integer count that denotes the number of trials that are needed to obtain one success. *X* is an integer observation from the Geometric distribution with this 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 $x = 1, 2, \ldots$

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 < p \leq 1$

Gompertz Distribution

x=**RAND**('GOMPERTZ',<λ,σ>)

Arguments

x

is an observation from the Gompertz distribution with this cumulative probability distribution:

$$F(x) = 1 - \exp(-\lambda * [\exp(x/\sigma) - 1])$$

The random values of *x* are in the interval (0, ∞).

λ

is a numeric shape parameter.

Default 1

σ

is a numeric scale parameter.

Default 1

Range σ > 0

Gumbel Distribution

x=RAND('GUMBEL',<μ, σ>)

Arguments

x

is an observation from the Gumbel distribution with this 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 location parameter.

Default 0

σ

is a numeric scale parameter.

Default 1

Range σ > 0

Hypergeometric Distribution

x=**RAND**('HYPER', *N*, *R*, *n*)

Arguments

x

is an integer observation from the Hypergeometric distribution with this probability density function:

$$f(x) = \frac{\binom{R}{x}\binom{N-R}{n-x}}{\binom{N}{n}}$$

Range $x = \max(0, (n - (N - R))), \ldots, \min(n, R)$

N

is a numeric constant, variable, or expression that specifies an integer population size parameter.

Range $N = 1, 2, \ldots$

R

is a numeric constant, variable, or expression that specifies an integer number of items in the category of interest.

Range $R = 0, 1, \ldots, N$

n

is a numeric constant, variable, or expression that specifies an integer sample size parameter.

Range $n = 1, 2, \ldots, N$

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 with this 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 with this 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 λ > 0

Lognormal Distribution

x=**RAND**('LOGNORMAL',<*θ*,*λ*>)

Arguments

x

is an observation from the Lognormal distribution with this 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 log-scale parameter.

Default 0

λ

is a shape parameter.

Default 1

Range λ > 0

Negative Binomial Distribution

x=**RAND**('NEGBINOMIAL', *p*, *k*)

Arguments

x

is an integer observation from the Negative Binomial distribution with this probability density function:

$$f(x) = \begin{cases} \binom{x+k-1}{k-1}(1-p)^x p^k & 0 < p < 1, x = 0, 1, \ldots \\ 1 & p = 1, x = 0 \end{cases}$$

Range *x* = 0, 1, ...

k

is an integer parameter that is the number of successes. However, noninteger *k* values are also allowed.

Range *k* = 1, 2, ...

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range 0 < *p* ≤ 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 this 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 a numeric constant, variable, or expression that specifies a mean parameter.

Default 0

λ
is a numeric constant, variable, or expression that specifies a standard deviation parameter.

Default 1

Range λ > 0

Pareto Distribution

x=**RAND**('PARETO', *a*,<*k*>)

Arguments

x
is an observation from the Pareto distribution with this 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 integer observation from the Poisson distribution with this probability density function:

$$f(x) = \frac{m^x e^{-m}}{x!}$$

Range *x* = 0, 1, ...

m
is a numeric constant, variable, or expression that specifies a mean parameter.

Range $m > 0$

Shifted Gompertz Distribution

x=**RAND**('SHGOMPERTZ', <*η*,*τ*>)

Arguments

x

is an observation from the shifted Gompertz distribution with this cumulative probability distribution:

$$F(x) = (1 - \exp(-\tau x))\exp(-\eta \exp(-\tau x))$$

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 this 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 constant, variable, or expression that specifies the degrees of freedom.

Range $df > 0$

Tabled Distribution

x=**RAND**('TABLE', *p1*, *p2*, ...)

Arguments

x
is an integer observation from one of these distributions:

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 this 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 a numeric constant, variable, or expression that specifies 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, here is the probability density function on (a,b):

$$f(x) = \frac{1}{|b-a|}$$

If you do not specify any parameters, the interval (0,1) is used. If you specify one parameter, 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 uniform random number generator that the RAND function uses is the Mersenne-Twister (Matsumoto and Nishimura 1998). This generator has a period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy. This algorithm underlies the generators for the other available distributions in the RAND function.

Wald (Inverse Gaussian) Distribution

x=**RAND**('WALD' <,*λ*<,*θ*>>)

x=**RAND**('IGAUSS' <,*λ*<,*θ*>>)

Arguments

x

is an observation from the Wald distribution with this 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.

Default 1

Range $\lambda > 0$

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 this 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 constant, variable, or expression that specifies a shape parameter.

Range $a > 0$

b

is a numeric constant, variable, or expression that specifies a scale parameter.

Range $b > 0$

Example

This example demonstrates the distributions and their output in the RAND function.

```
data Rand;
length d $7;
label d="Distribution" x="Result";
call streaminit(12345);
d='BERN';   x=rand('BERN', .75);       output;
d='BETA';   x=rand('BETA', 3, 1);      output;
```

```
d='BINOM';  x=rand('BINOM', 0.75, 10); output;
d='CAUCHY'; x=rand('CAUCHY');          output;
d='CHISQ';  x=rand('CHISQ', 22);       output;
d='ERLANG'; x=rand('ERLANG', 7);       output;
d='EXTVAL'; x=rand('EXTRVAL', 0, 1);   output;
d='EXPO';   x=rand('EXPO');            output;
d='F';      x=rand('F', 12, 322);      output;
d='GAMMA';  x=rand('GAMMA', 7.25);     output;
d='GEOM';   x=rand('GEOM', 0.02);      output;
d='GOMP';   x=rand('GOMP', 1, 0.5);    output;
d='GUMBEL'; x=rand('GUMBEL', 0, 2);    output;
d='HYPER';  x=rand('HYPER', 10, 3, 5); output;
d='INTEGER';x=rand('INTEGER', 1, 10);  output;
d='LAPLACE';x=rand('LAPLACE');         output;
d='LOGIST'; x=rand('LOGIST');          output;
d='LOGN';   x=rand('LOGN');            output;
d='NEGB';   x=rand('NEGB', 0.8, 5);    output;
d='NORMAL'; x=rand('NORMAL');          output;
d='PARETO'; x=rand('PARETO', 3, 1);    output;
d='POISSON';x=rand('POISSON', 6.1);    output;
d='SHGOMP'; x=rand('SHGOMP', 0.5, 1.2);output;
d='T';      x=rand('T', 4);            output;
d='TABLE';  x=rand('TABLE', .2, .5);   output;
d='TRIANG'; x=rand('TRIANG', 0.7);     output;
d='UNIFORM';x=rand('UNIFORM');         output;
d='WALD';   x=rand('WALD', 0.25, 2.1); output;
d='WEIB';   x=rand('WEIB', 1, 2);      output;
run;


proc print data=Rand label;
format x best8.;
run;
```

Output 3.64 *Output for the RAND Functions Example*

Obs	Distribution	Result
1	BERN	1
2	BETA	0.997871
3	BINOM	7
4	CAUCHY	1.100958
5	CHISQ	17.91355
6	ERLANG	6.125075
7	EXTVAL	2.020154
8	EXPO	0.65704
9	F	1.67051
10	GAMMA	14.88252
11	GEOM	15
12	GOMP	0.481501
13	GUMBEL	2.488377
14	HYPER	0
15	INTEGER	9
16	LAPLACE	1.351669
17	LOGIST	1.467011
18	LOGN	2.044847
19	NEGB	2
20	NORMAL	-2.66609
21	PARETO	1.118531
22	POISSON	5
23	SHGOMP	0.865726
24	T	-1.07634
25	TABLE	1
26	TRIANG	0.621272
27	UNIFORM	0.782733
28	WALD	0.597682
29	WEIB	8.450918

See Also

CALL Routines:

- "CALL STREAMINIT Routine" on page 352

References

Matsumoto, M., and T. Nishimura. 2002. "Mersenne Twister with Improved Initialization" 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. 2006. *Simulation*. San Diego, USA: Academic Press.

RANEXP Function

Returns a random variate from an exponential distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Exponential') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANEXP routine instead of the RANEXP function.

Syntax

RANEXP(*seed*)

Required Argument

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range *seed* < $2^{31}-1$

See "Concepts" on page 20 for more information about seed values

Details

The RANEXP function returns a variate that is generated from an exponential distribution with parameter 1. An inverse transform method applied to a RANUNI uniform variate is used.

An exponential variate X with parameter LAMBDA can be generated:

```
x=ranexp(seed)/lambda;
```

An extreme value variate X with location parameter ALPHA and scale parameter BETA can be generated:

```
x=alpha-beta*log(ranexp(seed));
```

A geometric variate X with parameter P can be generated as follows:

```
x=floor(-ranexp(seed)/log(1-p));
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "RNGs, Seeds, and Random Numbers from Distributions" on page 23.

Comparisons

The CALL RANEXP routine, an alternative to the RANEXP function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 1304

CALL Routines:

- “CALL RANEXP Routine” on page 304
- “CALL STREAMINIT Routine” on page 352

RANGAM Function

Returns a random variate from a gamma distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Gamma') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANGAM routine instead of the RANGAM function.

Syntax

RANGAM(*seed*, *a*)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See “Concepts” on page 20 for more information about seed values

a

is a numeric constant, variable, or expression that specifies the shape parameter.

Range $a > 0$

Details

The RANGAM function returns a variate that is generated from a gamma distribution with parameter *a*. For $a > 1$, an acceptance-rejection method due to Cheng (1977) is used. (See “References” on page 1609). For $a \leq 1$, an acceptance-rejection method due to Fishman is used (1978, Algorithm G2) (See “References” on page 1609).

A gamma variate X with shape parameter ALPHA and scale BETA can be generated:

```
x=beta*rangam(seed, alpha);
```

If 2*ALPHA is an integer, a chi-square variate X with 2*ALPHA degrees of freedom can be generated:

```
x=2*rangam(seed, alpha);
```

If N is a positive integer, an Erlang variate X can be generated:

```
x=beta*rangam(seed, N);
```

It has the distribution of the sum of N independent exponential variates whose means are BETA.

And finally, a beta variate X with parameters ALPHA and BETA can be generated:

```
y1=rangam(seed, alpha);
y2=rangam(seed, beta);
x=y1/(y1+y2);
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "RNGs, Seeds, and Random Numbers from Distributions" on page 23.

Comparisons

The CALL RANGAM routine, an alternative to the RANGAM function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 1304

CALL Routines:

- "CALL RANGAM Routine" on page 306
- "CALL STREAMINIT Routine" on page 352

RANGE Function

Returns the range of the nonmissing values.

Category: Descriptive Statistics

Syntax

RANGE(*argument-1* <, ...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one nonmissing argument is required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Details

The RANGE function returns the difference between the largest and the smallest of the nonmissing arguments.

Example

```
data
one;


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=;

run;
```

These statements produce these results:

```
x0=. x1=8 x2=4 x3=5 x4=4
```

RANK Function

Returns the position of a character in the ASCII collating sequence.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Operating environment: This function is supported under a Linux OS.

Syntax

RANK(*x*)

Required Argument

x

specifies a character constant, variable, or expression that contains a character in the ASCII collating sequence. If the length of *x* is greater than 1, you receive the rank of the first character in the string.

Details

The RANK function returns an integer 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

```
data
one;
```

```
n=rank('A');

   put
n=;

run;
```

These statements produce this result:

```
n=65
```

See Also

Functions:

- "BYTE Function" on page 214
- "COLLATE Function" on page 442

RANNOR Function

Returns a random variate from a normal distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Normal') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANNOR routine instead of the RANNOR function.

Syntax

RANNOR(*seed*)

Required Argument

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range *seed* < $2^{31}-1$

See "Concepts" on page 20 for more information about seed values

Details

The RANNOR function returns a variate that is generated from a normal distribution with mean 0 and variance 1. The Box-Muller transformation of RANUNI uniform variates is used.

A normal variate X with mean MU and variance S2 can be generated with this code:

```
x=MU+sqrt(S2)*rannor(seed);
```

A lognormal variate X with mean *exp*(MU + S2/2) and variance *exp*(2*MU + 2*S2) −*exp*(2*MU + S2) can be generated with this code:

```
x=exp(MU+sqrt(S2)*rannor(seed));
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see “RNGs, Seeds, and Random Numbers from Distributions” on page 23.

Comparisons

The CALL RANNOR routine, an alternative to the RANNOR function, gives greater control of the seed and random number streams.

See Also

Functions:

- “RAND Function” on page 1304

CALL Routines:

- “CALL RANNOR Routine” on page 309
- “CALL STREAMINIT Routine” on page 352

RANPOI Function

Returns a random variate from a Poisson distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Poisson') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANPOI routine instead of the RANPOI function.

Syntax

RANPOI(*seed*, *m*)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range *seed* < $2^{31}-1$

See "Concepts" on page 20 for more information about seed values

m

is a numeric constant, variable, or expression that specifies the mean of the distribution.

Range $m \geq 0$

Details

The RANPOI function returns a variate that is generated from a Poisson distribution with mean *m*. For *m* < 85, an inverse transform method applied to a RANUNI uniform variate is used (Fishman 1976). (See "References" on page 1609 .) For *m* ≥ 85, the normal approximation of a Poisson random variable is used. To expedite execution, internal variables are calculated only on initial calls (that is, with each new *m*).

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "RNGs, Seeds, and Random Numbers from Distributions" on page 23.

Comparisons

The CALL RANPOI routine, an alternative to the RANPOI function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 1304

CALL Routines:

- “CALL RANPOI Routine” on page 316
- “CALL STREAMINIT Routine” on page 352

RANTBL Function

Returns a random variate from a tabled probability distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Table') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANTBL routine instead of the RANTBL function.

Syntax

RANTBL(*seed*, *p1*...p_i..., p_n)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range *seed* < $2^{31}-1$

See “Concepts” on page 20 for more information about seed values

p_i

is a numeric constant, variable, or expression.

Range $0 \le p_i \le 1$ for $0 < i \le n$

Details

The RANTBL function returns a variate that is generated from the probability mass function defined by p_1 through p_n. An inverse transform method applied to a RANUNI uniform variate is used. RANTBL returns

1 *with probability* p_1

2 *with probability* p_2

.

.

.

n *with probability* p_n

$n+1$ *with probability* $1 - \sum_{i=1}^{n} p_i$ *if* $\sum_{i=1}^{n} p_i \leq 1$

If, for some index *j*<*n*, $\sum_{i=1}^{j} p_i \geq 1$, RANTBL returns only the indices 1 through *j* with the probability of occurrence of the index *j* equal to $1 - \sum_{i=1}^{j-1} p_i$.

Let n=3 and P1, P2, and P3 be three probabilities with P1+P2+P3=1, and M1, M2, and M3 be three variables. The variable X in these statements

```
array m{3} m1-m3;
x=m{rantbl(seed, of p1-p3)};
```

will be assigned one of the values of M1, M2, or M3 with probabilities of occurrence P1, P2, and P3, respectively.

See "Using Functions and CALL Routines" on page 7for information about random number functions and CALL routines.

Comparisons

The CALL RANTBL routine, an alternative to the RANTBL function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 1304

CALL Routines:

- "CALL RANTBL Routine" on page 319
- "CALL STREAMINIT Routine" on page 352

RANTRI Function

Returns a random variate from a triangular distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Triangle') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANTRI routine instead of the RANTRI function.

Syntax

RANTRI(*seed*, *h*)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range *seed* < $2^{31}-1$

See "Concepts" on page 20for more information about seed values

h

is a numeric constant, variable, or expression that specifies the mode of the distribution.

Range $0 < h < 1$

Details

The RANTRI function returns a variate that is generated from the triangular distribution on the interval (0,1) with parameter *h*, which is the modal value of the distribution. An inverse transform method applied to a RANUNI uniform variate is used.

A triangular distribution X on the interval (A,B) with mode C, where A ≤ C ≤ B, can be generated:

```
x=(b-a)*rantri(seed, (c-a)/(b-a))+a;
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "RNGs, Seeds, and Random Numbers from Distributions" on page 23.

Comparisons

The CALL RANTRI routine, an alternative to the RANTRI function, gives greater control of the seed and random number streams.

See Also

Functions:

- “RAND Function” on page 1304

CALL Routines:

- “CALL RANTRI Routine” on page 321
- “CALL STREAMINIT Routine” on page 352

RANUNI Function

Returns a random variate from a uniform distribution.

Category: Random Number

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND('Uniform') function.

Tip: If you want to change the seed value during execution, you must use the CALL RANUNI routine instead of the RANUNI function.

Syntax

RANUNI(*seed*)

Required Argument

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range *seed* < $2^{31}-1$

See “Concepts” on page 20 for more information about seed values

Details

The RANUNI function returns a number that is generated from the uniform distribution on the interval (0,1) using a prime modulus multiplicative generator with modulus $2^{31}-$ and multiplier 397204094 (Fishman and Moore 1982). (See "References" on page 1609.)

You can use a multiplier to change the length of the interval and an added constant to move the interval. For example,

```
random_variate=a*ranuni(seed)+b;
```

returns a number that is generated from the uniform distribution on the interval (b, a+b).

Comparisons

The CALL RANUNI routine, an alternative to the RANUNI function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 1304

CALL Routines:

- "CALL RANUNI Routine" on page 324
- "CALL STREAMINIT Routine" on page 352

RENAME Function

Renames a member of a SAS library, an entry in a SAS catalog, an external file, or a directory.

Categories: External Files
SAS File I/O

Syntax

RENAME(*old-name*, *new-name* <, *type* <, *description* <, *password* <, *generation*>>>>)

Required Arguments

old-name
specifies a character constant, variable, or expression that is the current name of a member of a SAS library, an entry in a SAS catalog, an external file, or an external directory.

For a data set, *old-name* can be a one-level or two-level name. For a catalog entry, *old-name* can be a one-level, two-level, or four-level name. For an external file or directory, *old-name* must be the full pathname of the file or the directory. If the value for *old-name* is not specified, then SAS uses the current directory.

new-name
specifies a character constant, variable, or expression that is the new one-level name for the library member, catalog entry, external file, or directory.

Optional Arguments

type
is a character constant, variable, or expression that specifies the type of element to rename. *Type* can be a null argument, or another value.

Here are valid values:

ACCESS	specifies a SAS/ACCESS descriptor that was created using SAS/ACCESS software.
CATALOG	specifies a SAS catalog or catalog entry.
DATA	specifies a SAS data set.
VIEW	specifies a SAS data set view.
FILE	specifies an external file or directory.

Default 'DATA'

description
specifies a character constant, variable, or expression that is the description of a catalog entry. You can specify *description* only when the value of *type* is CATALOG. *Description* can be a null argument.

password
is a character constant, variable, or expression that specifies the password for the data set that is being renamed. *Password* can be a null argument.

generation
is a numeric constant, variable, or expression that specifies the generation number of the data set that is being renamed. *Generation* can be a null argument.

Details

You can use the RENAME function to rename members of a SAS library or entries in a SAS catalog. SAS returns 0 if the operation was successful, and a value other than 0 if the operation was not successful.

To rename an entry in a catalog, specify the four-level name for *old-name* and a one-level name for *new-name*. You must specify CATALOG for *type* when you rename an entry in a catalog.

Operating Environment Information: The RENAME function works correctly for renaming data sets that reside on one volume. It does not rename data sets that span multiple volumes. SAS does not issue an error if you use the RENAME function to rename a data set that resides on multiple volumes. The return code that SAS issues might not be valid if you are renaming a data set that resides on more than one volume.

Examples

Example 1: Renaming Data Sets and Catalog Entries

The following examples rename a SAS data set from Data1 to Data2, and also rename a catalog entry from A.SCL to B.SCL.

```
rc1=rename('mylib.data1', 'data2');
rc2=rename('mylib.mycat.a.scl', 'b', 'catalog');
```

Example 2: Renaming an External File

The following examples rename external files.

```
   /* Rename a file that is located in another directory. */
rc=rename('/local/u/testdir/first',
          '/local/u/second', 'file');
   /* Rename a PC file. */
rc=rename('d:\temp', 'd:\testfile', 'file');
```

Example 3: Renaming a Directory

The following example renames a directory in the UNIX operating environment.

```
rc=rename('/local/u/testdir/', '/local/u/oldtestdir', 'file');
```

Example 4: Renaming a Generation Data Set

The following example renames the generation data set Work.One to Work.Two, where the password for Work.One#003 is *my-password*.

```
rc=rename('work.one', 'two',,, 3,'my-password');
```

See Also

Functions:

- “EXIST Function” on page 582
- “FDELETE Function” on page 599
- “FILEEXIST Function” on page 609

REPEAT Function

Returns a character value that consists of the first argument repeated n+1 times.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

REPEAT(*argument*, *n*)

Required Arguments

argument
specifies a character constant, variable, or expression.

n
specifies the number of times to repeat *argument*.

Restriction *n* must be greater than or equal to 0.

Details

In a DATA step, if the REPEAT 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.

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

```
data
one;

   x=repeat('ONE',
2);

   put
x=;

run;
```

These statements produce this result:

```
x=ONEONEONE
```

RESOLVE Function

Returns the resolved value of the argument after the argument has been processed by the macro facility.

Category: Macro

Note: The RESOLVE function cannot reference secure macros.

Syntax

RESOLVE(*argument*)

Required Argument

argument
is a character constant, variable, or expression with a value that is a macro expression.

Details

If the RESOLVE 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 bytes.

RESOLVE is fully documented in *SAS Macro Language: Reference*.

See Also

Functions:

- "SYMGET Function" on page 1432

REVERSE Function

Reverses a character string.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: DBCS equivalent function is KREVERSE in *SAS National Language Support (NLS): Reference Guide*.

Syntax

REVERSE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the REVERSE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The last character in the argument becomes the first character in the result, the next-to-last character in the argument becomes the second character in the result, and so on.

Note: Trailing blanks in the argument become leading blanks in the result.

Example

```
data
one;

   backward=reverse('xyz
');

   put
backward=;

run;
```

These statements produce this result:

```
backward=zyx
```

REWIND Function

Positions the data set pointer at the beginning of a SAS data set.

Category: SAS File I/O

Syntax

REWIND(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

Restriction The data set cannot be opened in IS mode.

Details

REWIND returns 0 if the operation was successful, ≠0 if it was not successful. After a call to REWIND, a call to FETCH reads the first observation in the data set.

If there is an active WHERE clause, REWIND moves the data set pointer to the first observation that satisfies the WHERE condition.

Example

This example calls FETCHOBS to fetch the tenth observation in the data set MYDATA. Next, the example calls REWIND to return to the first observation and fetch the first observation.

```
data
mydata1;

   set
sashelp.cars;

run;



%let dsid=%sysfunc(open(mydata1,
i));

%let rc=%sysfunc(fetchobs(&dsid,
10));

%let val=
%sysfunc(getvarc(&dsid,%sysfunc(varnum(&dsid,model))));

%put
&val;

%let rc=
%sysfunc(rewind(&dsid));

%let rc=
%sysfunc(fetch(&dsid));

%let val=
%sysfunc(getvarc(&dsid,%sysfunc(varnum(&dsid,model))));

%put
&val;

%let rc=%sysfunc(close(&dsid));
```

These statements produce these results:

```
A4 3.0 4dr
MDX
```

See Also

Functions:

- “FETCH Function” on page 600

- "FETCHOBS Function" on page 602
- "FREWIND Function" on page 737
- "NOTE Function" on page 1151
- "OPEN Function" on page 1183
- "POINT Function" on page 1221

RIGHT Function

Right aligns a character expression.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Tip: DBCS equivalent function is KRIGHT.

Syntax

RIGHT(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the RIGHT function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The RIGHT function returns an argument with trailing blanks moved to the start of the value. The length of the result is the same as the length of the argument.

Example

```
data new;
   length a $12 b $15;
   a='Due Date  ';
   b='*'||right(a);
   put a= $12. b= $15.;
```

```
run;
```

These SAS statements produce these results:

```
a=Due Date b=*    Due Date
```

See Also

Functions:

- “COMPRESS Function” on page 467
- “LEFT Function” on page 1048
- “TRIM Function” on page 1470

RMS Function

Returns the root mean square of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

RMS(*argument* <, *argument*, ..., *argument-n*>)

Required Argument

argument
is a numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the root mean square of the nonmissing values.

Let n be the number of arguments with nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The root mean square is

$$\sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}$$

Example

```
data
new;

   x1=rms(1,
7);

   x2=rms(., 1, 5,
11);

   x3=rms(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These SAS statements produce these results:

```
x1=5 x2=7 x3=6.0827625303
```

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.

Category: Rounding and Truncation

Syntax

ROUND(*argument* <, *rounding-unit*>)

Required Argument

argument
is a numeric constant, variable, or expression to be rounded.

Optional Argument

rounding-unit
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

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 result 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 you write these two values in SAS, they appear the same. If you compute the difference, however, you can see that the values are different.

```
data _null_;
   three=3;
   point_three=0.3;
   three_times_point_one=three*0.1;
   difference=point_three - three_times_point_one;
   put point_three= ;
   put three_times_point_one= ;
   put difference= ;
run;
```

SAS writes the following output to the log:

```
point_three=0.3
three_times_point_one=0.3
difference=-5.55112E-17
```

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 SAS program, the value is computed by the standard informat 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 SAS 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 SAS statements ...
```

If SAS 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 SAS 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 data sets 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 SAS 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 SAS statements ...
```

Producing Expected Results

In general, `ROUND(argument, 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. (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.)
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```
data rounding;
   d1=round(1234.56789, 100)     - 1200;
```

```
   d2=round(1234.56789, 10)      - 1230;
   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;
run;
proc print data=rounding noobs;
run;
```

Figure 3.3 *Output from Rounding Based on the Value of the Rounding Unit*

The SAS System

d1	d2	d3	d4	d5	d6	d7	d8	d9	d10
0	0	0	0	0	0	0	0	0	0

When the Rounding Unit Is the Reciprocal of an Integer

When the rounding unit is the reciprocal of an integer, the ROUND function computes the result by dividing by the integer. (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.) Therefore, you can safely compare the result from ROUND with the ratio of two integers, but not with a multiple of the rounding unit. For example:

```
data rounding2;
   drop pi unit;
   pi=arcos(-1);
   unit=1/7;
   d1=round(pi, unit) - 22/7;
   d2=round(pi, unit) - 22*unit;
run;
proc print data=rounding2 noobs;
run;
```

Figure 3.4 *Output from Rounding by the Reciprocal of an Integer*

The SAS System

d1	d2
0	1.1102E-16

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_;
   difference=round(1234.56789, .11111) - 11111*.11111;
   put difference=;
run;
```

SAS writes the following output to the log:

```
difference=0
```

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;
   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;
run;
proc print data=test noobs;
   format value 19.16;
run;
```

Figure 3.5 *Output from Rounding When Values Are Halfway between Multiples of the Rounding Unit*

The SAS System

i	value	round
8	0.4999999900000000	0
9	0.4999999990000000	0
10	0.4999999999000000	0
11	0.4999999999900000	0
12	0.4999999999990000	0
13	0.4999999999999000	1
14	0.4999999999999900	1
15	0.4999999999999900	1
16	0.5000000000000000	1
17	0.5000000000000000	1
8	-0.4999999900000000	0
9	-0.4999999990000000	0
10	-0.4999999999000000	0
11	-0.4999999999900000	0
12	-0.4999999999990000	0
13	-0.4999999999999000	-1
14	-0.4999999999999900	-1
15	-0.4999999999999900	-1
16	-0.5000000000000000	-1
17	-0.5000000000000000	-1

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 DATA step. This DATA step code will not always produce results exactly equivalent to the ROUND function.

```
data testfile;
   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;
run;
proc print data=testfile noobs;
   format value 19.16;
run;
```

Figure 3.6 *The Approximation is Relative to the Size of the Value to be Rounded*

i	value	epsilon	temp	fraction	round
1	0.4000000000000000	4E-13	0.90000	0.90000	0
2	0.4900000000000000	4.9E-13	0.99000	0.99000	0
3	0.4990000000000000	4.99E-13	0.99900	0.99900	0
4	0.4999000000000000	4.999E-13	0.99990	0.99990	0
5	0.4999900000000000	4.9999E-13	0.99999	0.99999	0
6	0.4999990000000000	5E-13	1.00000	1.00000	0
7	0.4999999000000000	5E-13	1.00000	1.00000	0
8	0.4999999900000000	5E-13	1.00000	1.00000	0
9	0.4999999990000000	5E-13	1.00000	1.00000	0
10	0.4999999999000000	5E-13	1.00000	1.00000	0
11	0.4999999999900000	5E-13	1.00000	1.00000	0
12	0.4999999999990000	5E-13	1.00000	1.00000	0
13	0.4999999999999000	5E-13	1.00000	0.00000	1
14	0.4999999999999900	5E-13	1.00000	0.00000	1
15	0.4999999999999900	5E-13	1.00000	0.00000	1
16	0.5000000000000000	5E-13	1.00000	0.00000	1
17	0.5000000000000000	5E-13	1.00000	0.00000	1

Comparisons

The ROUND, ROUNDE, and ROUNDZ functions are similar with four exceptions:

- ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDE returns an even multiple when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument.
- 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. ROUND and ROUNDE perform extra computations, called

fuzzing, to try to make the result agree with decimal arithmetic in the most common situations. ROUNDZ does not fuzz the result.

Example

The following example compares the results that are returned by the ROUND function with the results that are returned by the ROUNDE function.

```
data results;
   do x=0 to 4 by .25;
      Rounde=rounde(x);
      Round=round(x);
      output;
   end;
run;
proc print data=results noobs;
run;
```

Output 3.65 *Output That Is Returned by the ROUND and ROUNDE Functions*

The SAS System

x	Rounde	Round
0.00	0	0
0.25	0	0
0.50	0	1
0.75	1	1
1.00	1	1
1.25	1	1
1.50	2	2
1.75	2	2
2.00	2	2
2.25	2	2
2.50	2	3
2.75	3	3
3.00	3	3
3.25	3	3
3.50	4	4
3.75	4	4
4.00	4	4

See Also

Functions:

- "CEIL Function" on page 424
- "CEILZ Function" on page 426
- "FLOOR Function" on page 710
- "FLOORZ Function" on page 712
- "INT Function" on page 963
- "INTZ Function" on page 1020
- "ROUNDE Function" on page 1356
- "ROUNDZ Function" on page 1359

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.

Category: Rounding and Truncation

Syntax

ROUNDE(*argument* <, *rounding-unit*>)

Required Argument

argument
is a numeric constant, variable, or expression to be rounded.

Optional Argument

rounding-unit
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details

The ROUNDE function rounds the first argument to the nearest multiple of the second argument. If you omit the second argument, ROUNDE uses a default value of 1 for *rounding-unit*.

Comparisons

The ROUND, ROUNDE, and ROUNDZ functions are similar with four exceptions:

- ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDE returns an even multiple when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument.
- 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. ROUND and ROUNDE perform extra computations, called fuzzing, to try to make the result agree with decimal arithmetic in the most common situations. ROUNDZ does not fuzz the result.

Example

The following example compares the results that are returned by the ROUNDE function with the results that are returned by the ROUND function.

```
data results;
   do x=0 to 4 by .25;
      Rounde=rounde(x);
      Round=round(x);
      output;
   end;
run;
proc print data=results noobs;
run;
```

Output 3.66 *Output from the ROUNDE and ROUND Functions*

The SAS System

x	Rounde	Round
0.00	0	0
0.25	0	0
0.50	0	1
0.75	1	1
1.00	1	1
1.25	1	1
1.50	2	2
1.75	2	2
2.00	2	2
2.25	2	2
2.50	2	3
2.75	3	3
3.00	3	3
3.25	3	3
3.50	4	4
3.75	4	4
4.00	4	4

See Also

Functions:

- “CEIL Function” on page 424
- “CEILZ Function” on page 426
- “FLOOR Function” on page 710
- “FLOORZ Function” on page 712
- “INT Function” on page 963
- “INTZ Function” on page 1020
- “ROUND Function” on page 1348
- “ROUNDZ Function” on page 1359

ROUNDZ Function

Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.

Category: Rounding and Truncation

Syntax

ROUNDZ(*argument* <, *rounding-unit*>)

Required Argument

argument
is a numeric constant, variable, or expression to be rounded.

Optional Argument

rounding-unit
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details

The ROUNDZ function rounds the first argument to the nearest multiple of the second argument. If you omit the second argument, ROUNDZ uses a default value of 1 for *rounding-unit*.

Comparisons

The ROUND, ROUNDE, and ROUNDZ functions are similar with four exceptions:

- ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDE returns an even multiple when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument.
- 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. ROUND and ROUNDE perform extra computations, called

fuzzing, to try to make the result agree with decimal arithmetic in the most common situations. ROUNDZ does not fuzz the result.

Examples

Example 1: Comparing Results from the ROUNDZ and ROUND Functions

The following example compares the results that are returned by the ROUNDZ and the ROUND function.

```
data test;
   do i=10 to 17;
      Value=3.5 - 10**(-i);
      Roundz=roundz(value);
      Round=round(value);
      output;
   end;
   do i=16 to 12 by -1;
      value=3.5 + 10**(-i);
      roundz=roundz(value);
      round=round(value);
      output;
   end;
run;
proc print data=test noobs;
   format value 19.16;
run;
```

Output 3.67 *Output from the ROUNDZ and ROUND Functions*

The SAS System

i	Value	Roundz	Round
10	3.4999999999000000	3	3
11	3.4999999999900000	3	3
12	3.4999999999990000	3	4
13	3.4999999999999000	3	4
14	3.4999999999999900	3	4
15	3.5000000000000000	3	4
16	3.5000000000000000	4	4
17	3.5000000000000000	4	4
16	3.5000000000000000	4	4
15	3.5000000000000000	4	4
14	3.5000000000000100	4	4
13	3.5000000000001000	4	4
12	3.5000000000010000	4	4

Example 2: Output from the ROUNDZ Function

```
data
one;

input
val;

datalines;

223.456

;



data
new;

   set
one;


var1=val;
```

```
   a=roundz(var1,
1);


b=roundz(var1, .01);

   c=roundz(var1,
100);


d=roundz(var1);


e=roundz(var1, .3);

   put a= b= c= d=
e=;

run;
```

These statements produce these results:

```
a=223 b=223.46 c=200 d=223 e=223.5
```

See Also

Functions:

- “ROUND Function” on page 1348
- “ROUNDE Function” on page 1356

SAVING Function

Returns the future value of a periodic saving.

Category: Financial

Syntax

SAVING(*f*, *p*, *r*, *n*)

Required Arguments

f

is numeric, the future amount (at the end of *n* periods).

Range $f \geq 0$

p

is numeric, the fixed periodic payment.

Range $p \geq 0$

r

is numeric, the periodic interest rate expressed as a decimal.

Range $r \geq 0$

n

is an integer, the number of compounding periods.

Range $n \geq 0$

Details

The SAVING function returns the missing argument in the list of four arguments from a periodic saving. The arguments are related by

$$f = \frac{p(1+r)\left((1+r)^n - 1\right)}{r}$$

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 `number=saving(12000, 100, .05/12, .);`. The value 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.

```
data
new;

   number=saving(12000,
100, .05/12, .);

   put
number=;

run;
```

These statements produce this result:

```
number=97.18134646
```

SAVINGS Function

Returns the balance of a periodic savings by using variable interest rates.

Category: Financial

Syntax

SAVINGS(*base-date*, *initial-deposit-date*, *deposit-amount*, *deposit-number*, *deposit-interval*, *compounding-interval*, *date-1*, *rate-2* <, *date-2*, *rate-2*, ..., *date-n*, *rate-n*>)

Required Arguments

base-date
is a SAS date. The value that is returned is the balance of the savings at *base-date*.

initial-deposit-date
is a SAS date. *Initial-deposit-date* is the date of the first deposit. Subsequent deposits are at the beginning of subsequent deposit intervals.

deposit-amount
is numeric. All deposits are assumed constant. *deposit-amount* is the value of each deposit.

deposit-number
is a positive integer. *Deposit-number* is the number of deposits.

deposit-interval
is a SAS interval. *Deposit-interval* is the frequency at which deposits are made.

compounding-interval
is a SAS interval. *Compounding-interval* is the compounding interval.

date
is a SAS date. Each date is paired with a rate. *date* is the time at which *rate* takes effect.

rate
is a numeric percentage. Each rate is paired with a date. *rate* is the interest rate that starts on *date*.

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

```
data
new;

 /*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:*/


 amount_base1=SAVINGS("01jan2005"d, "01jan2000"d, 300, 24, "MONTH",
"QUARTER",
 "01jan2000"d, 4.00);


 /*If the interest rate increases by a quarter-point each year, then
the balance of
the account could be expressed as
follows:*/



 amount_base2=SAVINGS("01jan2005"d, "01jan2000"d, 300, 24, "MONTH",
"QUARTER",
"01jan2000"d, 4.00, "01jan2001"d, 4.25, "01jan2002"d,
 4.50, "01jan2003"d, 4.75, "01jan2004"d,
5.00);



 /*To determine the balance after one year of deposits, the following
statement
sets amount_base3 to the desired
balance:*/
```

```
 amount_base3=SAVINGS("01jan2001"d, "01jan2000"d, 300, 24, "MONTH",
"QUARTER",
"01jan2000"d, 4);


 put amount_base1= amount_base2=
amount_base3=;

run;
```

These statements produce these results:

```
amount_base1=8458.794159 amount_base2=8665.5059376 amount_base3=3978.6903712
```

The SAVINGS function ignores deposits after the base date, so the deposits after the reference date do not affect the value that is returned.

SCAN Function

Returns the *n*th word from a character string.

Category: Character

Restriction: This function is assigned an I18N Level 0 status and is designed for SBCS data. However, if the first argument, *string*, has multi-byte characters, then the SCAN function processes the DBCS data. For more information, see"Internationalization Compatibility for SAS String Functions" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Tip: The DBCS equivalent function is KSCAN.

Syntax

SCAN(*string*, *count* <, *character-list* <, *modifier*>>)

Required Arguments

string
specifies a character constant, variable, or expression.

count
is a nonzero numeric constant, variable, or expression that has an integer value. The integer value specifies the number of the word in the character string that you want SCAN to select. For example, a value of 1 indicates the first word, a value of 2 indicates the second word, and so on. The following rules apply:

- If *count* is positive, SCAN counts words from left to right in the character string.

- If *count* is negative, SCAN counts words from right to left in the character string.

Optional Arguments

character-list

specifies an optional character expression that initializes a list of characters. This list determines which characters are used as the delimiters that separate words. The following rules apply:

- By default, all characters in *character-list* are used as delimiters.
- Specifying a modifier can change the characters in *character-list*. For example, if you specify the K modifier in the *modifier* argument, all characters that are not in *character-list* are used as delimiters.

Note: For more information see "Using Default Delimiters in an ASCII Environment" on page 1369.

Tip You can add more characters to *character-list* by using other modifiers.

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 (that is, valid first characters in a SAS variable name by using VALIDVARNAME=V7) 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, then 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 (that is, the characters that can appear in a SAS variable name by using VALIDVARNAME=V7) 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.

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

Definition of “Delimiter” and “Word”

A *delimiter* is any of several characters that are used to separate words. You can specify the delimiters in the *charlist* 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:

- 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

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 an ASCII Environment

If you use the SCAN function with only two arguments and your computer uses ASCII characters, here are the default delimiters:

blank ! $ % & ( ) * + , - . / ; < ^ |

In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.

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 list of default delimiters for ASCII environments are not used. In other words, modifiers add to the list of delimiters that are explicitly specified by the *charlist* argument. Modifiers do not add to the list of default modifiers.

The Length of the Result

In a DATA step, most variables have a fixed length. If the word returned by the SCAN function is assigned to a variable that has a fixed length greater than the length of the returned word, then the value of that variable is padded with blanks. Macro variables have varying lengths and are not padded with blanks.

The maximum length of the word that is returned by the SCAN function depends on the environment from which it is called:

- In a DATA step, if the SCAN function returns a value to a variable that has not yet been given a length, that variable is given the length of the first argument.

This behavior is different from the behavior in previous releases of SAS. In previous releases, code that created a variable with a length of 200 might have produced a variable with a length that was greater than expected. If you need the SCAN function to assign a variable with a value that is different from the length of the first argument, use a LENGTH statement.

If you use the SCAN function in an expression that contains operators or other functions, a word that is returned by the SCAN function can have a length of up to 32,767 characters, except in a WHERE clause. In that case, the maximum length is 200 characters.

- In the SQL procedure, or in a WHERE clause in any procedure, the maximum length of a word that is returned by the SCAN function is 200 characters.
- In the macro processor, the maximum length of a word that is returned by the SCAN function is 65,534 characters.

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 1370. See also "Using the SCAN Function without the M Modifier" on page 1370.

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 behaves in these ways:

- ignores delimiters at the beginning or end of the string
- 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 *string* 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 multi-byte 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 firstlast;
   input String $60.;
   First_Word=scan(string, 1);
   Last_Word=scan(string, -1);
   datalines4;
Jack and Jill
& Bob & Carol & Ted & Alice &
Leonardo
! $ % & ( ) * + , - . / ;
;;;;
proc print data=firstlast;
run;
```

Output 3.68 *Output from Finding the First and Last Words in a String*

The SAS System

Obs	String	First_Word	Last_Word
1	Jack and Jill	Jack	Jill
2	& Bob & Carol & Ted & Alice &	Bob	Alice
3	Leonardo	Leonardo	Leonardo
4	! $ % & ( ) * + , - . / ;		

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.

```
data all;
   length word $20;
   drop string;
   string=' The quick brown fox jumps over the lazy dog.   ';
   do until(word=' ');
      count+1;
      word=scan(string, count);
      output;
   end;
run;
proc print data=all noobs;
run;
```

Output 3.69 *Output from Finding All Words without Using the M Modifier*

The SAS System

word	count
The	1
quick	2
brown	3
fox	4
jumps	5
over	6
the	7
lazy	8
dog	9
	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.

```
data comma;
   keep count word;
   length word $30;
   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;
run;
proc print data=comma noobs;
run;
```

Output 3.70 *Output from Finding All Words By Using the M and O Modifiers*

The SAS System

word	count
	1
leading	2
trailing	3
and multiple	4
	5
delimiters	6
	7
	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.

```
data test;
   keep count word word_r;
   length word word_r $30;
   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;
run;
proc print data=test noobs;
run;
```

Output 3.71 *Output from Comma-Separated Values and Substrings in Double Quotation Marks*

The SAS System

word	word_r	count
He said	He said	1
"She said, ""No!"""	She said, "No!"	2
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.

```
data digits;
   keep count digits;
   length digits $20;
   string='Call (800) 555-1234 now!';
   do until(digits=' ');
      count+1;
      digits=scan(string, count, , 'dko');
      output;
   end;
run;
proc print data=digits noobs;
run;
```

Output 3.72 *Output from Finding Substrings of Digits By Using the D and K Modifiers*

The SAS System

digits	count
800	1
555	2
1234	3
	4

See Also

Functions:

- "COUNTW Function"
- "FINDW Function"
- "KSCAN Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*
- "KSCANX Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*

CALL Routines:

- "CALL SCAN Routine" on page 326

SDF Function

Returns a survival function.

Category: Probability

See: "CDF Function" on page 392

Syntax

SDF(*distribution*, *quantile*, *parameter-1, ..., parameter-k*)

Required Arguments

distribution
is a character string that identifies the distribution.

Valid distributions are as follows:

Distribution	Argument
Bernoulli	`BERNOULLI`
Beta	`BETA`
Binomial	`BINOMIAL`
Cauchy	`CAUCHY`
Chi-Square	`CHISQUARE`

Distribution	Argument
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.

parameter-1, ..., parameter-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The SDF function computes the survival function (upper tail) from various continuous and discrete distributions. For more information, see "CDF Function" on page 392.

The SDF function for the Conway-Maxwell-Poisson distribution has the following form:

`SDF('CONMAXPOI',`y, λ, ν`)`

y is a nonnegative integer that represents counts data. λ is similar to the mean, as in the Poisson distribution. ν 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 1198.

Example

```
data
new;

   a=sdf('BERN',
0, .25);

   b=sdf('BETA', 0.2, 3,
4);

   c=sdf('BINOM', 4, .5,
10);

   d=sdf('CAUCHY',
2);

   e=sdf('CHISQ', 11.264,
11);

   f=sdf('CONMAXPOI', 12,
2.3, .4);

   g=sdf('EXPO',
1);

   h=sdf('F', 3.32,
2,3);

   i=sdf('GAMMA', 1,
3);
```

```
   j=sdf('GENPOISSON', .9,
1, .7);

   k=sdf('GEOMETRIC',
5, .3);

   l=sdf('HYPER', 2, 200, 50,
10);

   m=sdf('LAPLACE',
1);

   n=sdf('LOGISTIC',
1);

   o=sdf('LOGNORMAL',
1);

   p=sdf('NEGB', 1, .5,
2);

   q=sdf('NORMAL',
1.96);

   r=sdf('NORMALMIX', 2.3, 3, .33, .33, .34, .5, 1.5, 2.5, .79, 1.6,
4.3);
   s=sdf('PARETO', 1,
1);

   t=sdf('POISSON', 2,
1);

   u=sdf('T', .9,
5);

   v=sdf('TWEEDIE', .8,
5);

   w=sdf('UNIFORM',
0.25);

   x=sdf('WALD', 1,
2);

   y=sdf('WEIBULL', 1,
2);

   put
_all_;

run;
```

These statements produce these results:

```
a=0.25 b=0.90112 c=0.623046875 d=0.1475836177 e=0.4214186707 f=0.1970513877
g=0.3678794412
h=0.1736066398 i=0.9196986029 j=0.6321205588 k=0.117649 l=0.4763265919
m=0.1839397206
n=0.2689414214 o=0.5 p=0.5 q=0.0249978951 r=0.2818691277 s=1 t=0.0803013971
u=0.2046856002
v=0.4082370836 w=0.75 x=0.3723021618 y=0.3678794412
```

See Also

Functions:

- “CDF Function” on page 392
- “LOGCDF Function” on page 1081
- “LOGPDF Function” on page 1084
- “LOGSDF Function” on page 1087
- “PDF Function” on page 1193
- “QUANTILE Function” on page 1293
- “SQUANTILE Function” on page 1404

SEC Function

Returns the secant.

Category: Trigonometric

Syntax

SEC(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression and is expressed in radians.

Restriction *argument* cannot be an odd multiple of PI/2.

Comparisons

The SEC function is related to the COS function:

```
sec(x)=1/cos(x)
```

Example

```
data
new;


x=sec(0.5);


y=sec(0);


z=sec(3.14159/3);

   put x= y=
z=;

run;
```

These statements produce these results:

```
x=1.1394939273 y=1 z=1.9999969359
```

See Also

Functions:

- “COS Function” on page 485
- “COT Function” on page 487
- “CSC Function” on page 499
- “SIN Function” on page 1390
- “TAN Function” on page 1447

SECOND Function

Returns the seconds and milliseconds from a SAS time or datetime value.

Category: Date and Time

Syntax

SECOND(*time* | *datetime*)

Required Arguments

time
is a numeric constant, variable, or expression with a value that represents a SAS time value.

datetime
is a numeric constant, variable, or expression with a value that represents a SAS datetime value.

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.

The SECOND function returns seconds and milliseconds. The first three digits of the milliseconds are relevant.

Examples

Example 1

This example specifies the second of the minute that is associated with datetime. The first three digits for the milliseconds, 470, are relevant.

```
data_null;
   x=second(datetime());
   put x=;
run;
```

These statements produce this result:

```
x=58.470999956
```

Example 2

```
data one;
input tm time.;
datalines;
3:19:24
6:25:65
```

```
3:19:60;
data new;
   set one;
   s=second(tm);
   put s=;
run;
```

These statements produce these results:

```
s=24
s=5
s=0
```

See Also

Functions:

- “HOUR Function” on page 934
- “MINUTE Function” on page 1107

SHA256 Function

Returns an SHA256 message digest as a 32-byte binary string for a message consisting of a character string.

Categories: Binary Results
Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

See: “Hashing Functions and Hash-Based Message Authentication Code” on page 36 for information about hashing functions and HMAC.

Syntax

SHA256(*message*)

Required Argument

message
specifies a character constant, variable, or expression.

Note: The message can be a VARCHAR and can be any length. If the message is a character constant and contains invalid characters, write it as a hexadecimal constant.

Details

Results from the SHA256 function depend on the character encoding of the message. For example, 'abc' is '616263'x in ASCII but '818283'x in EBCDIC, which results in different digests.

The SHA256 function returns a binary value, so the result might contain unprintable characters. You can print the result in a readable form by using the $HEX64. format.

Example: Generating Results with the SHA256 Function

This example generates results that are returned by the SHA256 function:

```
data _null_;
   y=sha256('abc');
   z=sha256('access method');
   put y=$hex64.;
   put z=$hex64.;
run;
```

These statements produce these results for ASCII systems:

```
y=BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
z=F2758E91725621F59F2F80D15DE8824560EDC471EBE40A83BA6D1259B1605915
```

SHA256HEX Function

Returns the SHA256 digest for a specified message, and the digest is provided in hexadecimal representation.

Category: Character

Note: UTF-8 text is recommended for the SHA256HEX function arguments to ensure consistency across encodings. See "Example 2: Using the KCVT Function to Transcode to UTF-8" on page 1386 for an example that uses the KCVT function to transcode data to UTF-8.

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

SHA256HEX('*message*', *flag*)

Required Arguments

'message'
specifies a character constant, variable, or expression.

Note: The message can be a varchar and can be any length. If a varchar is used, the value must be in hexadecimal representation if the message contains any invalid UTF8 characters.

flag
indicates whether the argument *message* is regular characters or hexadecimal representation characters.

0 indicates that the expression in the argument *message* is regular characters.

1 indicates that the expression in the argument *message* 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, or A–F. Blanks in the hexadecimal representation string are ignored.

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.

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 $BINARY*w*. or $HEX*w*. formats to view readable results. The SHA256HEX function formats its output, so you do not have to use the $BINARY or $HEX formats.

Examples

Example 1: Generating Results with the SHA256HEX Function

This example generates results that are returned by the SHA256HEX function.

```
data _null_;
   y=sha256hex('abc');
   z=sha256hex('access method');
   put y=;
   put z=;
run;
```

The following output is displayed for ASCII systems:

```
y=BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
z=F2758E91725621F59F2F80D15DE8824560EDC471EBE40A83BA6D1259B1605915
```

Example 2: Using the KCVT Function to Transcode to UTF-8

The following example demonstrates how to use the KCVT function to transcode to UTF-8. The second digest matches between running UTF-8, which is the first section, and EUC-CN, which is the second section.

```
2   +data _null_;
3   +digest = sha256hex('为复兴社会');
4   +put digest=;
5   +digest = sha256hex(kcvt('为复兴社会','utf-8'));
6   +put digest=;
7   +run;

digest=77C731E15B8C8FAA0FB5ECA8579DFC66B1505B089D71A4141944C03188DBE355
digest=77C731E15B8C8FAA0FB5ECA8579DFC66B1505B089D71A4141944C03188DBE355
```

```
2   +data _null_;
3   +digest = sha256hex('为复兴社会');
4   +put digest=;
5   +digest = sha256hex(kcvt('为复兴社会','utf-8'));
6   +put digest=;
7   +run;

digest=9521159BD38E15FD642260DB1EA72815DAE2FA465144CC5F3C6BD1A38EDF94A2
digest=77C731E15B8C8FAA0FB5ECA8579DFC66B1505B089D71A4141944C03188DBE355
```

See Also

- "SHA256 Function" on page 1383
- "SHA256HMACHEX Function" on page 1387

SHA256HMACHEX Function

Returns the result of the message digest of a specified string using the HMAC algorithm.

Category: Character

Notes: The SHA256HMACHEX function verifies the data integrity and authentication of a message.

For more information, see Hash-based message authentication code (HMAC).

UTF-8 text is recommended for the SHA256HMACHEX function arguments to ensure consistency across encodings. See "Example 2: Using the KCVT Function to Transcode to UTF-8" on page 1386 for an example that uses the KCVT function to transcode data to UTF-8.

See: "Hashing Functions and Hash-Based Message Authentication Code" on page 36 for information about hashing functions and HMAC.

Syntax

SHA256HMACHEX('*key*', '*message*' <, *flag*>);

Required Arguments

key
specifies a character constant, variable, or expression.

message
specifies a secret key padded to the right with extra zeros to the input block size of the hash function.

Note: For SAS Viya platform, the message can be a varchar and can be any length. If a varchar is used, the value must be in hexadecimal representation if the message contains any invalid UTF8 characters.

Optional Argument

flag
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);
```

Details

The SHA256HMACHEX function converts a string, based on the SHA256 algorithm, to a 256-bit hash value.

Example: Generating Results with the SHA256HMACHEX Function

This example generates results that are returned by the SHA256HMACHEX function.

```
data _null_;
   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';
run;
```

The following result is generated from the code:

```
matched
```

See Also

"SHA256HEX Function" on page 1384

SIGN Function

Returns the sign of a value.

Category: Mathematical

Syntax

SIGN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The SIGN function returns the following values:

-1
if *argument* < 0

0
if *argument* = 0

1
if *argument* > 0.

Example

```
data
new;


x=sign(-5);


y=sign(5);


z=sign(0);

   put x= y=
z=;

run;
```

These statements produce these results:

```
x=-1 y=1 z=0
```

SIN Function

Returns the sine.

Category: Trigonometric

Syntax

SIN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression and is expressed in radians. If the magnitude of *argument* is so great that `mod(argument,pi)` is accurate to less than about three decimal places, SIN returns a missing value.

Example

```
data
new;


x=sin(0.5);


y=sin(0);


z=sin(3.14159/4);

   put x= y=
z=;

run;
```

These statements produce these results:

```
x=0.4794255386 y=0 z=0.7071063121
```

SINH Function

Returns the hyperbolic sine.

Category: Hyperbolic

Syntax

SINH(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The SINH function returns the hyperbolic sine of the argument, which is given by

$$(\varepsilon^{argument} - \varepsilon^{-argument})/2$$

Example

```
data
new;


x=sinh(0);

   y=sinh(1);

z=sinh(-1.0);

   put x= y= z=;
run;
```

These statements produce these results:

```
x=0 y=1.1752011936 z=-1.175201194
```

SKEWNESS Function

Returns the skewness of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

SKEWNESS(*argument-1*, *argument-2*, *argument-3* <,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

At least three nonmissing arguments are required. Otherwise, the function returns a missing value. If all nonmissing arguments have equal values, the skewness is mathematically undefined. The SKEWNESS function returns a missing value and sets _ERROR_ equal to 1.

The argument list can consist of a variable list.

Example

```
data
new;

   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=;

run;
```

These statements produce these results:

```
x1=-1.732050808
x2=0.5901286564
x3=1.7320508076
x4=-0.953097714
```

SLEEP Function

Suppresses the execution of a program that invokes this function for a period of time.

Category: Special

Syntax

SLEEP(*n* <, *unit*>)

Required Argument

n

is a numeric constant, variable, or expression that specifies the number of seconds for which you want to suspend execution of a program. Negative or missing values for *n* are invalid.

Range $n \geq 0$

Tips 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(30.25);` suspends execution for 30 milliseconds.

If you do not use the *unit* argument, *n* is the next integer value that is less than n.

Optional Argument

unit
specifies the unit of time in seconds as a multiple of 10, which is applied to n. For example, 1 corresponds to 1 second; .001 corresponds to 1 millisecond; and 5 corresponds to 5 seconds.

Details

The SLEEP function suspends the execution of a program that invokes this function for a period of time that you specify. The program can be a DATA step, macro, IML, SCL, or any program that can invoke a function.

The SLEEP function suspends the execution of a program and returns the value of the first argument.

When you submit a program that calls the SLEEP function, the SLEEP window appears, telling you when SAS is going to wake up. You can inhibit the SLEEP window by starting SAS with the NOSLEEPWINDOW system option. 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 step to call the SLEEP function; follow this DATA step 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 example tells SAS to delay the execution of the DATA step PAYROLL for 20 seconds:

```
data payroll;
   time_slept=sleep(20,1);
   ...more SAS statements...
run;
```

Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following example tells SAS to suspend the execution of the DATA step BUDGET until March 1, 2013, at 3:00 AM. SAS calculates the length of the suspension based on the target date and the date and time that the DATA step begins to execute.

```
data budget;
   sleeptime='01mar2013:03:00'dt-'01mar2013:2:59:30'dt;
   time_calc=sleep(sleeptime,1);
   put 'Calculation of sleep time:';
   put sleeptime='seconds';
run;
```

These statements produce this result:

```
Calculation of sleep time:
sleeptime=30 seconds
```

See Also

CALL Routines:

- "CALL SLEEP Routine" on page 338

SMALLEST Function

Returns the *k*th smallest nonmissing value.

Category: Descriptive Statistics

Syntax

SMALLEST(*k*, *value-1* <, *value-2*..., *value-n*>)

Required Arguments

k
is a numeric constant, variable, or expression that specifies which value to return.

value
specifies a numeric constant, variable, or expression.

Details

If *k* is missing, less than zero, or greater than the number of values, the result is a missing value and _ERROR_ is set to 1. Otherwise, if *k* is greater than the number of nonmissing values, the result is a missing value, but _ERROR_ is not set to 1.

Comparisons

The SMALLEST function differs from the ORDINAL function in that the SMALLEST function ignores missing values. The ORDINAL function counts missing values.

Example

This example compares the values that are returned by the SMALLEST function with the values that are returned by the ORDINAL function.

```
data comparison;
   label smallest_num='SMALLEST Function' ordinal_num='ORDINAL
Function';
   do k = 1 to 4;
      smallest_num=smallest(k, 456, 789, .Q, 123);
      ordinal_num=ordinal (k, 456, 789, .Q, 123);
      output;
   end;
run;
proc print data=comparison label noobs;
   var k smallest_num ordinal_num;
   title 'Results from the SMALLEST and the ORDINAL Functions';
run;
```

Output 3.73 *Comparison of Values: The SMALLEST and ORDINAL Functions*

Results From the SMALLEST and the ORDINAL Functions

k	SMALLEST Function	ORDINAL Function
1	123	Q
2	456	123
3	789	456
4	.	789

See Also

Functions:

- “LARGEST Function” on page 1042
- “ORDINAL Function” on page 1187
- “PCTL Function” on page 1190

SORT Function

Sorts a list of variables.

Categories: Character
Sort

Alias: SORTC, SORTN

Syntax

SORT(*variable*)

Required Argument

variable
specifies a character constant, variable, or expression.

The SORT variables can be the following:

- numeric variables
- INT64, if the DATA step supports INT64
- UINT64, if the DATA step supports UINT64
- character variables with the same length
- VARCHAR variables with the same encoding.

Details

The SORT function returns a 1 if the function is successful and a 0 is the function is not successful.

Examples

Example 1

This example demonstrates the SORT function with numeric *variable* values.

```
data _null_;
   x=2;
   y=3;
   z=1;
   j=sort(x,y,z);
   put j= x= y= z=;
run;
```

These statements produce these results:

```
j=1 x=1 y=2 z=3
```

Example 2

This example demonstrates the SORT function with character *variable* values.

```
data _null_;
   length x y z $3;
   x='bb';
   y='ccc';
   z='a';
   j=sort(x,y,z);
   put j= x= y= z=;
run;
```

These statements produce these results:

```
j=1 x=a y=bb z=ccc
```

Example 3

This example demonstrates the SORT function with varchar *variable* values.

```
data _null_;
   length x y z varchar(*);
   x='bb'; y='ccc'; z='a';
   j=sort(x,y,z);
   put j= x= y= z=;
run;
```

These statements produce these results:

```
j=1 x=a y=bb z=ccc
```

SOUNDEX Function

Encodes a string to facilitate searching.

Category: Character

Restrictions: SOUNDEX algorithm is English-biased.

This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Syntax

SOUNDEX(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the SOUNDEX 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.

The Basics

The SOUNDEX function encodes a character string according to an algorithm that was originally developed by Margaret K. Odell and Robert C. Russell (US Patents 1261167 (1918) and 1435663 (1922)). The algorithm is described in *The Art of Computer Programming, Volume 3: Sorting and Searching* (Knuth 1998). (See "References" on page 1609.) The SOUNDEX algorithm is English-biased and is less useful for languages other than English.

The SOUNDEX function returns a copy of the *argument* that is encoded by using these steps:

1 SOUNDEX retains the first letter in the *argument* and discards these letters:

A E H I O U W Y

2 SOUNDEX assigns the following numbers to these classes of letters:

- 1: B F P V

- 2: C G J K Q S X Z
- 3: D T
- 4: L
- 5: M N
- 6: R

3 If two or more adjacent letters have the same classification from step 2, discard all but the first letter. (Adjacent refers to the position in the word before you discard the letters.)

The algorithm that is described in Knuth adds trailing zeros and truncates the result to the length of 4. You can perform these operations with other SAS functions.

Example

```
data
new;


x=soundex('Paul');

   put
x=;


word='amnesty';


x=soundex(word);

   put
x=;

run;
```

These statements produce these results:

```
x=P4
x=A523
```

SPEDIS Function

Determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Syntax

SPEDIS(*query*, *keyword*)

Required Arguments

query
identifies the word to query for the likelihood of a match. SPEDIS removes trailing blanks before comparing the value.

keyword
specifies a target word for the query. SPEDIS removes trailing blanks before comparing the value.

Details

Length of Returned Variable

In a DATA step, if the SPEDIS 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.

The Basics

SPEDIS returns the distance between the query and a keyword, a nonnegative value that is usually less than 100 but never greater than 200 with the default costs.

SPEDIS computes an asymmetric spelling distance between two words as the normalized cost for converting the keyword to the query word by using a sequence of operations. SPEDIS(*QUERY*, *KEYWORD*) is *not* the same as SPEDIS(*KEYWORD*, *QUERY*).

Costs for each operation that is required to convert the keyword to the query are listed in the following table:

Operation	Cost	Explanation
match	0	no change
singlet	25	delete one of a double letter
doublet	50	double a letter
swap	50	reverse the order of two consecutive letters

Operation	Cost	Explanation
truncate	50	delete a letter from the end
append	35	add a letter to the end
delete	50	delete a letter from the middle
insert	100	insert a letter in the middle
replace	100	replace a letter in the middle
firstdel	100	delete the first letter
firstins	200	insert a letter at the beginning
firstrep	200	replace the first letter

The distance is the sum of the costs divided by the length of the query. If this ratio is greater than one, the result is rounded down to the nearest whole number.

Comparisons

The SPEDIS function is similar to the COMPLEV and COMPGED functions, but COMPLEV and COMPGED are much faster, especially for long strings.

Example

Here is an example of using the SPEDIS function.

```
data words;
   input Operation $ Query $ Keyword $;
   Distance=spedis(query, keyword);
   Cost=distance * length(query);
   datalines;
match       fuzzy        fuzzy
singlet     fuzy         fuzzy
doublet     fuuzzy       fuzzy
swap        fzuzy        fuzzy
truncate    fuzz         fuzzy
append      fuzzys       fuzzy
delete      fzzy         fuzzy
insert      fluzzy       fuzzy
replace     fizzy        fuzzy
firstdel    uzzy         fuzzy
firstins    pfuzzy       fuzzy
firstrep    wuzzy        fuzzy
several     floozy       fuzzy
```

```
;
proc print data=words;
run;
```

Output 3.74 *Costs for SPEDIS Operations*

The SAS System

Obs	Operation	Query	Keyword	Distance	Cost
1	match	fuzzy	fuzzy	0	0
2	singlet	fuzy	fuzzy	6	24
3	doublet	fuuzzy	fuzzy	8	48
4	swap	fzuzy	fuzzy	10	50
5	truncate	fuzz	fuzzy	12	48
6	append	fuzzys	fuzzy	5	30
7	delete	fzzy	fuzzy	12	48
8	insert	fluzzy	fuzzy	16	96
9	replace	fizzy	fuzzy	20	100
10	firstdel	uzzy	fuzzy	25	100
11	firstins	pfuzzy	fuzzy	33	198
12	firstrep	wuzzy	fuzzy	40	200
13	several	floozy	fuzzy	50	300

See Also

Functions:

- “COMPGED Function” on page 456
- “COMPLEV Function” on page 463
- “SOUNDEX Function” on page 1399

SQRT Function

Returns the square root of a value.

Category: Mathematical

Syntax

SQRT(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression. *Argument* must be nonnegative.

Example

```
data
new;


a=sqrt(36);


b=sqrt(25);


c=sqrt(4.4);

   put a=;
   put b=;
   put
c=;

run;
```

These statements produce these results:

```
a=6
b=5
c=2.0976176963
```

SQUANTILE Function

Returns the quantile from a distribution when you specify the right probability (SDF).

Category: Quantile

See: "SDF Function" on page 1376

Syntax

SQUANTILE(*distribution*, *probability*, *parameter-1, ..., parameter-k*)

Required Arguments

distribution

is a character constant, variable, or expression that identifies the distribution.

Valid distributions are as follows:

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

Distribution	Argument
Pareto	PARETO
Poisson	POISSON
T	T
Tweedie	TWEEDIE
Uniform	UNIFORM
Wald (inverse Gaussian)	WALD|IGAUSS
Weibull	WEIBULL

Note: Except for F, NORMALMIX, and T, 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 that are appropriate for the specific distribution.

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 "Details" on page 394 in the CDF function.

The SQUANTILE function for the Conway-Maxwell-Poisson distribution returns the smallest integer whose SDF value is less than or equal to 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 1198.

For more information about the distributions that are listed in the table, see "PDF Function" on page 1193.

Examples

Example 1: Using the LOGISTIC Distribution

```
data;
   dist='logistic';
   sdf=squantile(dist, 1.e-20);
   put sdf=;
   p=sdf(dist, sdf);
   put p=/* p will be 1.e-20 */;
run;
```

These statements produce these results:

```
sdf=46.05170186
p=1E-20
```

Example 2: Using the Conway-Maxwell-Poisson Distribution

```
data _null_;
   y=squantile('conmaxpoi',.2,2.3,.4);
   put y=;
run;
```

These statements produce this result:

```
y=12
```

See Also

Functions:

- "CDF Function" on page 392
- "LOGCDF Function" on page 1081
- "LOGPDF Function" on page 1084
- "LOGSDF Function" on page 1087
- "PDF Function" on page 1193
- "QUANTILE Function" on page 1293
- "SDF Function" on page 1376

STD Function

Returns the standard deviation of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

STD(*argument-1*, *argument-2* <, ... *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Example

```
data
one;

   input val1 val2 val3
val4;


datalines;

2 4 6
8

;



data
new;

   set
one;


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=;

run;
```

These statements produce these results:

```
x1=2.8284271247
x2=2.8284271247
x3=1.9235384062
x4=0.5224377453
```

STDERR Function

Returns the standard error of the mean of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

STDERR(*argument-1*, *argument-2* <, ...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Example

```
data
one;

   input val1 val2 val3
val4;
```

```
datalines;

2 4 6
8

;



data
new;

   set
one;


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=;

run;
```

These statements produce these results:

```
x1=2
x2=2
x3=0.8602325267
x4=0.3799224911
```

STFIPS Function

Converts state postal codes to FIPS state codes.

Category: State and ZIP code

Syntax

STFIPS(*postal-code*)

Required Argument

postal-code
specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

The STFIPS function converts a two-character state postal code (or worldwide GSA geographic code for U.S. territories) to the corresponding numeric U.S. Federal Information Processing Standards (FIPS) code.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed-case state name.

Example

SAS writes the following output to the log:

```
data
new;


fips=stfips('NC');

   put
fips=;




state=stname('NC');

   put
state=;
```

```
state=stnamel('NC');

   put
state=;

run;
```

These statements produce these results:

```
fips=37
state=NORTH CAROLINA
state=North Carolina
```

See Also

Functions:

- “FIPNAME Function” on page 703
- “FIPNAMEL Function” on page 705
- “FIPSTATE Function” on page 707
- “STNAME Function” on page 1412
- “STNAMEL Function” on page 1414

STNAME Function

Converts state postal codes to uppercase state names.

Category: State and ZIP code

Syntax

STNAME(*postal-code*)

Required Argument

postal-code
specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

The STNAME function converts a two-character state postal code (or worldwide GSA geographic code for U.S. territories) to the corresponding state name in uppercase.

Note: For Version 6, the maximum length of the value that is returned is 200 characters. For Version 7 and later, the maximum length is 20 characters.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed-case state name.

Example

```
data
new;


fips=stfips('NC');

   put
fips=;




state=stname('NC');

   put
state=;



state=stnamel('NC');

   put
state=;

run;
```

These statements produce these results:

```
fips=37
state=NORTH CAROLINA
state=North Carolina
```

See Also

Functions:

- “FIPNAME Function” on page 703
- “FIPNAMEL Function” on page 705
- “FIPSTATE Function” on page 707
- “STFIPS Function” on page 1410
- “STNAMEL Function” on page 1414

STNAMEL Function

Converts state postal codes to mixed case state names.

Category: State and ZIP code

Syntax

STNAMEL(*postal-code*)

Required Argument

postal-code
specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

If the STNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The STNAMEL function converts a two-character state postal code (or worldwide GSA geographic code for U.S. territories) to the corresponding state name in mixed case.

Note: For Version 6, the maximum length of the value that is returned is 200 characters. For Version 7 and beyond, the maximum length is 20 characters.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed case state name.

Example

The following examples show the differences when using STFIPS, STNAME, and STNAMEL:

```
data
new;


fips=stfips('NC');

   put
fips=;




state=stname('NC');

   put
state=;




state=stnamel('NC');

   put
state=;

run;
```

These statements produce these results:

```
fips=37
state=NORTH CAROLINA
state=North Carolina
```

See Also

Functions:

- “FIPNAME Function” on page 703
- “FIPNAMEL Function” on page 705
- “FIPSTATE Function” on page 707
- “STFIPS Function” on page 1410
- “STNAME Function” on page 1412

STRIP Function

Returns a character string with all leading and trailing blanks removed.

Category: Character

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

STRIP(*string*)

Required Argument

string
is a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the STRIP 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.

The Basics

The STRIP function returns the argument with all leading and trailing blanks removed.

Assigning the results of STRIP to a variable does not affect the length of the receiving variable. 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 leading or trailing blanks.

Comparisons

The following list compares the STRIP function with the TRIM and TRIMN functions:

- For strings that are blank, the STRIP and TRIMN functions return a string with a length of zero, whereas the TRIM function returns a single blank.
- For strings that lack leading blanks, the STRIP and TRIMN functions return the same value.
- For strings that lack leading blanks but have at least one non-blank character, the STRIP and TRIM functions return the same value.

Note: `STRIP(`*`string`*`)` returns the same result as `TRIMN(LEFT(`*`string`*`))`, but the STRIP function runs faster.

Example

This example shows the results of using the STRIP function to delete leading and trailing blanks.

```
data lengthn;
   input string $char8.;
   original='*' || string || '*';
   stripped='*' || strip(string) || '*';
   datalines;
abcd
  abcd
    abcd
abcdefgh
 x y z
;
proc print data=lengthn;
run;
```

Output 3.75 *Output from the STRIP Function*

The SAS System

Obs	string	original	stripped
1	abcd	*abcd *	*abcd*
2	abcd	* abcd *	*abcd*
3	abcd	* abcd*	*abcd*
4	abcdefgh	*abcdefgh*	*abcdefgh*
5	x y z	* x y z *	*x y z*

See Also

Functions:

- “CAT Function” on page 375
- “CATS Function” on page 383
- “CATT Function” on page 385
- “CATX Function” on page 388
- “LEFT Function” on page 1048
- “TRIM Function” on page 1470
- “TRIMN Function” on page 1472

SUBPAD Function

Returns a substring that has a length that you specify, using blank padding if necessary.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

SUBPAD(*string*, *position* <, *length*>)

Required Arguments

string
specifies a character constant, variable, or expression.

position
is a positive integer that specifies the position of the first character in the substring.

Optional Argument

length
is a nonnegative integer that specifies the length of the substring. If you do not specify *length*, the SUBPAD function returns the substring that extends from the position that you specify to the end of the string.

Details

In a DATA step, if the SUBPAD 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 substring that you specify extends beyond the length of the string, the result is padded with blanks.

Comparisons

The SUBPAD function is similar to the SUBSTR function except for the following differences:

- If the value of *length* in SUBPAD is zero, SUBPAD returns a zero-length string. If the value of *length* in SUBSTR is zero, SUBSTR
  - writes a note to the log stating that the third argument is invalid
  - sets _ERROR_=1
  - returns the substring that extends from the position that you specified to the end of the string.
- If the substring that you specify extends past the end of the string, SUBPAD pads the result with blanks to yield the length that you requested. If the substring that you specify extends past the end of the string, SUBSTR
  - writes a note to the log stating that the third argument is invalid
  - sets _ERROR_=1
  - returns the substring that extends from the position that you specified to the end of the string.

See Also

Functions:

- "SUBSTRN Function" on page 1424

SUBSTR (left of =) Function

Replaces character value contents.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: The DBCS equivalent functions are KUPDATE and KUPDATEB.

Syntax

SUBSTR(*variable*, *position* <, *length*>)=*characters-to-replace*

Required Arguments

variable
specifies a character variable.

position
specifies a numeric constant, variable, or expression that is the beginning character position.

characters-to-replace
specifies a character constant, variable, or expression that replaces the contents of *variable*.

Tip Enclose a literal string of characters in quotation marks.

Optional Argument

length
specifies a numeric constant, variable, or expression that is the length of the substring that is replaced.

Restriction *length* cannot be larger than the length of the expression that remains in *variable* after *position*.

Tip If you omit *length*, SAS uses all of the characters on the right side of the assignment statement to replace the values of *variable*.

Details

If you use an undeclared variable, it is assigned a default length of 8 when the SUBSTR function is compiled.

When you use the SUBSTR function on the left side of an assignment statement, SAS replaces the value of *variable* with the expression on the right side. SUBSTR replaces *length* characters, starting at the character that you specify in *position*.

The SUBSTR function and INDEX functions return a character position when using a varchar column. The SUBSTR function and INDEX functions return a byte position when using a character column.

Example

```
data
new;


a='KIDNAP';

   substr(a, 1,
3)='CAT';

   put
a=;




b=a;

   substr(b,
4)='TY';

   put
b=;

run;
```

These statements produce these results:

```
a=CATNAP
b=CATTY
```

See Also

Functions:

- "SUBSTR (right of =) Function" on page 1422

SUBSTR (right of =) Function

Extracts a substring from an argument.

Category: Character

Restrictions: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

This function supports the VARCHAR type.

Tip: The DBCS equivalent functions are KSUBSTR and KSUBSTRB.

Syntax

SUBSTR(*string*, *position* <, *length*>)

Required Arguments

string
specifies a character constant, variable, or expression.

position
specifies a numeric constant, variable, or expression that is the beginning character position.

Optional Argument

length
specifies a numeric constant, variable, or expression that is the length of the substring to extract.

Interaction: If *length* is zero, a negative value, or larger than the length of the expression that remains in *string* after *position*, SAS extracts the remainder of the expression. SAS also sets _ERROR_ to 1 and prints a note to the log indicating that the *length* argument is invalid.

Tip: If you omit *length*, SAS extracts the remainder of the expression.

Details

In a DATA step, if the SUBSTR (right of =) function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The SUBSTR function returns a portion of an expression that you specify in *string*. The portion begins with the character that you specify by *position*, and is the number of characters that you specify in *length*.

The SUBSTR function and INDEX functions return a character position when using a varchar column. The SUBSTR function and INDEX functions return a byte position when using a character column.

Example

```
data
new;


date='06MAY98';

   month=substr(date, 3,
3);

   year=substr(date, 6,
2);

   put month=
year=;

run;
```

These statements produce these results:

```
month=MAY year=98
```

See Also

Functions:

- “SUBPAD Function” on page 1418
- “SUBSTR (left of =) Function” on page 1420
- “SUBSTRN Function” on page 1424

SUBSTRN Function

Returns a substring, allowing a result with a length of zero.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Tip: KSUBSTR has the same functionality.

Syntax

SUBSTRN(*string*, *position* <, *length*>)

Required Arguments

string
specifies a character or numeric constant, variable, or expression.

If *string* is numeric, it is converted to a character value that uses the BEST32. format. Leading and trailing blanks are removed, and no message is sent to the SAS log.

position
is an integer that specifies the position of the first character in the substring.

Optional Argument

length
is an integer that specifies the length of the substring. If you do not specify *length*, the SUBSTRN function returns the substring that extends from the position that you specify to the end of the string.

Details

Length of Returned Variable

In a DATA step, if the SUBSTRN function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The Basics

The following information applies to the SUBSTRN function:

- The SUBSTRN function returns a string with a length of zero if either *position* or *length* has a missing value.
- If the position that you specify is nonpositive, the result is truncated at the beginning so that the first character of the result is the first character of the string. The length of the result is reduced accordingly.
- If the length that you specify extends beyond the end of the string, the result is truncated at the end so that the last character of the result is the last character of the string.

Using the SUBSTRN Function in a Macro

If you call SUBSTRN by using the %SYSFUNC macro, then the macro processor resolves the first argument (*string*) to determine whether the argument is character or numeric. If you do not want the first argument to be evaluated as a macro expression, use one of the macro-quoting functions in the first argument.

Comparisons

The following table lists comparisons between the SUBSTRN and SUBSTR functions:

Condition	Function	Result
the value of *position* is nonpositive	SUBSTRN	returns a result beginning at the first character of the string.
the value of *position* is nonpositive	SUBSTR	■ writes a note to the log stating that the second argument is invalid. ■ sets _ERROR_ =1. ■ returns the substring that extends from the position that you specified to the end of the string.
the value of *length* is nonpositive	SUBSTRN	returns a result with a length of zero.
the value of *length* is nonpositive	SUBSTR	■ writes a note to the log stating that the third argument is invalid. ■ sets _ERROR_ =1. ■ returns the substring that extends from the position that you specified to the end of the string.

Condition	Function	Result
the substring that you specify extends past the end of the string	SUBSTRN	truncates the result.
the substring that you specify extends past the end of the string	SUBSTR	■ writes a note to the log stating that the third argument is invalid. ■ sets _ERROR_=1. ■ returns the substring that extends from the position that you specified to the end of the string.

Examples

Example 1: Manipulating Strings with the SUBSTRN Function

This example shows how to manipulate strings with the SUBSTRN function.

```
data test;
   retain string "abcd";
   drop string;
   do Position=-1 to 6;
      do Length=max(-1,-position) to 7-position;
         Result=substrn(string, position, length);
         output;
      end;
   end;
   datalines;
abcd
;
proc print noobs data=test;
run;
```

Output 3.76 *Partial Output from the SUBSTRN Function*

The SAS System

Position	Length	Result
-1	1	
-1	2	
-1	3	a
-1	4	ab
-1	5	abc
-1	6	abcd
-1	7	abcd
-1	8	abcd
0	0	
0	1	
0	2	a
0	3	ab
0	4	abc
0	5	abcd
0	6	abcd
0	7	abcd
1	-1	
1	0	
1	1	a
1	2	ab
1	3	abc
1	4	abcd
1	5	abcd
1	6	abcd
2	-1	
2	0	
2	1	b
2	2	bc
2	3	bcd
2	4	bcd
2	5	bcd

Example 2: Comparison between the SUBSTR and SUBSTRN Functions

This example compares the results of using the SUBSTR function and the SUBSTRN function when the first argument is numeric.

```
data _null_;
     substr_result="*" || substr(1234.5678,2,6) || "*";
     put substr_result=;
     substrn_result="*" || substrn(1234.5678,2,6)  || "*";
     put substrn_result=;
run;
```

These statements produce these results:

```
substr_result=*  1234*
substrn_result=*234.56*
```

See Also

Functions:

- "SUBPAD Function" on page 1418
- "SUBSTR (left of =) Function" on page 1420
- "SUBSTR (right of =) Function" on page 1422

SUM Function

Returns the sum of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

SUM(*argument-1* <, *argument-2, ..., argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. If all the arguments have missing values, then one of these actions occurs:

- If you use only one argument, the value of that argument is returned.
- If you use two or more arguments, a standard missing value (.) is returned.

Otherwise, the result is the sum of the nonmissing values. The argument list can consist of a variable list.

Example

```
data
one;

   input val1 val2 val3
val4;


datalines;

4 9 3
8

;



data
new;

   set
one;

   x1=sum(4, 9, 3,
8);


x2=sum(val1,val2,val3,val4, .);


x3=sum(56);

   x4=sum(of val1-
val2);



   y1=34; y2=12; y3=74;
y4=39;

   result=sum(of val1-val4, of y1-
y4);

   x5=sum(of val1-val3,
5);

   x6=sum(val1-
val2);
```

```
   x7=sum(of
y:);

   put
_all_;

run;
```

These statements produce these results:

```
val1=4 val2=9 val3=3 val4=8 x1=24 x2=24 x3=56 x4=13 y1=34 y2=12 y3=74 y4=39
result=183 x5=21 x6=-5
x7=159 _ERROR_=0 _N_=1
```

SUMABS Function

Returns the sum of the absolute values of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

SUMABS(*value-1* <, *value-2* ..., *value-n*>)

Required Argument

value
specifies a numeric expression.

Details

If all arguments have missing values, then the result is a 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 example returns the sum of the absolute values of the nonmissing arguments.

```
data _null_;
   x=sumabs(1, ., -2, 0, 3, .q, -4);
   put x=;
```

```
run;
```

These statements produce this result:

```
x=10
```

Example 2: Calculating the Sum of Absolute Values When You Use a Variable List

The following example uses a variable list and returns the sum of the absolute value of the nonmissing arguments.

```
data _null_;
   x1=1;
   x2=3;
   x3=4;
   x4=3;
   x5=1;
   x=sumabs(of x1-x5);
   put x=;
run;
```

These statements produce this result:

```
x=12
```

SYMEXIST Function

Returns an indication of the existence of a macro variable.

Category: Macro

See: "SYMEXIST Function" in *SAS Macro Language Reference for SAS Viya Workbench*

Syntax

SYMEXIST(*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name
- a character expression that constructs a macro variable name

Details

The SYMEXIST function searches any enclosing local symbol tables and then the global symbol table for the indicated macro variable and returns `1` if the macro variable is found or `0` if the macro variable is not found.

For more information, see the"SYMEXIST Function" in *SAS Macro Language Reference for SAS Viya Workbench*.

SYMGET Function

Returns the value of a macro variable during DATA step execution.

Category: Macro

Note: This function supports the VARCHAR type.

Syntax

SYMGET(*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name
- a character expression that constructs a macro variable name

Details

If the SYMGET 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.

The SYMGET function returns the value of a macro variable during DATA step execution. For more information, see the "SYMGET Function" in *SAS Macro Language Reference for SAS Viya Workbench*.

See Also

- *SAS Macro Language: Reference*

CALL Routines:

- "CALL SYMPUT Routine" on page 361

SYMGLOBL Function

Returns an indication of whether a macro variable is in global scope to the DATA step during DATA step execution.

Category: Macro

See: "SYMGLOBL Function" in *SAS Macro Language Reference for SAS Viya Workbench*

Syntax

SYMGLOBL(*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand.
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name.
- a character expression that constructs a macro variable name.

Details

The SYMGLOBL function searches only the global symbol table for the indicated macro variable and returns `1` if the macro variable is found or `0` if the macro variable is not found.

For more information, see "SYMGLOBL Function" in *SAS Macro Language Reference for SAS Viya Workbench*.

SYMLOCAL Function

Returns an indication of whether a macro variable is in local scope to the DATA step during DATA step execution.

Category: Macro

See: "SYMLOCAL Function" in *SAS Macro Language Reference for SAS Viya Workbench*

Syntax

SYMLOCAL(*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand.
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name.
- a character expression that constructs a macro variable name.

Details

The SYMLOCAL function searches the enclosing local symbol tables for the indicated macro variable and returns `1` if the macro variable is found or `0` if the macro variable is not found.

For more information, see "SYMLOCAL Function" in *SAS Macro Language Reference for SAS Viya Workbench*.

SYSEXIST Function

Returns a value that indicates whether an operating-environment variable exists in your environment.

Categories: SAS File I/O
Special

Syntax

SYSEXIST(*argument*)

Required Argument

argument
specifies a character variable that is the name of an operating-environment variable that you want to test.

Details

The SYSEXIST function searches for the existence of an operating-environment variable and returns 1 if the variable is found or 0 if the variable is not found.

Comparisons

The SYSEXIST function tests for the existence of an operating-environment variable. The SYSGET function retrieves the value of an operating-environment variable.

Example

The following example assumes that HOME is a valid operating-environment variable in your environment and that TEST is not valid. SYSEXIST tests both values.

```
data _null_;
   rc=sysexist("HOME");
   put rc=;
   rc=sysexist("TEST");
   put rc=;
run;
```

These statements produce these results:

```
rc=1
rc=0
```

If SYSEXIST returns a value of 1, then the variable that is being tested is an operating-environment variable. If SYSEXIST returns a value of 0, then the variable that is being tested is not an operating-environment variable in your environment.

See Also

Functions:

- "SYSGET Function" on page 1436

SYSGET Function

Returns the value of the specified operating-environment variable.

Category: Special

Syntax

Linux:

SYSGET('*environment-variable*')

Required Argument

environment-variable
is a character constant, variable, or expression with a value that is the name of an environment variable under Linux.

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 Linux:

```
here=sysget('HOME');
```

Since trailing blanks are treated as significant digits by the SYSGET function, environment variable names that are passed to these functions should be trimmed of trailing blanks. Use the TRIM function to remove trailing blanks. See "Example: Obtain Environment Variable Values under Linux" on page 1437 for an example of using the TRIM function with the SYSGET function to trim variable names.

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 Linux

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 Linux operating environment.

Example: Obtain Environment Variable Values under Linux

This example obtains the value of two environment variables in the Linux environment:

```
data _null_;
   length result $200;
   input env_var $;
   result=sysget(trim(env_var));
   put env_var= result=;
   datalines;
USER
PATH
;
```

Executing this DATA step for user ABCDEF displays these lines:

```
ENV_VAR=USER RESULT=abcdef
ENV_VAR=PATH RESULT=path-for-abcdef
```

See Also

Functions:

- “ENVLEN Function” on page 577

SYSMSG Function

Returns error or warning message text from processing the last data set or external file function.

Categories: SAS File I/O
External Files

Syntax

SYSMSG()

Details

SYSMSG returns the text of error messages or warning messages that are produced when the access function of a data set or external file encounters an error condition. If no error message is available, the returned value is blank. The internally stored error message is reset to blank after a call to SYSMSG. Subsequent calls to SYSMSG before another error condition occurs return blank values.

Example

This example uses SYSMSG to write to the SAS log the error message generated if FETCH cannot copy the next observation into the data set data vector. The return code is 0 only when a record is fetched successfully.

```
data
one;

   input
a;


stop;


datalines;

1

;



%macro
test;

   %let dsid=
%sysfunc(open(one));

   %let rc=
%sysfunc(fetch(&dsid));

   %if &rc ne 0 %then %put
%sysfunc(sysmsg());

   %let rc=
%sysfunc(close(&dsid));

%mend
test;
```

```
%test
```

These statements produce this result:

```
WARNING: End of file.
```

See Also

Functions:

- “FETCH Function” on page 600
- “SYSRC Function” on page 1445

SYSPARM Function

Returns the system parameter string.

Category: Special

Syntax

SYSPARM()

Details

If the SYSPARM 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.

SYSPARM enables you to access a character string specified with the SYSPARM= system option at SAS invocation or in an OPTIONS statement.

Note: If the SYSPARM= system option is not specified, the SYSPARM function returns a string with a length of zero.

Example

```
options
sysparm='yes';
```

```
data
one;

   if sysparm()='yes' then
do;

   put 'This
worked';


end;

run;
```

These statements produce this result:

```
This worked
```

See Also

System Options:

- "SYSPARM= System Option" in *SAS Macro Language Reference for SAS Viya Workbench*

SYSPROCESSID Function

Returns the process ID of the current process.

Category: Special

Syntax

SYSPROCESSID()

Details

The SYSPROCESSID function returns the 32-character hexadecimal ID of the current process. This ID can be passed to the SYSPROCESSNAME function to obtain the name of the current process.

Examples

Example 1: Using a DATA Step

The following DATA step writes the current process ID to the SAS log:

```
data _null_;
   id=sysprocessid();
   put id=;
run;
```

These statements produce this result:

```
id=41DB6788482C49BA4018000000000000
```

Example 2: Using SAS Macro Language

The following SAS Macro Language code writes the current process ID to the SAS log:

```
%let id=%sysfunc(sysprocessid());
%put &id;
```

These statements produce this result:

```
41DB6788482C49BA4018000000000000
```

See Also

Functions:

- "SYSPROCESSNAME Function" on page 1441

SYSPROCESSNAME Function

Returns the process name that is associated with a given process ID, or returns the name of the current process.

Category: Special

Syntax

SYSPROCESSNAME(<*process_id*>)

Required Argument

process_id
specifies a 32–character hexadecimal process ID.

Details

The SYSPROCESSNAME function returns the process name associated with the process ID that you supply as an argument. You can use the value returned from the SYSPROCESSID function as the argument to SYSPROCESSNAME. If you omit the argument, then SYSPROCESSNAME returns the name of the current process.

You can also use the values stored in the automatic macro variables SYSPROCESSID and SYSSTARTID as arguments to SYSPROCESSNAME.

Examples

Example 1: Using SYSPROCESSNAME without an Argument in a DATA Step

The following DATA step writes the current process name to the SAS log:

```
data _null_;
   name=sysprocessname();
   put name=;
run;
```

These statements produce this result:

```
name=DMS Process
```

Example 2: Using SYSPROCESSNAME with an Argument in SAS Macro Language

The following SAS Macro Language code writes the process name associated with the given process ID to the SAS log:

```
%let id=&sysprocessid;
%let name=%sysfunc(sysprocessname(&id));
%put &name;
```

These statements produce this result:

```
DMS Process
```

See Also

Functions:

- "SYSPROCESSID Function" on page 1440

SYSPROD Function

Determines whether a product is licensed.

Category: Special

Syntax

SYSPROD(*product-name*)

Required Argument

product-name
specifies a character constant, variable, or expression with a value that is the name of a SAS product.

Requirement *Product-name* must be the correct official name of the product or solution.

Details

The SYSPROD function returns 1 if a specific SAS software product is licensed, 0 if the SAS software product is not licensed for your system, and –1 if the product name is not recognized. If SYSPROD indicates that a product is licensed, it means that the final license expiration date has not passed.

It is possible for a SAS software product to exist on your system even though the product is no longer licensed. In this case, SAS cannot access this product. Similarly, it is possible for a product to be licensed but not installed.

Use SYSPROD in the DATA step, in an IML step, or in an SCL program.

You can list all of the licensed products on your system with this code:

```
proc setinit;
run;
```

Example

```
data
one;




/* If SAS/GRAPH software is currently licensed, then SYSPROD
returns a value of
1.

If SAS/GRAPH software is not currently licensed, then SYSPROD
returns a value of 0.
*/


a=sysprod('graph');




/* SYSPROD returns a value of -1 because ABC is not a valid
product name.
*/


b=sysprod('abc');




/* SYSPROD always returns a value of 1 because the Base product
must be
licensed

for the SYSPROD function to run successfully.
*/


c=sysprod('base');


   d=sysprod('base
sas');




   put a=;
   put b=;
   put c=;
```

```
   put
d=;




run;
```

These statements produce these results:

```
a=1
b=-1
c=1
d=1
```

SYSRC Function

Returns a system error number.

Categories: SAS File I/O
External Files

Syntax

SYSRC()

Details

SYSRC returns the error number for the last system error encountered by a call to one of the data set functions or external file functions.

Example

This example determines the error message if FILEREF does not exist:

```
%macro
test(fref);

   %if %sysfunc(fileref(&fref)) ne 0
%then
```

```
   %put %sysfunc(sysrc()) -
%sysfunc(sysmsg());

%mend
test;



%test(myfile)
```

See Also

Functions:

- “FILEREF Function” on page 616
- “SYSMSG Function” on page 1437

SYSTEM Function

Issues an operating environment command during a SAS session, and returns the system return code.

Category: Special

Restriction: This function is not supported in a DATA step that runs in CAS.

Syntax

SYSTEM(*command*)

Required Argument

command
specifies any of the following: a system command that is enclosed in quotation marks (explicit character string), an expression whose value is a system command, or the name of a character variable whose value is a system command that is executed.

Restriction The length of the command cannot be greater than 1,024 characters, including trailing blanks.

Example: General Example

This example demonstrates the functionality for the SYSTEM function.

```
data null;

   rc = system('echo Hello, World');

   put rc=;

run;
```

See Also

CALL Routines:

- "CALL SYSTEM Routine" on page 369

TAN Function

Returns the tangent.

Category: Trigonometric

Syntax

TAN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression and is expressed in radians. If the magnitude of *argument* is so great that `mod(argument,pi)` is accurate to less than about three decimal places, TAN returns a missing value.

Restriction The value cannot be an odd multiple of π /2.

Example

```
data
one;

   input
val;


datalines;
```

```
0.5

0

3.14159

;



data
new;

   set
one;


x=tan(val);

   y=tan(val/
3);

   put x=
y=;

run;
```

These statements produce these results:

```
x=0.5463024898 y=0.1682272183
x=0 y=0
x=-2.65359E-6 y=1.7320472695
```

TANH Function

Returns the hyperbolic tangent.

Category: Hyperbolic

Syntax

TANH(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The TANH function returns the hyperbolic tangent of the argument, which is given by this equation

$$\frac{\left(\varepsilon^{argument} - \varepsilon^{-argument}\right)}{\left(\varepsilon^{argument} + \varepsilon^{-argument}\right)}$$

Example

```
data
one;

   input
val;


cards;

0

0.5

-0.5

;



data
new;

   set
one;


x=tanh(val);

   put
x=;

run;
```

These statements produce these results:

```
x=0
x=0.4621171573
x=-0.462117157
```

TIME Function

Returns the current time of day as a numeric SAS time value.

Category: Date and Time

Interaction: If the value of the TIMEZONE= system option is set to a time zone name or time zone ID, the date and time values that are returned for this function are determined by the time zone.

Syntax

TIME()

Examples

Example 1: Displaying the Current Time

SAS assigns CURRENT a SAS time value that corresponds to 14:32:00 if the following statements are executed exactly at 2:32 PM:

```
data new;
   current=time();
   put current=time.;
run;
```

These statements produce this result:

```
current=10:58:26
```

Example 2: Determining a Time Value for a North America/Los_Angeles Time Zone

This example shows how the TIME function returns a value based on the value of the TIMEZONE= system option.

```
option timezone='America/Los_Angeles';
data _null_;
   t1=time();
   put t1=nltimap15.;
run;
```

These statements produce this result:

```
t1=11:10:24 AM
```

Example 3: Determining a Time Value for a Europe/London Time Zone

This example shows how the TIME function returns a value based on the value of the TIMEZONE= system option.

```
option timezone='Europe/London';
data _null_;
   t2=time();
   put t2=nltimap15.;
run;
```

These statements produce this result:

```
t2=07:11:08 PM
```

See Also

Functions:

- “DATE Function” on page 516
- “DATETIME Function” on page 520
- “TODAY Function” on page 1457

TIMEPART Function

Extracts a time value from a SAS datetime value.

Category: Date and Time

Syntax

TIMEPART(*datetime*)

Required Argument

datetime
is a numeric constant, variable, or expression that represents a SAS datetime value.

Example

SAS assigns TIME a SAS value that corresponds to 10:40:17 if the following statements are executed exactly at 10:40:17 a.m. on any date:

```
data new;
   datim=datetime();
   time=timepart(datim);
   put time=;
run;
```

These statements produce this result:

```
time=39297.181
```

TIMEVALUE Function

Returns the equivalent of a reference amount at a base date by using variable interest rates.

Category: Financial

Syntax

TIMEVALUE(*base-date*, *reference-date*, *reference-amount*, *compounding-interval*, *date-1*, *rate-1* <, *date-2*, *rate-2*, ..., *date-n*, *rate-n*>)

Required Arguments

base-date
is a SAS date. The value that is returned is the time value of *reference-amount* at *base-date*.

reference-date
is a SAS date. *Reference-date* is the date of *reference-amount*.

reference-amount
is numeric. *Reference-amount* is the amount at *reference-date*.

compounding-interval
is a SAS interval. *Compounding-interval* is the compounding interval. An example is *month*.

date
is a SAS date. Each date is paired with a rate. *Date* is the time at which *rate* takes effect.

rate
is a numeric percentage. Each rate is paired with a date. *Rate* is the interest rate that starts on *date*.

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

```
data
new;

 /*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:*/

amount_base1=TIMEVALUE("01jan2001"d, "01jan2000"d, 1000, "MONTH",
"01jan2000"d, 10);

/*If the interest rate jumps to 20% halfway through the year, the
resulting
calculation would be as follows: */

amount_base2=TIMEVALUE("01jan2001"d, "01jan2000"d, 1000, "MONTH",
"01jan2000"d, 10, "01jul2000"d, 20);

/*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:*/
amount_base3=TIMEVALUE("01jan2001"d, "01jan2000"d, 1000, "MONTH",
"01jul2000"d, 20, "01jan2000"d, 10);
   put amount_base1=
   put amount_base2=
put
amount_base3=;

run;
```

These statements produce these results:

```
amount_base1=1104.7130674
amount_base2=1160.6365778
amount_base3=1160.6365778
```

TINV Function

Returns a quantile from the *t* distribution.

Category: Quantile

Syntax

TINV(*p*, *df* <, *nc*>)

Required Arguments

p
is a numeric probability.

Range $0 < p < 1$

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Optional Argument

nc
is an optional numeric noncentrality parameter.

Details

The TINV function returns the *p*th 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.

Note: TINV is the inverse of the PROBIT function.

Example

```
data _null_;
   x=tinv(.95,2);
   y=tinv(.95,2.5,3);
   put x=;
   put y=;
run;
```

These statements produce these results:

```
x=2.9199855804
y=11.033833625
```

See Also

Functions:

- "QUANTILE Function" on page 1293

TNONCT Function

Returns the value of the noncentrality parameter from the Student's *t* distribution.

Category: Mathematical

Syntax

TNONCT(*x*, *df*, *prob*)

Required Arguments

x
is a numeric random variable.

df
is a numeric degrees of freedom parameter.

Range $df > 0$

prob
is a probability.

Range $0 < prob < 1$

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 this 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 example computes the noncentrality parameter from the *t* distribution.

```
data work;
   x=2;
   df=4;
   do nc=1 to 3 by .5;
      prob=probt(x, df, nc);
      ncc=tnonct(x, df, prob);
      output;
   end;
run;
proc print;
run;
```

Output 3.77 *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

TODAY Function

Returns the current date as a numeric SAS date value.

Category: Date and Time

Alias: DATE

Interaction: If the value of the TIMEZONE= system option is set to a time zone name or time zone ID, the date and time values that are returned for this function are determined by the time zone.

Syntax

TODAY()

Details

The TODAY function produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

Examples

Example 1: Using the TODAY Function in an Accounting Example

These statements illustrate a practical use of the TODAY function:

```
data _null_;
   datedue=35;
   tday=today();
   if (tday-datedue)> 15 then
      do;
      put 'As of ' tday date9. ' Account #'
      account 'is more than 15 days overdue.';
   end;
run;
```

These statements produce this result:

```
As of 02MAR2021 Account #. is more than 15 days overdue.
```

Example 2: Determining the Current Date for a North America/Denver Time Zone

This example shows how the TODAY function returns a value based on the value of the TIMEZONE= system option.

```
option timezone='America/Denver';
data _null_;
   d1=today();
   put d1=nldate.;
run;
```

These statements produce this result:

```
d1=March 02, 2021
```

Example 3: Determining the Current Date for an Asia/Seoul Time Zone

This example shows how the TODAY function returns a value based on the value of the TIMEZONE= system option.

```
option timezone='Asia/Seoul';
data _null_;
   d2=today();
   put d2=nldate.;
run;
```

These statements produce this result:

```
d2=March 02, 2021
```

See Also

Functions:

- “DATE Function” on page 516
- “DATETIME Function” on page 520
- “TIME Function” on page 1450

TRANSLATE Function

Replaces specific characters in a character expression.

Category: Character

Restriction: This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: The DBCS equivalent function is KTRANSLATE.

Syntax

TRANSLATE(*source*, *to-1*, *from-1* <, ...*to-n*, *from-n*>)

Required Arguments

source
specifies a character constant, variable, or expression that contains the original character string.

to
specifies the characters that you want TRANSLATE to use as substitutes.

from
specifies the characters that you want TRANSLATE to replace.

Details

General Information

Values of *to* and *from* correspond on a character-by-character basis. TRANSLATE changes the first character of *from* to the first character of *to*, and so on.

If *to* has fewer characters than *from*, TRANSLATE changes the extra *from* characters to blanks. If *to* has more characters than from, TRANSLATE ignores the extra *to* characters. See "Example 3:".

In a DATA step, if the TRANSLATE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

There is no functional difference between using several pairs of short arguments or fewer pairs of longer arguments.

Comparisons

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).

Note: The order of arguments is different between the TRANWRD and TRANSLATE functions. In TRNWRD, the second argument is target and the third argument is replacement. In TRANSLATE, the second argument is *to* and the third argument is *from*. The character string that is specified by target is similar to the

from argument in TRANSLATE. The character string that is specified in replacement is similar to the *to* argument in TRANSLATE.

Examples

Example 1

```
data new;
   x=translate('XYZW', 'AB', 'VW');        /
*1*/

   string1='AABBAABABB';                   /
*2*/

   y=translate(string1,'12','AB');         /
*3*/

   put x=;                                 /*4*/
   put y=;                                 /
*5*/

run;
```

1 Replace the letters VW with AB in the source character string, XYZW and assign the value to X.

2 Assign the character string, AABBAABABB to string1.

3 Replace the letters AB with 12 in the character string AABBAABABB and assign the value to Y.

4 Write the value of X to the log.

5 Write the value of Y to the log.

These statements produce these results:

```
x=XYZB
y=1122112122
```

Example 2

This example shows the process of attempting to translate repeated characters.

```
data new;
   x=TRANSLATE('1111111111','23','11');    /*1*/
   put x=;    /*2*/
run;
```

1 Replace all occurrences of the numbers 11 with 23 and assigns the value to X.

There are two conflicted rules when using the TRANSLATE function: rule one is replace '1' with '2; rule two is replace '1' with '3'. TRANSLATE accepts only the first rule and ignores the second rule. So, '1' is replaced with '2'. The result is '2222222222'. This rule applies to Viya 3.5 and SAS 9.4M7. Prior to these versions, the output would be '3333333333'.

2 Write the value of X to the log.

```
x=2222222222
```

Example 3

This example demonstrates the functionality if *to* or *from* have fewer characters.

```
data new;
   x=translate('XYZW', 'A', 'VZ');      1
   put x=;
   y=translate('XYZW', 'AB', 'Y');      2
run;
```

1 The *to* argument has fewer characters than the *from* argument, so V corresponds to A. Z does not have a corresponding letter, so it is replaced with a blank.

2 The *from* argument has fewer characters than the *to* argument, so Y is replaced with A. There is no corresponding letter for B, so it is ignored.

```
x=XY W
y=XAZW
```

See Also

Functions:

- "TRANWRD Function" on page 1465

TRANSTRN Function

Replaces or removes all occurrences of a substring in a character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

TRANSTRN(*source*, *target*, *replacement*)

Required Arguments

source
specifies a character constant, variable, or expression that you want to translate.

target
specifies a character constant, variable, or expression that is searched for in *source*.

Requirement The length for *target* must be greater than zero.

replacement
specifies a character constant, variable, or expression that replaces *target*.

Details

Length of Returned Variable

In a DATA step, if the TRANSTRN function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes. You can use the LENGTH statement, before calling TRANSTRN, to change the length of the value.

The Basics

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* string and the *replacement* string. To remove all occurrences of *target*, specify *replacement* 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. However, in performing this scan, 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

The following statements and values produce these results:

```
data
one;

   input name
$20.;


datalines;

Mrs. Jane
Doe

Miss Joan
Smith

;



data
new;

   set
one;

   name=transtrn(name, "Mrs.",
"Ms.");

   name=transtrn(name, "Miss",
"Ms.");

   put
name=;

run;
```

These statements produce these results:

```
name=Ms. Jane Doe
name=Ms. Joan Smith
```

Example 2: Removing Blanks from the Search String

In this example, the TRANSTRN function does not replace the source string because the target string contains blanks.

```
data list;
   input salelist $;
   length target $10 replacement $3;
   target='FISH';
   replacement='NIP';
   salelist=transtrn(salelist, target, replacement);
   put salelist;
   datalines;
CATFISH
;
```

The LENGTH statement pads *target* with blanks to the length of 10, which causes the TRANSTRN function to search for the character string `'FISH '` in SALELIST. 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*:

```
salelist=transtrn(salelist,trim(target), replacement);
put salelist;
```

Now, this line is written to the SAS log:

```
CATNIP
```

Example 3: Zero Length in the Third Argument of the TRANSTRN Function

This example shows the results of the TRANSTRN function when the third argument, *replacement*, has a length of zero. In the DATA step, a character constant that consists of two quotation marks represents a single blank and not a zero-length string. The results for *string1* are different from the results for *string2*.

```
data _null_;
   string1='*' || transtrn('abcxabc', 'abc', trimn('')) || '*';
   put string1=;
   string2='*' || transtrn('abcxabc', 'abc',  '') || '*';
  put string2=;
run;
```

These statements produce these results:

```
string1=*x*
string2=* x *
```

See Also

Functions:

- “TRANSLATE Function” on page 1458

TRANWRD Function

Replaces all occurrences of a substring in a character string.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

TRANWRD(*source*, *target*, *replacement*)

Required Arguments

source
specifies a character constant, variable, or expression that you want to translate.

target
specifies a character constant, variable, or expression that is searched for in *source*.

Requirement The length for *target* must be greater than zero.

replacement
specifies a character constant, variable, or expression that replaces *target*. When the replacement string has a length of zero, TRANWRD uses a single blank instead.

Details

Length of Returned Variable

In a DATA step, if the TRANWRD function returns a value to a variable that has not previously been assigned a length, that variable is given a length of 200 bytes. You can use the LENGTH statement, before calling TRANWRD, to change the length of the value.

The Basics

The TRANWRD function copies the value in *source* to the result string while searching for all non-overlapping substrings in *source* that are equal to the value in

target. Each of these substrings is omitted from the result and the value of *replacement* is copied in its place. The TRANWRD function does not remove trailing blanks in the *target* string or the *replacement* 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. However, in performing this scan, 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 TRANWRD scans for substrings and replaces those substrings with a second substring.

Note: The order of arguments is different between the TRANWRD and TRANSLATE functions. In TRANWRD, the second argument is *target* and the third argument is *replacement*. In TRANSLATE, the second argument is *to* and the third argument is *from*. The character string specified by *target* is similar to the *from* argument in TRANSLATE. The character string specified by *replacement* is similar to the *to* argument in TRANSLATE.

Examples

Example 1: Replacing All Occurrences of a Word

The following statements and values produce these results:

```
data
one;

   input name
$20.;


datalines;

Mrs. Jane
Doe

Miss Joan
Smith
```

```
;



data
new;

   set
one;

    name=tranwrd(name, "Mrs.",
"Ms.");

    name=tranwrd(name, "Miss",
"Ms.");

    put
name=;

run;
```

These statements produce these results:

```
name=Ms. Jane Doe
name=Ms. Joan Smith
```

Example 2: Removing Blanks from the Search String

In this example, the TRANWRD function does not replace the source string because the target string contains blanks.

```
data list;
   input salelist $;
   length target $10 replacement $3;
   target='FISH';
   replacement='NIP';
   salelist=tranwrd(salelist, target, replacement);
   put salelist;
   datalines;
CATFISH
;
```

The LENGTH statement pads *target* with blanks to the length of 10, which causes the TRANWRD function to search for the character string `'FISH '` in SALELIST. 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*:

```
salelist=tranwrd(salelist,trim(target), replacement);
put salelist;
```

Now, this line is written to the SAS log:

```
CATNIP
```

Example 3: Zero Length in the Third Argument of the TRANWRD Function

This example shows the results of the TRANWRD function when the third argument, *replacement*, has a length of zero. In this case, TRANWRD uses a single blank. In the DATA step, a character constant that consists of two consecutive quotation marks represents a single blank and not a zero-length string. The results for *string1* and *string2* are the same.

```
data _null_;
   string1='*' || tranwrd('abcxabc', 'abc', trimn('')) || '*';
   put string1=;
   string2='*' || tranwrd('abcxabc', 'abc', '') || '*';
   put string2=;
run;
```

These statements produce these results:

```
string1=* x *
string2=* x *
```

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 example, the TRANWRD function is used twice: to replace three commas with one comma and to replace the ending two commas with a period:

```
data _null_;
   mytxt='If you exercise your power to vote,,,then your opinion will
be heard,,';
   newtext=tranwrd(mytxt, ',,,', ',');
   newtext2=tranwrd(newtext, ',,' , '.');
   put // mytxt= / newtext= / newtext2=;
run;
```

These statements produce these results:

```
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 1458
- “TRANSTRN Function” on page 1461

TRIGAMMA Function

Returns the value of the trigamma function.

Category: Mathematical

Syntax

TRIGAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Restriction Nonpositive integers are invalid.

Details

The TRIGAMMA function returns the derivative of the DIGAMMA function. For *argument* > 0, the TRIGAMMA function is the second derivative of the LGAMMA function.

Example

```
data
new;


x=trigamma(3);

   put
x=;

run;
```

These statements produce this result:

```
x=0.3949340668
```

TRIM Function

Removes trailing blanks from a character string and returns one blank if the string is missing.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type. If the argument is blank, KTRIM returns whitespace with a length of 0 for the varchar type.

Tip: The DBCS equivalent function is "KTRIM Function" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

Syntax

TRIM(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the TRIM 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.

The Basics

TRIM copies a character argument, removes trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIM returns one blank. TRIM is useful for concatenating because concatenation does not remove trailing blanks.

Assigning the results of TRIM to a variable does not affect the length of the receiving variable. If the trimmed value is shorter than the length of the receiving variable, SAS pads the value with new blanks as it assigns the value to the variable.

Examples

Example 1: Removing Trailing Blanks

The following statements and this data line produce these results:

```
data
test;

   input part1 $ 1-10 part2 $
11-20;

   hasblank=part1||
part2;

   noblank=trim(part1)||
part2;

   put
hasblank=;

   put
noblank=;


datalines;

apple
sauce

;
```

These statements produce these results:

```
hasblank=apple     sauce
noblank=applesauce
```

Example 2: Concatenating a Blank Character Expression

```
data
new;

   x="A"||trim("
")||"B";

   z="
";

   y=">"||
trim(z)||"<";

   put x=
y=;

run;
```

These statements produce these results:

```
x=A B y=> <
```

See Also

Functions:

- “COMPRESS Function” on page 467
- “LEFT Function” on page 1048
- “RIGHT Function” on page 1346
- “TRIMN Function” on page 1472

TRIMN Function

Removes trailing blanks from character expressions and returns a string with a length of zero if the expression is missing.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

TRIMN(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the TRIMN 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.

Assigning the results of TRIMN to a variable does not affect the length of the receiving variable. If the trimmed value is shorter than the length of the receiving variable, SAS pads the value with new blanks as it assigns it to the variable.

The Basics

TRIMN copies a character argument, removes all trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIMN returns a string with a length of zero. TRIMN is useful for concatenating because concatenation does not remove trailing blanks.

Comparisons

The TRIMN and TRIM functions are similar. TRIMN returns a string with a length of zero for a blank string. TRIM returns one blank for a blank string.

Example

```
data
new;

   x="A"||
trimn("")||"B";

   z="
";

   y=">"||
trimn(z)||"<";

   put x=
y=;

run;
```

These statements produce these results:

```
x=AB
y=><
```

See Also

Functions:

- “COMPRESS Function” on page 467
- “LEFT Function” on page 1048
- “RIGHT Function” on page 1346

■ "TRIM Function" on page 1470

TRUNC Function

Truncates a numeric value to a specified number of bytes.

Category: Rounding and Truncation

Syntax

TRUNC(*number*, *length*)

Required Arguments

number
specifies a numeric constant, variable, or expression.

length
specifies an integer.

Details

The TRUNC function truncates a full-length *number* (stored as a double number) to a smaller number of bytes, as specified in *length*, 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

This example uses the TRUNC function.

```
data test;
  length x 3;
  x=1/5;
run;
data test2;
  set test;
  if x ne 1/5 then
     put 'x ne 1/5';
  if x eq trunc(1/5,3) then
     put 'x eq trunc(1/5,3)';
run;
```

These statements produce these results:

```
x ne 1/5
x eq trunc(1/5,3)
```

The variable X is stored with a length of 3. Therefore, each of the comparisons is true.

TSLVL Function

provides information about the specifics of the installation, including the release, hot fixes, and maintenance images.

Syntax

TSLVL(<*option-1*,–*option-n*>)

Optional Arguments

Here are the available options that you can request. Options are not case sensitive.

A
additional track information.

B
Script information is in this format:`<Script DVD>/script/<Script Name>.bld`

```
data _null_;
   x = tslvl('sasxkern', 'b');
   put x=;
run;
```

```
x=sup:sasxkern:/sas/dev/mva-vbviya,
```

C
changes the separator from a comma to a colon; also decodes the image description if the I option is specified.

```
data _null_;
   x = tslvl('sasxkern', 'cdpti');
   put x=;
run;
```

```
x=:V.04.00M0D12012020:cbase:dev/mva-vbviya:SAS supervisor kernal primary routines.
```

D
port date (for example, 9.03.01B0D02282010).

E
number of seconds since January 1, 1970 (UNIX date). SAS dates are based on the number of seconds since January 1, 1960.

H
hot fix information.

I
image description. Use with the Z option to decode the field.

M
maintenance release information.

N
suppresses error messages if the module is not found.

P
product number.

S
returns a colon-separated list that comprises the Script DVD, Script Name, and Script Location in that order.

Note: Use the Z option to decode the fields.

```
<Script DVD>:<Script Name>:<Script Location>
```

```
data _null_;
   x = tslvl('sasxkern', 'sz');
   put x=;
run;
```

```
x=sup:sasxkern:/sas/dev/mva-vbviya,
```

T
SAS track information.

Z
decode any fields that are encoded.

Details

The output for the TSLVL function returns one character string for each requested information item, separated by commas to facilitate parsing the results.

If any information for an image is not available, you might receive the following errors:

- ImageNotLoaded indicates that the requested image is not loaded.
- VersionNotFound indicates that version information could not be found.
- NotAvailable indicates other problems in obtaining version information.

Example

The following example calls the TSLVL function twice. The first instance results in the release number, SAS 9.4 Technical Support level 1, Maintenance level 5. This was the sixth release of SAS 9.4. The second instance requests the track, the port

date, and the UNIX time at which the image was built. This image was built in the main development track for SAS 9.4M5 on April 22, 2018.

```
data x;
   y=tslvl('sasxkern');
   put y=;
   a=tslvl('sasxkern','tde');
   put a=;
run;
```

```
y=9.04 TS1M5
a=dev/mva-v940m5b,9.04.01M5P04222018,1524439845
NOTE: The data set WORK.X has 1 observations and 2 variables.
```

TYPEOF Function

Returns a value that indicates whether the argument is character or numeric.

Category: Character

Restriction: The TYPEOF function is used exclusively with the Graph Template Language (GTL) and in WHERE clauses, but not in DATA steps.

See: "VTYPE Function"

Syntax

TYPEOF(*column*)

Required Argument

column
specifies a value that designates character or numeric.

Here are the valid values:

C
indicates that the argument is a character value.

N
indicates that the argument is a numeric value.

Example

This example specifies the TYPEOF function in a WHERE expression.

```
data y; y='abc'; run;

proc print data=y;
```

```
      where seechar(typeof,'typeof is')='C';

run;
```

TZONEID Function

Returns the current time zone ID.

Category: Date and Time

Alias: TZID

Restriction: This function is assigned an I18N Level 2 status and designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

TZONEID(<*time-zone-id*>)

Optional Argument

time-zone-id
specifies a region/area value that is defined by SAS. When you specify a zone ID, the time zone that SAS uses is determined by the time zone name and daylight savings time rules.

Details

The TZONEID function returns a blank value if the TIMEZONE= option is blank or a user-defined time zone is specified.

The TZONEID function validates the timezone ID. If you specify the timezone ID, the function returns the timezone ID if it is valid or returns a blank value if the ID is invalid.

Example

In the first example, the TIMEZONE option is set to JST. In the second example, TIMEZONE is set to a blank value. In the third example TIMEZONE is set to user-specified time zone. In the fourth example a valid timezoneid and an invalid timezoneid is displayed.

Statements	Results
options timezone=jst; data _null_ ; tzid=tzoneid() ; put tzid=; run;	tzid=ASIA/TOKYO
options timezone=''; data _null_ ; tzid=tzoneid() ; put tzid=; run;	tzid=
options timezone='xxx-12'; /* user defined timezone */ data _null_ ; tzid=tzoneid() ; put tzid=; run;	tzid=ETC/GMT-12

data null;
name_valid=tzoneid('asia/tokyo');
name_invalid=tzoneid('Milky Way');
put name_valid =;
put name_invalid=;
run;
	name_valid=ASIA/TOKYO
	name_invalid=

TZONENAME Function

Returns the current standard or daylight savings time, time zone name.

Category: Date and Time

Restriction: This function is assigned an I18N Level 2 status and designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

TZONENAME()

TZONENAME(<*time-zone-id*,*datetime*>)

Optional Arguments

time-zone-id
specifies a *region/area* value that is defined by SAS. When you specify a zone ID, the time zone that SAS uses is determined by time zone name and daylight savings time rules.

See For a list of the time zone IDs, see "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

datetime
specifies a SAS datetime value.

Details

The TZONENAME function returns a blank value if the TIMEZONE= option and the time-zone-id argument are blank.

The TZONENAME function returns the timezone name based on the specified timezone and datetime. If the SAS datetime is not specified, then the current date is used. If time-zone-id is not specified, then the timezone ID that is specified with the TIMEZONE= option is used.

Example

In the first example, the TIMEZONE option is set to a blank value. In the second example, TIMEZONE is set to timezone name, JST. In the third example, TIMEZONE is set to a user-specified time zone. In the fourth example, TIMEZONE is set to a time zone ID.

Statements	Results

`options tz='';`
`data _null_;`
`  tzname=tzonename() ;`
`  put tzname =;`
`run;`
	`tzname=`

`options tz='jst';`
`data _null_;`
`  tzname=tzonename() ;`
`  put tzname =;`
`run;`
	`tzname=JST`

`options tz='xxx-12';`
`data _null_;`
`  tzname=tzonename() ;`
`  put tzname =;`
`run;`
	`tzname=XXX`

Statements	Results

`options tz='America/Chicago';`
`data _null_;`
`  tzname=tzonename('01SEP2014:01:01:01'dt);`
`  put tzname =;`
`run;`
	`tzname=CDT`

TZONEOFF Function

Returns the user time zone offset.

Category: Date and Time

Restriction: This function is assigned an I18N Level 2 status and designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

TZONEOFF()

TZONEOFF(<*time-zone-id*,*datetime*>)

Optional Arguments

time-zone-id
specifies a *region/area* value that is defined by SAS. When you specify a time zone ID, the time zone that SAS uses is determined by time zone name and daylight savings time rules.

See For a list of time zone IDs, see "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*.

datetime
specifies a SAS datetime value.

Details

If no arguments are specified, the TZONEOFF function returns the time zone offset for the specified TIMEZONE option. The TZONEOFF (time-zone-id) function with the time zone ID argument returns the time zone offset for the specified time zone ID. The TZONEOFF function with the time zone ID name returns the time zone offset for the specified time zone name. We recommend that you use the time zone ID, since it is not locale dependent.

If SASDTM is not provided, TZONEOFF returns the current timezone offset. If SASDTM is provided, it returns the offset to get the local time for specified time value.

Example

The first example has no argument, so the TZONEOFF function returns an offset for the current SAS session. The second example returns an offset based on a specific time zone ID. The third example returns an offset based on a specific time zone ID and a specific date and time. The fourth example returns an offset based on the Time Zone option and a specific date.

If the SAS datetime is not specified, then the TZONEOFF function returns the current timezone offset. If the SAS datetime is specified, then the function returns the offset to provide the local time for the specified time value.

Statements	Results
`option TIMEZONE='AUSTRALIA/MELBOURNE';` `%PUT %SYSFUNC(TZONEOFF());`	`39600`
`option TIMEZONE='AUSTRALIA/MELBOURNE';` `%PUT %SYSFUNC(TZONEOFF(EUROPE/ROME));`	`3600`

`data _null_ ;`
`  dt1='05DEC2012:08:17:52'dt  ;`
`  dt2='05JUN2012:08:17:52'dt  ;`
`  offset1= TZONEOFF('EUROPE/MOSCOW', dt1) ;`
`  offset2= TZONEOFF('EUROPE/MOSCOW', dt2) ;`
`put offset1= / offset2= ;`
`run ;`
	`offset1=14400`
	`offset2=14400`

`option TIMEZONE='EUROPE/MOSCOW' ;`
`data _null_ ;`
`  dt1='05DEC2012:08:17:52'dt  ;`
`  dt2='05JUN2012:08:17:52'dt  ;`
`  offset1= TZONEOFF(dt1) ;`
`  offset2= TZONEOFF(dt2) ;`
`put offset1= / offset2= ;`
`run ;`
	`offset1=14400`
	`offset2=14400`

TZONES2U Function

Converts a SAS date time value to a UTC date time value.

Category: Date and Time

Restriction: This function is assigned an I18N Level 2 status and designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

TZONES2U(*datetime*,<*time-zone-id*>)

Required Argument

datetime
specifies a SAS datetime value.

Optional Argument

time-zone-id
specifies a region/area value that is defined by SAS. When you specify a time zone ID, the time zone that SAS uses is determined by time zone name and daylight savings time rules.

See For a list of time zone IDs, see "Time Zone IDs and Time Zone Names" in *SAS National Language Support (NLS) Reference for SAS Viya Workbench*

Details

The TZONES2U() function returns UTC-based time for the specified TIMEZONE. The TZONES2U(time-zone-id) function with the time zone ID argument returns UTC-based time for the specified time zone ID. If the time zone name is not valid for the current locale, you receive an error. If time-zone-id is not specified, then the timezone ID that is specified with the TIMEZONE= option is used.

Example

The following example converts a SAS date time into UTC time.

Statements	Results

`option locale=ja_JP TZ='JST' ;`
`data _null_ ;`
`  dt='05Nov2012:08:17:52'dt  ;`
`  utc1 = tzones2u(dt) ;`
`  utc2 = tzones2u(dt,'ASIA/TOKYO') ;`
`  utc3 = tzones2u(dt,'JST') ;`
`  put dt= /utc1= is8601dz. //;`
`  put dt= /utc2= is8601dz. // ;`
`  put dt= /utc3= is8601dz. // ;`
	`dt=1667722672`
	`utc1=2012-11-04T23:17:52+00:00`
	
	`dt=1667722672`
	`utc2=2012-11-04T23:17:52+00:00`
	
	`dt=1667722672`
	`utc3=2012-11-04T23:17:52+00:00`

Statements	Results
`run ;`	

TZONEU2S Function

Converts a UTC date time value to a SAS date time value.

Category: Date and Time

Restriction: This function is assigned an I18N Level 2 status and designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

TZONEU2S(*UTC-date-time value*<, *time-zone-id*>)

Required Argument

UTC-date time value
specifies a Coordinated Universal Time (UTC) datetime value

Optional Argument

time-zone-id
specifies a region/area value that is defined by SAS. When you specify a zone ID, the time zone that SAS uses is determined by the time zone name and daylight savings time rules.

Details

The TZONEU2S(datetime) function returns the SAS datetime for a UTC time for the specified TIMEZONE option. The TZONEU2S(datetime, time-zone-id) function with the time zone ID argument, returns the SAS datetime for the UTC time for the specified time zone ID.

Example

The following example converts a UTC date time to three specific SAS date time values.

Statements	Results

`option locale=fr_FR TZ='AMERICA/DENVER';`
`data _null_;`
`  utc_date = '2012-09-02T02:34:56+00:00';`
`  udt = input(utc_date,is8601dz.);`
`  sdt1 = tzoneu2s(udt);`
`  sdt2 = tzoneu2s(udt,'EUROPE/AMSTERDAM');`
`  sdt3 = tzoneu2s(udt,'CET');`
`put sdt1= datetime. / sdt2= datetime. /`
`sdt3= datetime.;`
`run;`
	`sdt1=01SEP12:20:34:56`
	`sdt2=02SEP12:04:34:56`
	`sdt3=02SEP12:03:34:56`

UNIFORM Function

Returns a random variate from a uniform distribution.

Category: Random Number

Alias: RANUNI

Restriction: *This function is deprecated.* The function is suitable for small samples and for applications that do not require a sophisticated random-number generator. It is not suitable for parallel and distributed processing. For more demanding applications, use the STREAMINIT subroutine and the RAND(NORMAL) function.

See: "RANUNI Function" on page 1337 "RAND Function" on page 1304 "CALL STREAMINIT Routine" on page 352

UPCASE Function

Converts all lowercase single-width English alphabet letters in an argument to uppercase.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Syntax

UPCASE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the UPCASE 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.

The UPCASE function copies a character argument, converts all lowercase single-width English alphabet letters to uppercase letters, and returns the altered value as a result.

The results of the UPCASE function depend directly on the translation table that is in effect (see TRANTAB system option) and indirectly on the ENCODING system option and the LOCALE system option.

Example

```
data
one;

   input names
$13.;


datalines;

SaraH

Mylinda

Ryan

MaRk T.
Smith

;



data
new;

   set
one;
```

```
name=upcase(names);

   put
name=;

run;
```

These statements produce these results:

```
name=SARAH
name=MYLINDA
name=RYAN
name=MARK T. SMITH
```

See Also

Functions:

- “LOWCASE Function” on page 1089
- “PROPCASE Function” on page 1259

URLDECODE Function

Returns a string that was decoded using the URL escape syntax.

Category: Web Tools

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Syntax

URLDECODE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable in a DATA Step

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.

The URLDECODE and URLENCODE functions do not verify that the bytes produced by the escape sequences are valid characters based on the encoding.

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 symbols:

- 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

Argument can be decoded using either SAS session encoding or UTF-8 encoding. To decode *argument* by using the SAS session encoding, set the system option URLENCODING=SESSION. To decode *argument* by using UTF-8 encoding, set the system option URLENCODING=UTF8.

Example

```
data
new;

   x1=urldecode('abc
+def');

   x2=urldecode('why
%3F');


x3=urldecode('%41%42%43%23%31');

   put x1= x2=
x3=;

run;
```

These statements produce these results:

```
x1=abc def
x2=why?
x3=ABC#1
```

See Also

Functions:

- "URLENCODE Function" on page 1489

System Options:

- "URLENCODING= System Option" in *SAS System Options for SAS Viya Workbench*

URLENCODE Function

Returns a string that was encoded using the URL escape syntax.

Category: Web Tools

Restriction: This function is assigned an I18N Level 1 status. If possible, avoid I18N Level 1 functions if you are using a non-English language. Under certain circumstances, the I18N Level 1 functions might not work correctly with Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) encodings. For more information, see Internationalization Compatibility.

Syntax

URLENCODE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable in a DATA Step

If the URLENCODE 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.

The URLDECODE and URLENCODE functions do not verify that the bytes produced by the escape sequences are valid characters based on the encoding.

The Basics

Argument can be encoded using either SAS session encoding or UTF-8 encoding. To encode *argument* by using the SAS session encoding, set the system option

URLENCODING=SESSION. To encode *argument* by using UTF-8 encoding, set the system option URLENCODING=UTF8.

The URLENCODE function encodes characters that might otherwise be significant when used in a URL. This function encodes all characters except for these characters and symbols:

- 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.

Example

```
data new;
   x1=urlencode('abc def');
   put x1=;
run;
```

These statements produce this result:

```
x1=abc%20def
```

See Also

Functions:

- "URLDECODE Function" on page 1487

System Options:

- "URLENCODING= System Option" in *SAS System Options for SAS Viya Workbench*

USS Function

Returns the uncorrected sum of squares of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

USS(*argument-1* <, ...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one nonmissing argument is required. Otherwise, the function returns a missing value. If you have more than one argument, the argument list can consist of a variable list, which is preceded by OF.

Example

```
data
one;

   input val1 val2 val3
val4;


datalines;

4 2 3.5
6

;



data
new;

   set
one;

   x1=uss(of val1-
val4);
```

```
   x2=uss(of val1-
val4, .);

   x3=uss(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These statements produce this result:

```
x1=68.25 x2=68.25 x3=9316.125
```

UUIDGEN Function

Returns a Universally Unique Identifier (UUID) as a string of 36 hexidecimal characters and hyphens or a binary value of 16 bytes.

Syntax

UUIDGEN(<*maximum-warnings* <, *binary-result*>>)

Optional Arguments

maximum-warnings
specifies an integer value that represents the maximum number of warnings that this function writes to the log.

Default 1

Note This option limits the number of error messages in MVA without multithreading. If there are multiple threads, the limit might apply to the number of messages in each thread or to the total number of messages for all threads.

binary-result
specifies an integer value that indicates whether this function should return a binary result. Nonzero indicates a binary result should be returned. Zero indicates that a character result should be returned.

Default 0

Note Use a $HEX32 format when writing a binary UUID to avoid garbling or truncating the output.

Details

Length of Returned Variable in a DATA Step

If the UUIDGEN 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.

The Basics

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
```

A character UUID contains 36 characters. If a character result is assigned to a character variable that is not VARCHAR and has a length exceeding 36 characters, the result is padded with blanks. If a character result is assigned to a character variable with a maximum length less than 36 characters, the result is truncated and is not a valid UUID. Truncation does not cause an error message.

A binary UUID contains 16 bytes. If a binary result is assigned to a character variable that is not VARCHAR and has a length exceeding 16 bytes, the result is padded with bytes having a numeric value of zero. If a character result is assigned to a character variable with a maximum length less than 16 bytes, the result is truncated and is not a valid UUID. Truncation does not cause an error message.

Example

The first example returns a UUID as a string of hexidecimal characters. The second example returns a UUI as a binary string.

```
data _null_;
   length hex $36;
   hex = uuidgen();
   put hex=;
run;
```

```
hex=8343fd83-e635-438d-9277-550a5b8460f3
```

```
data _null_;
   length bin $16;
   bin = uuidgen(1,1);
   put bin= $hex32.;
run;
```

```
bin=611B02633A4C9D4895B5714267F76B62
```

VAR Function

Returns the variance of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

VAR(*argument-1*, *argument-2* <, ...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Example

```
data
one;

   input val1 val2 val3
val4;


datalines;

4 2 3.5
6

;



data
new;

   set
one;

   array list(*) val1-
val4;
```

```
   x1=var(of val1-
val4);

   x2=var(val1,
val4, .);

   x3=var(of x1-
x2);

   put x1= x2=
x3=;

run;
```

These statements produce this result:

```
x1=2.7291666667 x2=2 x3=0.2658420139
```

VARFMT Function

Returns the format that is assigned to a SAS data set variable.

Category: SAS File I/O

See: "Using the VAR and V Functions" for information about using the VAR and V functions.

Syntax

VARFMT(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

variable-number
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

If a format is not assigned to the variable, a blank string is returned.

Examples

Example 1: Using VARFMT to Obtain the Format of the Variable NAME

This example obtains the format of the variable NAME in the SAS data set MYDATA. Outside of a macro definition, this code works only for 9.4M5 and later.

```
%let dsid=%sysfunc(open(mydata, i));
%if &dsid %then
   %do;
      %let fmt=%sysfunc(varfmt(&dsid,
                        %sysfunc(varnum
                        (&dsid, NAME))));
      %let rc=%sysfunc(close(&dsid));
   %end;
```

Example 2: Using VARFMT to Obtain the Format of All the Numeric Variables in a Data Set

This example creates a data set that contains the name and formatted content of each numeric variable in the SAS data set MYDATA.

```
data vars;
   length name $ 8 content $ 12;
   drop dsid i num rc fmt;
   dsid=open("mydata", "i");
   num=attrn(dsid, "nvars");
   do while (fetch(dsid)=0);
      do i=1 to num;
         name=varname(dsid, i);
         if (vartype(dsid, i)='N') then do;
            fmt=varfmt(dsid, i);
            if fmt='' then fmt="BEST12.";
            content=putc(putn(getvarn
                        (dsid,i),fmt),"$char12.");
            output;
            end;
      end;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- “VARINFMT Function” on page 1497
- “VARNUM Function” on page 1504

VARINFMT Function

Returns the informat that is assigned to a SAS data set variable.

Category: SAS File I/O

Syntax

VARINFMT(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

variable-number
specifies the number of the variable's position in the SAS data set.

Tip The VARNUM function returns this number. This number is next to the variable in the list that is produced by the CONTENTS procedure.

Details

If no informat has been assigned to the variable, a blank string is returned.

Examples

Example 1: Using VARINFMT to Obtain the Informat of the Variable NAME

This example obtains the informat of the variable NAME in the SAS data set MYDATA.

```
data
mydata;

   informat name
$7.;

   input
name;

   label name='First
names';


datalines;

Mylinda

Joe

;




%macro
test(dsn);

   %let dsid=%sysfunc(open(&dsn,
i));

   %if &dsid %then
%do;

      %let fmt=%sysfunc(varinfmt(&dsid, %sysfunc(varnum (&dsid,
NAME))));

      %put
&=fmt;

      %let rc=
%sysfunc(close(&dsid));


%end;

%mend
test;



%test(mydata)
```

These statements produce this result:

```
FMT=$7.
```

Example 2: Using VARINFMT to Obtain the Informat of All the Variables in a Data Set

This example creates a data set that contains the name and informat of the variables in MYDATA.

```
data vars;
   length name $ 8 informat $ 10 ;
   drop dsid i num rc;
   dsid=open("mydata", "i");
   num=attrn(dsid, "nvars");
   do i=1 to num;
      name=varname(dsid, i);
      informat=varinfmt(dsid, i);
      output;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- “OPEN Function” on page 1183
- “VARFMT Function” on page 1495
- “VARNUM Function” on page 1504

VARLABEL Function

Returns the label that is assigned to a SAS data set variable.

Category: SAS File I/O

Syntax

VARLABEL(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

variable-number
specifies the number of the variable’s position in the SAS data set.

Tip The VARNUM function returns this number. This number is next to the variable in the list that is produced by the CONTENTS procedure. The VARNUM function returns this number.

Details

If no label has been assigned to the variable, a blank string is returned.

Comparisons

VLABEL returns the label that is associated with the given variable.

Example

This example obtains the label of the variable NAME in the SAS data set MYDATA.

Example Code 3.3 *Obtaining the Label of the Variable NAME*

```
data
mydata;

   input name
$;

   label name='First
names';

datalines;

Mylinda

Joe

;



%macro
test(dsn);

%let dsid=%sysfunc(open(&dsn,
i));

   %if &dsid %then
%do;
```

```
      %let fmt=%sysfunc(varlabel(&dsid, %sysfunc(varnum (&dsid,
NAME))));

   %put
&=fmt;

   %let rc=
%sysfunc(close(&dsid));

%end;

%mend
test;



%test(mydata)
```

These statements produce this result:

```
FMT=First names
```

See Also

Functions:

- “OPEN Function” on page 1183
- “VARNUM Function” on page 1504

VARLEN Function

Returns the length of a SAS data set variable.

Category: SAS File I/O

See: “Using the VAR and V Functions” for information about using the VAR and V functions.

Syntax

VARLEN(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

variable-number
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

VLENGTH returns the compile-time (allocated) size of the given variable.

Example

- This example obtains the length of the variable ADDRESS in the SAS data set MYDATA. Outside of a macro definition this code works only for SAS 9.4M5 and later.

```
%let dsid=%sysfunc(open(mydata, i));
%if &dsid %then
   %do;
      %let len=%sysfunc(varlen(&dsid,
                       %sysfunc(varnum
                       (&dsid,ADDRESS))));
      %let rc=%sysfunc(close(&dsid));
   %end;
```

- This example creates a data set that contains the name, type, and length of the variables in MYDATA.

```
data vars;
   length name $ 8 type $ 1;
   drop dsid i num rc;
   dsid=open("mydata", "i");
   num=attrn(dsid, "nvars");
   do i=1 to num;
      name=varname(dsid, i);
      type=vartype(dsid, i);
      length=varlen(dsid, i);
      output;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- "OPEN Function" on page 1183

■ "VARNUM Function" on page 1504

VARNAME Function

Returns the name of a SAS data set variable.

Category: SAS File I/O

See: "Using the VAR and V Functions" for information about using the VAR and V functions.

Syntax

VARNAME(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

variable-number
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Example

This example copies the names of the first five variables in the SAS data set CITY (or all of the variables if there are fewer than five) into a macro variable.

```
%macro names;
   %let dsid=%sysfunc(open(city, i));
   %let varlist=;
   %do i=1 %to
      %sysfunc(min(5,%sysfunc(attrn
                              (&dsid, nvars))));
      %let varlist=&varlist %sysfunc(varname
                                  (&dsid, &i));
   %end;
   %put varlist=&varlist;
%mend names;
%names
```

See Also

Functions:

- “OPEN Function” on page 1183
- “VARNUM Function” on page 1504

VARNUM Function

Returns the number of a variable's position in a SAS data set.

Category: SAS File I/O

Syntax

VARNUM(*data-set-id*, *variable-name*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

variable-name
specifies the variable's name.

Details

VARNUM returns the number of a variable's position in a SAS data set, or 0 if the variable is not in the SAS data set. This is the same variable number that is next to the variable in the output from PROC CONTENTS.

Example

- This example obtains the number of a variable's position in the SAS data set CITY, given the name of the variable.

```
%let dsid=%sysfunc(open(city, i));
%let citynum=%sysfunc(varnum(&dsid, CITYNAME));
%let rc=%sysfunc(fetch(&dsid));
%let cityname=%sysfunc(getvarc
                      (&dsid, &citynum));
```

- This example creates a data set that contains the name, type, format, informat, label, length, and position of the variables in Sasuser.Houses.

```
data vars;
   length name $ 8 type $ 1
          format informat $ 10 label $ 40;
   drop dsid i num rc;
   dsid=open("sasuser.houses", "i");
   num=attrn(dsid, "nvars");
   do i=1 to num;
      name=varname(dsid, i);
      type=vartype(dsid, i);
      format=varfmt(dsid, i);
      informat=varinfmt(dsid, i);
      label=varlabel(dsid, i);
      length=varlen(dsid, i);
      position=varnum(dsid, name);
      output;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- “OPEN Function” on page 1183
- “VARNAME Function” on page 1503

VARRAY Function

Returns a value that indicates whether the specified name is an array.

Category: Variable Information

Restriction: Use only with the DATA step

Note: This function supports the VARCHAR type.

Syntax

VARRAY(*name*)

Required Argument

name
specifies a name that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

VARRAY returns 1 if the given name is an array; it returns 0 if the given name is not an array.

Comparisons

- VARRAY returns a value that indicates whether the specified name is an array. VARRAYX returns a value that indicates whether the value of the specified expression is an array.
- VARRAY does not accept an expression as an argument. VARRAYX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;


a=varray(x);


b=varray(x1);

   put
a=;

   put
b=;

run;
```

These statements produce this result:

```
a=1
b=0
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VARRAYX Function

Returns a value that indicates whether the value of the specified argument is an array.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VARRAYX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression.

Restriction The value of the specified expression cannot denote an array reference.

Details

VARRAYX returns 1 if the value of the given argument is the name of an array; VARRAYX returns 0 if the value of the given argument is not the name of an array.

Comparisons

- VARRAY returns a value that indicates whether the specified name is the name of an array. VARRAYX returns a value that indicates whether the value of the specified expression is the name of an array.
- VARRAY does not accept an expression as an argument. VARRAYX accepts expressions, but the value of the specified variable cannot denote an array reference.

- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;

   array vx(4) $6 vx1 vx2 vx3 vx4 ('x' 'x1' 'x2'
'x3');


y=varrayx(vx(1));


z=varrayx(vx(2));

   put
y=;

   put
z=;

run;
```

These statements produce this result:

```
y=1
z=0
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VARTYPE Function

Returns the data type of a SAS data set variable.

Category: SAS File I/O

Note: The VARTYPE function supports integer variables: INT64, INT32, INT16, INT8.

See: "Using the VAR and V Functions" for information about using the VAR and V functions.

Syntax

VARTYPE(*data-set-id*, *variable-number*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

variable-number
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

VARTYPE returns C for a character variable or N for a numeric variable.

Example: Using VARTYPE to Determine Which Variables Are Character

This example creates a data set that contains the name and formatted contents of each character variable in the SAS data set MYDATA.

```
%let
mydata=sashelp.cars;



data
vars;

   length name $ 8 content $
20;

   drop dsid i num fmt
rc;
```

```
   dsid=open("&mydata",
"i");

   num=attrn(dsid,
"nvars");

   do while
(fetch(dsid)=0);

      do i=1 to
num;

   name=varname(dsid,
i);

   fmt=varfmt(dsid,
i);

   if (vartype(dsid, i)='C') then
do;

   content=getvarc(dsid,
i);

   if (fmt ne '' )
then

   content=left(putc(content,
fmt));


output;

end;

end;

end;

rc=close(dsid);

run;



proc
print;

run;
```

These statements produce this result. This output is a partial output.

The SAS System

Obs	name	content
1	Make	Acura
2	Model	MDX
3	Type	SUV
4	Origin	Asia
5	DriveTra	All
6	Make	Acura
7	Model	RSX Type S 2dr
8	Type	Sedan
9	Origin	Asia
10	DriveTra	Front
11	Make	Acura
12	Model	TSX 4dr
13	Type	Sedan
14	Origin	Asia
15	DriveTra	Front
16	Make	Acura
17	Model	TL 4dr
18	Type	Sedan
19	Origin	Asia
20	DriveTra	Front

See Also

Functions:

- “VARNUM Function” on page 1504

VERIFY Function

Returns the position of the first character in a string that is not in specified data strings.

Category: Character

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Note: This function supports the VARCHAR type.

Tip: The DBCS equivalent function is KVERIFY .

Syntax

VERIFY(*source*, *excerpt-1* <, ...*excerpt-n*>)

Required Arguments

source
specifies a character constant, variable, or expression.

excerpt
specifies a character constant, variable, or expression. If you specify more than one excerpt, separate the excerpts with a comma.

Details

The VERIFY function returns the position of the first character in *source* that is not present in any *excerpt*. If VERIFY finds every character in *source* in at least one *excerpt*, VERIFY returns a 0.

Example

```
data scores;
   input Grade : $1. @@;
   check='abcdf';
   if verify(grade, check)>0 then
      put @1 'INVALID ' grade=;
   datalines;
a b c b c d f a a q a b d d b
;
```

These statements produce this result:

```
INVALID Grade=q
```

See Also

Functions:

- “FINDC Function” on page 683

VFORMAT Function

Returns the format that is associated with the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step.

Note: This function supports the VARCHAR type.

Syntax

VFORMAT(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VFORMAT 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.

VFORMAT returns the complete format name, which includes the width and the period (for example, $CHAR20.).

VFORMAT returns BEST*w*. or $*w*. if no format has been assigned to the variable.

Comparisons

- VFORMAT returns the format that is associated with the specified variable. However, VFORMATX evaluates the argument to determine the variable name. The function then returns the format that is associated with that variable name.
- VFORMAT does not accept an expression as an argument. VFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array x(3) x1-
x3;

   format x1
comma8.2;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vformat(x(1));

   put
y=;

run;
```

SAS writes the following output to the log:

```
y=COMMA8.2
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VFORMATD Function

Returns the decimal value of the format that is associated with the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VFORMATD(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VFORMATD returns the format decimal value that is associated with the specified variable. However, VFORMATDX evaluates the argument to determine the variable name. The function then returns the format decimal value that is associated with that variable name.
- VFORMATD does not accept an expression as an argument. VFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the

Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array x(3) x1-
x3;

   format x1
comma8.2;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vformatd(x(1));

   put
y=;

run;
```

These statements produce this result:

```
y=2
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VFORMATDX Function

Returns the decimal value of the format that is associated with the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VFORMATDX(*expression*)

Required Argument

expression
specifies a SAS character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

- VFORMATD returns the format decimal value that is associated with the specified variable. However, VFORMATDX evaluates the argument to determine the variable name. The function then returns the format decimal value that is associated with that variable name.
- VFORMATD does not accept an expression as an argument. VFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array x(3) x1-
x3;

   format x1
comma8.2;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vformatdx(vx(1));


z=vformatdx('x'||'1');

   put
y=;

   put
z=;

run;
```

These statements produce these results:

```
y=2
z=2
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VFORMATN Function

Returns the format name that is associated with the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VFORMATN(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VFORMATN 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.

VFORMATN returns only the format name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VFORMATN returns the format name that is associated with the specified variable. However, VFORMATNX evaluates the argument to determine the variable name. The function then returns the format name that is associated with that variable name.

- VFORMATN does not accept an expression as an argument. VFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array vx(3) x1-
x3;

   format x1
best6.;


y=vformatn(vx(1));

   put
y=;

run;
```

These statements produce this result:

```
y=BEST
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VFORMATNX Function

Returns the format name that is associated with the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VFORMATNX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VFORMATNX 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.

VFORMATNX returns only the format name, which does not include the length or the period (for example, $CHAR).

Comparisons

- VFORMATN returns the format name that is associated with the specified variable. However, VFORMATNX evaluates the argument to determine the variable name. The function then returns the format name that is associated with that variable name.

- VFORMATN does not accept an expression as an argument. VFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');

   format x1
best6.;


y=vformatnx(vx(1));

   put
y=;

run;
```

These statements produce this result:

```
y=BEST
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VFORMATW Function

Returns the format width that is associated with the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VFORMATW(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VFORMATW returns the format width that is associated with the specified variable. However, VFORMATWX evaluates the argument to determine the variable name. The function then returns the format width that is associated with that variable name.
- VFORMATW does not accept an expression as an argument. VFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');

   format x1
best6.;


y=vformatw(vx(1));

   put
y=;

run;
```

These statements produce this result:

```
y=6
```

See Also

Functions:

- Variable Information functions in “SAS Functions and CALL Routines by Category” on page 100

VFORMATWX Function

Returns the format width that is associated with the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VFORMATWX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Comparisons

- VFORMATW returns the format width that is associated with the specified variable. However, VFORMATWX evaluates the argument to determine the variable name. The function then returns the format width that is associated with that variable name.
- VFORMATW does not accept an expression as an argument. VFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;
```

```
   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');

   format x1
best6.;

   format x2
20.10;


y=vformatwx(vx(1));


z=vformatwx(vx(2));

   put
y=;

   put
z=;

run;
```

These statements produce these results:

```
y=6
z=20
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VFORMATX Function

Returns the format that is associated with the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VFORMATX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VFORMATX 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.

VFORMATX returns the complete format name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VFORMAT returns the format that is associated with the specified variable. However, VFORMATX evaluates the argument to determine the variable name. The function then returns the format that is associated with that variable name.
- VFORMAT does not accept an expression as an argument. VFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 3. x2 6. x3 :
comma10.3;


datalines;

123 456789
$1,023,948

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');

   format x1
best6.;

   format x2
20.10;


y=vformatx(vx(1));


z=vformatx(vx(2));

   put
y=;

   put
z=;

run;
```

These statements produce these results:

```
y=BEST6.
z=F20.10
```

See Also

Functions:

- Variable Information functions in"SAS Functions and CALL Routines by Category" on page 100

VINARRAY Function

Returns a value that indicates whether the specified variable is a member of an array.

Category: Variable Information

Restriction: Use only with the DATA step.

Note: This function supports the VARCHAR type.

Syntax

VINARRAY(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

VINARRAY returns 1 if the given variable is a member of an array; VINARRAY returns 0 if the given variable is not a member of an array.

Comparisons

- VINARRAY returns a value that indicates whether the specified variable is a member of an array. However, VINARRAYX evaluates the argument to determine the variable name. The function then returns a value that indicates whether the variable name is a member of an array.
- VINARRAY does not accept an expression as an argument. VINARRAYX accepts expressions, but the value of the specified expression cannot denote an array reference.

- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;


y=vinarray(x);


Z=vinarray(x1);

   put
y=;

   put
z=;

run;
```

These statements produce these results:

```
y=0
z=1
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINARRAYX Function

Returns a value that indicates whether the value of the specified argument is a member of an array.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINARRAYX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

VINARRAYX returns 1 if the value of the given argument is a member of an array; VINARRAYX returns 0 if the value of the given argument is not a member of an array.

Comparisons

- VINARRAY returns a value that indicates whether the specified variable is a member of an array. However, VINARRAYX evaluates the argument to determine the variable name. The function then returns a value that indicates whether the variable name is a member of an array.
- VINARRAY does not accept an expression as an argument. VINARRAYX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;

   array vx(4) $6 vx1 vx2 vx3 vx4 ('x' 'x1' 'x2'
'x3');
```

```
y=vinarrayx(vx(1));


z=vinarrayx(vx(2));

   put
y=;

   put
z=;

run;
```

These statements produce these results:

```
y=0
z=1
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMAT Function

Returns the informat that is associated with the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step.

Note: This function supports the VARCHAR type.

Syntax

VINFORMAT(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VINFORMAT 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.

VINFORMAT returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VINFORMAT returns the informat that is associated with the specified variable. However, VINFORMATX evaluates the argument to determine the variable name. The function then returns the informat that is associated with that variable name.
- VINFORMAT does not accept an expression as an argument. VINFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   infile
datalines;

   informat x1 x2 x3
comma9.3;

   input x1 x2
x3;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vinformat(x1);

   put
y=;

datalines;
```

```
$1,500,000  $2,500,000
$3,500,000

;
```

These statements produce this result:

```
y=COMMA9.3
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMATD Function

Returns the decimal value of the informat that is associated with the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINFORMATD(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VINFORMATD returns the informat decimal value that is associated with the specified variable. However, VINFORMATDX evaluates the argument to determine the variable name. The function then returns the informat decimal value that is associated with that variable name.

- VINFORMATD does not accept an expression as an argument. VINFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   infile
datalines;

   informat x1 x2 x3
comma9.3;

   input x1 x2
x3;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vinformatd(x1);

   put
y=;

datalines;

$1,500,000  $2,500,000
$3,500,000

;
```

These statements produce this result:

```
y=3
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMATDX Function

Returns the decimal value of the informat that is associated with the value of the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINFORMATDX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified variable cannot denote an array reference.

Comparisons

- VINFORMATD returns the informat decimal value that is associated with the specified variable. However, VINFORMATDX evaluates the argument to determine the variable name. The function then returns the informat decimal value that is associated with that variable name.
- VINFORMATD does not accept an expression as an argument. VINFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   infile
datalines;
```

```
   informat x1 x2 x3
comma9.3;

   input x1 x2
x3;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vinformatdx(vx(1));

   put
y=;

datalines;

$1,500,000  $2,500,000
$3,500,000

;
```

These statements produce this result:

```
y=3
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMATN Function

Returns the informat name that is associated with the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINFORMATN(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VINFORMATN 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.

VINFORMATN returns only the informat name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VINFORMATN returns the informat name that is associated with the specified variable. However, VINFORMATNX evaluates the argument to determine the variable name, and then returns the informat name that is associated with that variable name.
- VINFORMATN does not accept an expression as an argument. VINFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 $3. x2 $6. x3
$4.;


datalines;

abc defghi
jklm

;
```

```
data
new;

   set
one;

   informat x1 x2 x3
$char6.;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vinformatn(x2);

   put
y=;

run;
```

These statements produce this result:

```
y=$CHAR
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMATNX Function

Returns the informat name that is associated with the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINFORMATNX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VINFORMATNX 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.

VINFORMATNX returns only the informat name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VINFORMATN returns the informat name that is associated with the specified variable. However, VINFORMATNX evaluates the argument to determine the variable name, and then returns the informat name that is associated with that variable name.
- VINFORMATN does not accept an expression as an argument. VINFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 $3. x2 $6. x3
$4.;


datalines;

abc defghi
jklm
```

```
;



data
new;

   set
one;

   informat x1 x2 x3
$char6.;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vinformatnx(vx(2));

   put
y=;

run;
```

These statements produce this result:

```
y=$CHAR
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMATW Function

Returns the informat width that is associated with the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINFORMATW(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VINFORMATW returns the informat width that is associated with the specified variable. However, VINFORMATWX evaluates the argument to determine the variable name, and then returns the informat width that is associated with that variable name.
- VINFORMATW does not accept an expression as an argument. VINFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, type, and length, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 $3. x2 $8. x3
$4.;


datalines;

abc defghi
jklm

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');
```

```
y=vinformatw(vx(2));

   put
y=;

run;
```

These statements produce this result:

```
y=6
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMATWX Function

Returns the informat width that is associated with the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINFORMATWX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Comparisons

- VINFORMATW returns the informat width that is associated with the specified variable. However, VINFORMATWX evaluates the argument to determine the

variable name, and then returns the informat width that is associated with that variable name.

- VINFORMATW does not accept an expression as an argument. VINFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 $3. x2 $8. x3
$4.;


datalines;

abc defghi
jklm

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vinformatwx(vx(2));

   put
y=;

run;
```

These statements produce this result:

```
y=8
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VINFORMATX Function

Returns the informat that is associated with the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VINFORMATX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VINFORMATX 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.

VINFORMATX returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VINFORMAT returns the informat that is associated with the specified variable. However, VINFORMATX evaluates the argument to determine the variable name, and then returns the informat that is associated with that variable name.

- VINFORMAT does not accept an expression as an argument. VINFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, length, and type, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 $3. x2 $8. x3
$4.;


datalines;

abc defghi
jklm

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vinformatx(vx(2));

   put
y=;

run;
```

These statements produce this result:

```
y=$8.
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VLABEL Function

Returns the label that is associated with the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step.

Note: This function supports the VARCHAR type.

Syntax

VLABEL(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VLABEL 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 there is no label, VLABEL returns the variable name.

Comparisons

- VLABEL returns the label of the specified variable or the name of the specified variable, if no label exists. However, VLABELX evaluates the argument to determine the variable name, and then returns the label that is associated with that variable name, or the variable name if no label exists.
- VLABEL does not accept an expression as an argument. VLABELX accepts expressions, but the value of the specified expression cannot denote an array reference.
- VLABEL has the same functionality as CALL LABEL.

- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;

   label
x1='Test1';


y=vlabel(x(1));

   put
y=;

run;
```

These statements produce this result:

```
y=Test1
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VLABELX Function

Returns the label that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VLABELX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VLABELX 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 there is no label, VLABELX returns the variable name.

Comparisons

- VLABEL returns the label of the specified variable or the name of the specified variable, if no label exists. However, VLABELX evaluates the argument to determine the variable name, and then returns the label that is associated with that variable name, or the variable name if no label exists.
- VLABEL does not accept an expression as an argument. VLABELX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 $3. x2 $8. x3
$4.;


datalines;

abc defghi
jklm

;
```

```
data
new;

   set
one;

   label
x1='Test1';

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');

   do i = 1 to
3;


l=vlabelx(vx(i));

    put
l=;


end;

run;
```

These statements produce these results:

```
l=Test1
l=x2
l=x3
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VLENGTH Function

Returns the compile-time (allocated) size of the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step.

Note: This function supports the VARCHAR type.

Syntax

VLENGTH(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- LENGTH examines the variable at run time, trimming trailing blanks to determine the length. VLENGTH returns a compile-time constant value, which reflects the maximum length.
- LENGTHC returns the same value as VLENGTH, but LENGTHC can be used in any calling environment and its argument can be any expression.
- VLENGTH returns the length of the specified variable. However, VLENGTHX evaluates the argument to determine the variable name, and then returns the compile-time size that is associated with that variable name.
- VLENGTH does not accept an expression as an argument. VLENGTHX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data _null_;
   length x $8;
   x='abc';
   y=vlength(x);
   z=length(x);
   put y=;
   put z=;
run;
```

These statements produce these results:

```
y=8
z=3
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VLENGTHX Function

Returns the compile-time (allocated) size for the variable with a name that is the same as the value of the argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VLENGTHX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Comparisons

- LENGTH examines the variable at run time, trimming trailing blanks to determine the length. However, VLENGTHX evaluates the argument to determine the variable name, and then returns the compile-time size that is associated with that variable name.
- LENGTHC accepts an expression as the argument but returns the length of the value of the expression, not the length of the variable that has a name equal to the value of the expression.
- VLENGTH returns the length of the specified variable. VLENGTHX returns the length for the value of the specified expression.
- VLENGTH does not accept an expression as an argument. VLENGTHX accepts expressions, but the value of the specified expression cannot denote an array reference.

- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
one;

   input x1 $3. x2 $8. x3
$4.;

datalines;

abc defghi
jklm

;



data
new;

   set
one;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');

   array x(3)
$;

    do i = 1 to
3;


y=vlengthx(vx(i));


z=length(x(i));

   put
y=;

   put
z=;


end;

run;
```

These statements produce these results:

```
y=3
z=3
y=8
z=6
y=4
z=4
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VNAME Function

Returns the name of the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step.

Note: This function supports the VARCHAR type.

Syntax

VNAME(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VNAME 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.

Comparisons

- VNAME returns the name of the specified variable. However, VNAMEX evaluates the argument to determine a variable name. If the name is a known variable name, the function returns that name. Otherwise, the function returns a blank.
- VNAME does not accept an expression as an argument. VNAMEX accepts expressions, but the value of the specified expression cannot denote an array reference.
- VNAME has the same functionality as CALL VNAME.
- Related functions return the value of other variable attributes such as the variable label, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;


y=vname(x(1));

   put
y=;

run;
```

These statements produce this result:

```
y=x1
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VNAMEX Function

Validates the value of the specified argument as a variable name.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VNAMEX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VNAMEX 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.

Comparisons

- VNAME returns the name of the specified variable. However, VNAMEX evaluates the argument to determine a variable name. If the name is a known variable name, the function returns that name. Otherwise, the function returns a blank.
- VNAME does not accept an expression as an argument. VNAMEX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable label, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vnamex(vx(1));


z=vnamex('x'||'1');

   put
y=;

   put
z=;

run;
```

These statements produce these results:

```
y=x1
z=x1
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VTYPE Function

Returns the type (character or numeric) of the specified variable.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VTYPE(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VTYPE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 1.

VTYPE returns N for numeric variables, C for character variables, and V for VARCHAR variables.

Comparisons

- VTYPE returns the type of the specified variable. However, VTYPEX evaluates the argument to determine the variable name, and then returns the type (character or numeric) that is associated with that variable name.
- VTYPE does not accept an expression as an argument. VTYPEX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;


y=vtype(x(1));
```

```
   put
y=;

run;
```

SAS writes the following output to the log:

```
y=N
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VTYPEX Function

Returns the type (character or numeric) for the value of the specified argument.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VTYPEX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VTYPEX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 1.

VTYPEX returns C for character variables, N for numeric variables, and V for VARCHAR variables.

Comparisons

- VTYPE returns the type of the specified variable. However, VTYPEX evaluates the argument to determine the variable name, and then returns the type (character or numeric) that is associated with that variable name.
- VTYPE does not accept an expression as an argument. VTYPEX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes such as the variable name, informat, and format, among others. For a complete list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100.

Example

```
data
new;

   array x(3) x1-
x3;

   array vx(3) $6 vx1 vx2 vx3 ('x1' 'x2'
'x3');


y=vtypex(vx(1));

   put
y=;

run;
```

These statements produce this result:

```
y=N
```

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

VVALUE Function

Returns the formatted value that is associated with the variable that you specify.

Category: Variable Information

Restriction: Use only with the DATA step.

Note: This function supports the VARCHAR type.

Syntax

VVALUE(*variable*)

Required Argument

variable
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VVALUE 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.

VVALUE returns a character string that contains the current value of the variable that you specify. The value is formatted using the current format that is associated with the variable.

Comparisons

- VVALUE returns the value that is associated with the variable that you specify. However, VVALUEX evaluates the argument to determine the variable name, and then returns the value that is associated with that variable name.
- VVALUE does not accept an expression as an argument. VVALUEX accepts expressions, but the value of the expression cannot denote an array reference.
- VVALUE and an assignment statement both return a character string that contains the current value of the variable that you specify. With VVALUE, the value is formatted using the current format that is associated with the variable. However, with an assignment statement, the value is unformatted.

- The PUT function enables you to reformat a specified variable or constant. VVALUE uses the current format that is associated with the variable.

Example

```
data
one;

   input
val;

datalines;

9999

1000.50

;



data
new;

   set
one;

   format val
comma10.2;


v=vvalue(val);

   put
v;

run;
```

These statements produce these values:

```
9,999.00
1,000.50
```

See Also

Functions:

- “VVALUEX Function” on page 1563
- Variable Information functions in “SAS Functions and CALL Routines by Category” on page 100

VVALUEX Function

Returns the formatted value that is associated with the argument that you specify.

Category: Variable Information

Note: This function supports the VARCHAR type.

Syntax

VVALUEX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VVALUEX 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.

VVALUEX returns a character string that contains the current value of the argument that you specify. The value is formatted by using the format that is currently associated with the argument.

Comparisons

- VVALUE accepts a variable as an argument and returns the value of that variable. However, VVALUEX accepts a character expression as an argument, evaluates the expression to determine the variable name, and then returns the value that is associated with that variable name.
- VVALUE does not accept an expression as an argument, but it does accept array references. VVALUEX accepts expressions, but the value of the expression cannot denote an array reference.
- VVALUEX and an assignment statement both return a character string that contains the current value of the variable that you specify. With VVALUEX, the

value is formatted by using the current format that is associated with the variable. However, with an assignment statement the value is unformatted.

- The PUT function enables you to reformat a specified variable or constant. VVALUEX uses the current format that is associated with the variable.

Example

```
data
one;

input date
date7.;

datalines;

31mar18

;



data
new;

 set
one;


date2='date';

  format date
date7.;


datevalue=vvaluex(date2);

  put
datevalue=;

run;
```

These statements produce this result:

```
datevalue=31MAR18
```

See Also

Functions:

- “VVALUE Function” on page 1561

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 100

WEEK Function

Returns the week-number value.

Category: Date and Time

Syntax

WEEK(<*sas-date*>, <*'descriptor'*>)

Optional Arguments

sas-date
specifies the SAS data value. If the *sas-date* argument is not specified, the WEEK function returns the week-number value of the current date.

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('31dec2020'd, 'u')` is 52. 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 1566

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 1566

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('31dec2020'd, 'w')` is 52.

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 1566

Default U

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 52. 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('01jan2020'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, 01jan2006 and 30dec2005 occur in the same week. The first day (Monday) of that week is 26dec2005. Therefore, `week('01jan2006'd, 'v')` and `week('30dec2005'd, 'v')` both return a value of 52. This means that both dates occur in week 52 of the year 2005.

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('01feb2021'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 example shows the values of the U, V, and W descriptors for dates near the end of certain years and the beginning of the next year. Examining the full data set illustrates how the behavior can differ between the various descriptors depending on the day of the week for January 1. The output displays the first 20 observations:

```
title 'Values of the U, V, and W Descriptors';
data a(drop=i date0 date1 y);
     date0='20dec2020'd;
     do y=0 to 5;
        date1=intnx("YEAR", date0, y, 's');
        do i=0 to 20;
          date=intnx("DAY", date1, i);
          year=YEAR(date);
          week=week(date);
```

```
            week_u=week(date, 'u');
            week_v=week(date, 'v');
            week_w=week(date, 'w');
            output;
         end;
      end;
      format date WEEKDATX17.;
run;
proc print;
run;
```

Output 3.78 *Partial Output from Identifying the Values of the U, V, and W Descriptors*

Values of the U, V, and W Descriptors

Obs	date	year	week	week_u	week_v	week_w
1	Sun, 20 Dec 2020	2020	51	51	51	50
2	Mon, 21 Dec 2020	2020	51	51	52	51
3	Tue, 22 Dec 2020	2020	51	51	52	51
4	Wed, 23 Dec 2020	2020	51	51	52	51
5	Thu, 24 Dec 2020	2020	51	51	52	51
6	Fri, 25 Dec 2020	2020	51	51	52	51
7	Sat, 26 Dec 2020	2020	51	51	52	51
8	Sun, 27 Dec 2020	2020	52	52	52	51
9	Mon, 28 Dec 2020	2020	52	52	53	52
10	Tue, 29 Dec 2020	2020	52	52	53	52
11	Wed, 30 Dec 2020	2020	52	52	53	52
12	Thu, 31 Dec 2020	2020	52	52	53	52
13	Fri, 1 Jan 2021	2021	0	0	53	0
14	Sat, 2 Jan 2021	2021	0	0	53	0
15	Sun, 3 Jan 2021	2021	1	1	53	0
16	Mon, 4 Jan 2021	2021	1	1	1	1
17	Tue, 5 Jan 2021	2021	1	1	1	1
18	Wed, 6 Jan 2021	2021	1	1	1	1
19	Thu, 7 Jan 2021	2021	1	1	1	1
20	Fri, 8 Jan 2021	2021	1	1	1	1

See Also

Functions:

- “INTNX Function” on page 1004

Formats:

- “WEEKUw. Format” in *SAS Formats and Informats for SAS Viya Workbench*

- "WEEKVw. Format" in *SAS Formats and Informats for SAS Viya Workbench*
- "WEEKWw. Format" in *SAS Formats and Informats for SAS Viya Workbench*

Informats:

- "WEEKUw. Informat" in *SAS Formats and Informats for SAS Viya Workbench*
- "WEEKVw. Informat" in *SAS Formats and Informats for SAS Viya Workbench*
- "WEEKWw. Informat" in *SAS Formats and Informats for SAS Viya Workbench*

WEEKDAY Function

From a SAS date value, returns an integer that corresponds to the day of the week.

Category: Date and Time

Syntax

WEEKDAY(*date*)

Required Argument

date
specifies a SAS expression that represents a SAS date value.

Details

The WEEKDAY function produces an integer that represents the day of the week, where 1=Sunday, 2=Monday, ..., 7=Saturday.

Example

```
data one;
input date date7.;
datalines;
25dec21;
data new;
set one;
   y=weekday(date);
   put y=;
run;
```

These statements produce this result:

```
y=7
```

WHICHC Function

Searches for a character value that is equal to the first argument, and returns the index of the first matching value.

Category: Search

Restriction: This function is assigned an I18N Level 2 status, and is designed for use with SBCS, DBCS, and MBCS (UTF8). For more information, see Internationalization Compatibility.

Syntax

WHICHC(*string*, *value-1* <, *value-2*, ..., *value-n*>)

Required Arguments

string
is a character constant, variable, or expression that specifies the value to search for.

value
is a character constant, variable, or expression that specifies the value to be searched.

Details

The WHICHC function searches the second and subsequent arguments for a value that is equal to the first argument, and returns the index of the first matching value.

If *string* is missing, then WHICHC returns a missing value. Otherwise, WHICHC compares the value of *string* with *value-1*, *value-2*, and so on, in sequence. If argument *value-i* equals *string*, then WHICHC returns the positive integer *i*. If *string* does not equal any subsequent argument, then WHICHC returns 0.

Using WHICHC is useful when the values that are being searched are subject to frequent change. If you need to perform many searches without changing the values that are being searched, using the HASH object is much more efficient.

Example

The following example searches the array for the first argument and returns the index of the first matching value.

```
data _null_;
   array fruit (*) $12 fruit1-fruit3 ('watermelon' 'apple' 'banana');
   x1=whichc('watermelon', of fruit[*]);
   x2=whichc('banana', of fruit[*]);
   x3=whichc('orange', of fruit[*]);
   put x1= / x2= / x3=;
run;
```

These statements produce these results:

```
x1=1
x2=3
x3=0
```

See Also

Functions:

- "WHICHN Function" on page 1571

Other References:

- "SAS Operators" in *SAS Language Programmer's Guide for SAS Viya Workbench*
- "Using the Hash and Hash Iterator Objects" in *SAS Component Objects: Reference*

WHICHN Function

Searches for a numeric value that is equal to the first argument, and returns the index of the first matching value.

Category: Search

Syntax

WHICHN(*argument*, *value-1* <, *value-2*, ..., *value-n*>)

Required Arguments

argument
is a numeric constant, variable, or expression that specifies the value to search for.

value
is a numeric constant, variable, or expression that specifies the value to be searched.

Details

The WHICHN function searches the second and subsequent arguments for a value that is equal to the first argument, and returns the index of the first matching value.

If *argument* is missing, then WHICHN returns a missing value. Otherwise, WHICHN compares the value of *argument* with *value-1*, *value-2*, and so on, in sequence. If *value-i* equals *argument*, then WHICHN returns the positive integer *i*. If *argument* does not equal any subsequent argument, then WHICHN returns 0.

Using WHICHN is useful when the values that are being searched are subject to frequent change. If you need to perform many searches without changing the values that are being searched, using the HASH object is much more efficient.

Example

The following example searches the array for the first argument and returns the index of the first matching value.

```
data _null_;
   array dates[*] Columbus Hastings Nicea US_Independence missing
                  Magna_Carta Gutenberg
                  (1492 1066 325 1776 . 1215 1450);
   x0=whichn(., of dates[*]);
   x1=whichn(1492, of dates[*]);
   x2=whichn(1066, of dates[*]);
   x3=whichn(1450, of dates[*]);
   x4=whichn(1000, of dates[*]);
   put x0= / x1= / x2= / x3= / x4=;
run;
```

These statements produce these results:

```
x0=.
x1=1
x2=2
x3=7
x4=0
```

See Also

Functions:

- "WHICHC Function" on page 1570

Other References:

- "SAS Operators" in *SAS Language Programmer's Guide for SAS Viya Workbench*
- "Using the Hash and Hash Iterator Objects" in *SAS Component Objects: Reference*

YEAR Function

Returns the year from a SAS date value.

Category: Date and Time

Syntax

YEAR(*date*)

Required Argument

date
specifies a SAS expression that represents a SAS date value.

Details

The YEAR function produces a four-digit numeric value that represents the year.

Example

```
data one;
   input date date7.;
datalines;
25dec21;
data new;
   set one;
   y=year(date);
   put y;
run;
```

These statements produce this result:

```
y=2021
```

See Also

Functions:

- “DAY Function” on page 521
- “MONTH Function” on page 1120

YIELDP Function

Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.

Category: Financial

Syntax

YIELDP(*A*, *c*, *n*, *K*, k_0, *p*)

Required Arguments

A

specifies the face value.

Range $A > 0$

c

specifies the nominal annual coupon rate, expressed as a fraction.

Range $0 \leq c < 1$

n

specifies the number of coupons per year.

Range $n > 0$ and is an integer

K

specifies the number of remaining coupons from settlement date to maturity.

Range $K > 0$ and is an integer

***k*$_0$**
specifies the time from settlement date to the next coupon as a fraction of the annual basis.

Range $0 < k_0 \leq \frac{1}{n}$

p
specifies the price with accrued interest.

Range $p > 0$

Details

The YIELDP function is based on the following relationship:

$$P = \sum_{k=1}^{K} c(k) \frac{1}{\left(1 + \frac{y}{n}\right)^{tk}}$$

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 example, 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_;
   y=yieldp(1000, .01, 4, 14, .165, 800);
   put y=;
run;
```

These statements produce this result:

```
y=0.0775031248
```

YRDIF Function

Returns the difference in years between two dates according to specified day count conventions; returns a person's age.

Category: Date and Time

Syntax

YRDIF(*start-date*, *end-date*, < *basis*>)

Required Arguments

start-date
specifies a SAS date value that identifies the starting date.

end-date
specifies a SAS date value that identifies the ending date.

Optional Argument

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 is computed.

If you do not specify a third argument, AGE becomes the default value for *basis*.

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 include 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 example, YRDIF returns the difference in years between two dates based on each of the options for *basis*.

```
data _null_;
   sdate='16oct2008'd;
```

```
   edate='16feb2021'd;
   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= / yactact= / yact360= / yact365= ;
run;
```

These statements produce these results:

```
y30360=12.333333333
yactact=12.336409911
yact360=12.516666667
yact365=12.345205479
```

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_;
   sdate='16oct2008'd;
   edate='16feb2021'd;
   age=yrdif(sdate, edate, 'AGE');
   put age= 'years';
run;
```

These statements produce this result:

```
age=12.336986301 years
```

See Also

Functions:

- "DATDIF Function" on page 513

References

"Day Count Convention." *Wikipedia*, 2010. Available Day count convention.

ISDA International Swaps and Derivatives Association, Inc "EMU and Market Conventions: Recent Developments." 1998. Wikipedia. Available EMU and Market Conventions: Recent Developments.

Mayle, Jan. 1994. *Standard Securities Calculation Methods – Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. NY, NY: Securities Industry Association.

YYQ Function

Returns a SAS date value from year and quarter year values.

Category: Date and Time

Syntax

YYQ(*year*, *quarter*)

Required Arguments

year
specifies a two-digit or four-digit integer that represents the year. The YEARCUTOFF= system option defines the year value for two-digit dates.

quarter
specifies the quarter of the year (1, 2, 3, or 4).

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 missing, or if the quarter value is not valid, the result is missing.

Example

```
data null;
   DateValue=yyq(2021,3);
   put DateValue;
   put DateValue date7.;
   put DateValue date9.;
run;
```

These statements produce these results:

```
22462
01JUL21
01JUL2021
```

```
data null;
   StartOfQtr=yyq(21,4);
   put StartOfQtr;
   put StartOfQtr=worddate.;
```

```
run;
```

These statements produce these results:

```
22554
StartOfQtr=October 1, 2021
```

See Also

Functions:

- "QTR Function" on page 1292
- "YEAR Function" on page 1573

System Options:

- "YEARCUTOFF= System Option" in *SAS System Options for SAS Viya Workbench*

ZIPCITY Function

Returns a city name and the two-character postal code that corresponds to a ZIP code.

Category: State and ZIP code

Syntax

ZIPCITY(*ZIP-code*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip If the value of *ZIP-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPCITY assumes that the value is 01040.

Details

The Basics

If the ZIPCITY function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPCITY returns a city name and the two-character postal code that corresponds to its five-digit ZIP code argument. ZIPCITY returns the character values in mixed-case. If the ZIP code is unknown, ZIPCITY returns a blank value.

Note: The Sashelp.Zipcode data set must be present when you use this function. If you remove the data set, ZIPCITY returns unexpected results.

How the ZIP Code Is Translated to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. The function then finds the corresponding state for that range of ZIP codes. The start and end ZIP code values for each state allow for exceptions.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Determining When the State Postal Code Table Was Last Updated

The Sashelp.Zipcode data set contains postal code information that is used with ZIPCITY and other ZIP code functions. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=sashelp.zipcode;
run;
```

Output from the CONTENTS procedure provides the date of the last update, along with the contents of the Sashelp.Zipcode data set.

Note: You can download the latest version of the Sashelp.Zipcode file from the Technical Support Web site. Select **Zipcode Data Set** from the Name column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set.

If you use an invalid ZIP code (one that is not found in the Sashelp.Zipcode data set), SAS returns a message that indicates there is an invalid argument in the ZIPCITY function.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the name of the city in mixed-case and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed-case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or worldwide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Example

This example returns a city name and the two-character state postal code that corresponds to a ZIP code.

```
data one;
   input zips $10.;
   datalines;
27511
30306
94110
96801
;

data new;

   set
one;

      if index(zips,'-') then city=zipcity(substr(zips, 1,
5));

      else
city=zipcity(zips);

   put zips
city;

run;
```

These statements produce these results:

```
27511 Cary, NC
30306 Atlanta, GA
94110 San Francisco, CA
96801 Honolulu, HI
```

See Also

Functions:

- “ZIPFIPS Function” on page 1584
- “ZIPNAME Function” on page 1586
- “ZIPNAMEL Function” on page 1588
- “ZIPSTATE Function” on page 1590

ZIPCITYDISTANCE Function

Returns the geodetic distance between two ZIP code locations.

Categories: Distance
State and ZIP code

Alias: ZIPCITYDIST

Syntax

ZIPCITYDISTANCE(*ZIP-code-1*, *ZIP-code-2*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains the ZIP code of a location in the United States of America.

Details

The ZIPCITYDISTANCE function returns the geodetic distance in miles between two ZIP code locations. The centroid of each ZIP code is used in the calculation.

The Sashelp.Zipcode data set must be present when you use this function. If you remove the data set, then ZIPCITYDISTANCE returns unexpected results.

The Sashelp.Zipcode data set contains postal code information that is used with ZIPCITYDISTANCE and other ZIP code functions. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=sashelp.zipcode;
run;
```

Output from the CONTENTS procedure provides the date of the last update, along with the contents of the SASHELP.ZIPCODE data set.

Note: You can download the latest version of the Sashelp.Zipcode file from the SAS external website. The file is located at the Technical Support Web site. Select **Zipcode Data Set** from the **Name** column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set. See Updating Zipcode Data Set for more information.

If you use an invalid ZIP code (one that is not found in the Sashelp.Zipcode data set), then SAS returns a message that indicates there is an invalid argument in the ZIPCITYDISTANCE function.

Example

In the following example, the first ZIP code identifies a location in San Francisco, CA, and the second ZIP code identifies a location in Bangor, ME. ZIPCITYDISTANCE returns the distance in miles between these two locations.

```
data _null_;
   distance=zipcitydistance('94103', '04401');
   put 'Distance from San Francisco, CA, to Bangor, ME: ' distance 4.
' miles';
run;
```

These statements produce this result:

```
Distance from San Francisco, CA, to Bangor, ME: 2782 miles
```

See Also

Functions:

- “ZIPCITY Function” on page 1580

ZIPFIPS Function

Converts ZIP codes to two-digit FIPS codes.

Category: State and ZIP code

Syntax

ZIPFIPS(*ZIP-code*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip If the value of *ZIP-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPFIPS assumes that the value is 01040.

Details

The Basics

The ZIPFIPS function returns the two-digit numeric U.S. Federal Information Processing Standards (FIPS) code that corresponds to its five-digit ZIP code argument.

Example

This example converts ZIP codes to two-digit FIPS codes.

```
data one;
   input zips $10.;
   datalines;
27511
01040
;

data new;
   set one;
   state=zipfips(zips);
run;

proc print data=new;
run;
```

Here are the two-digit FIPS codes.

Obs	zips	state
1	27511	37
2	01040	25

See Also

Functions:

- "ZIPCITY Function" on page 1580
- "ZIPNAME Function" on page 1586
- "ZIPNAMEL Function" on page 1588
- "ZIPSTATE Function" on page 1590

ZIPNAME Function

Converts ZIP codes to uppercase state names.

Category: State and ZIP code

Syntax

ZIPNAME(*ZIP-code*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip If the value of *ZIP-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPNAME assumes that the value is 01040.

Details

The Basics

If the ZIPNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPNAME returns the name of the state or U.S. territory that corresponds to its five-digit ZIP code argument. ZIPNAME returns character values up to 20 characters long, all in uppercase.

How the ZIP Code Is Translated to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. The function then finds the corresponding state for that range of ZIP codes. The start and end ZIP code values for each state allow for exceptions.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be

added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the mixed-case name of the city and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed-case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or worldwide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Example

This example converts ZIP codes to uppercase state names.

```
data one;
   input zips $10.;
   datalines;
27511
01040
59017
90049-1392
;

data new;
   set one;
   state=zipname(zips);
run;

proc print data=new;
run;
```

Here are the uppercase state names.

Obs	zips	state
1	27511	NORTH CAROLINA
2	01040	MASSACHUSETTS
3	59017	MONTANA
4	90049-1392	CALIFORNIA

See Also

Functions:

- "ZIPCITY Function" on page 1580
- "ZIPFIPS Function" on page 1584
- "ZIPNAMEL Function" on page 1588
- "ZIPSTATE Function" on page 1590

ZIPNAMEL Function

Converts ZIP codes to mixed-case state names.

Category: State and ZIP code

Syntax

ZIPNAMEL(*ZIP-code*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip If the value of *ZIP-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPNAMEL assumes that the value is 01040.

Details

The Basics

If the ZIPNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPNAMEL returns the name of the state or U.S. territory that corresponds to its five-digit ZIP code argument. ZIPNAMEL returns mixed-case character values up to 20 characters long.

How the ZIP Code Is Translated to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. The function then finds the corresponding state for that range of ZIP codes. The start and end ZIP code values for each state allow for exceptions.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the name of the city in mixed-case and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed-case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or worldwide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Example

This example converts ZIP codes to mixed-case state names.

```
data one;
   input zips $10.;
   datalines;
27511
01040
59017
```

```
90049-1392
;

data new;
   set one;
   state=zipnamel(zips);
run;

proc print data=new;
run;
```

Here are the mixed-case state names.

Obs	zips	state
1	27511	North Carolina
2	01040	Massachusetts
3	59017	Montana
4	90049-1392	California

See Also

Functions:

- “ZIPCITY Function” on page 1580
- “ZIPFIPS Function” on page 1584
- “ZIPNAME Function” on page 1586
- “ZIPSTATE Function” on page 1590

ZIPSTATE Function

Converts ZIP codes to two-character state postal codes.

Category: State and ZIP code

Syntax

ZIPSTATE(*ZIP-code*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains a valid five-digit ZIP code.

Tip If the value of *ZIP-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPSTATE assumes that the value is 01040.

Details

The Basics

If the ZIPSTATE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPSTATE returns the two-character state postal code (or worldwide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument. ZIPSTATE returns character values in uppercase.

Note: ZIPSTATE does not validate the ZIP code.

How the ZIP Code Is Converted to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. The function then finds the corresponding state for that range of ZIP codes. The zone table does not validate ZIP code values.

Typically, a zone does not span multiple states. Exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Army Post Office (APO) and Fleet Post Office (FPO) Postal Codes

The ZIPSTATE function recognizes APO and FPO ZIP codes. These military ZIP codes correspond to their exit bases in the United States. The ZIP codes are contained in the Sashelp.Zipmil data set. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=sashelp.zipmil;
run;
```

Output from the CONTENTS procedure provides the date of the last update and the contents of the Sashelp.Zipmil data set.

Note: You can download the latest version of the Sashelp.Zipmil data set from the Technical Support website. Select **Zipcode Data Set** from the Name column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set.

Determining When the State Postal Code Table Was Last Updated

Except for APO and FPO addresses, the Sashelp.Zipcode data set contains postal code information that is used with ZIPSTATE and other ZIP code functions. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=sashelp.zipcode;
run;
```

Output from the CONTENTS procedure provides the date of the last update and the contents of the Sashelp.Zipcode data set.

Note: You can download the latest version of the Sashelp.Zipcode data set from the Technical Support website. Select **Zipcode Data Set** from the Name column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the mixed-case name of the city and the two-character state postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed-case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or worldwide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Example: ZIPSTATE Function

This example converts the specified ZIP codes to the corresponding states.

```
data one;
   input zips $10.;
   datalines;
27511
```

```
30306
94110
96801
;

data new;
   set one;
   state=zipstate(zips);
   put state zips;
run;
```

These statements produce these results:

```
NC 27511
GA 30306
CA 94110
HI 96801
```

See Also

Functions:

- “ZIPCITY Function” on page 1580
- “ZIPFIPS Function” on page 1584
- “ZIPNAME Function” on page 1586
- “ZIPNAMEL Function” on page 1588

PART 3

Appendixes

Appendix 1
Tables of Perl Regular Expression (PRX) Metacharacters **1597**

Appendix 2
References **1609**

1596

Appendix 1

Tables of Perl Regular Expression (PRX) Metacharacters

Tables of Perl Regular Expression (PRX) Metacharacters **1597**
General Constructs 1597
Basic Perl Metacharacters 1598
Metacharacters and Replacement Strings 1600
Other Quantifiers 1600
Greedy and Lazy Repetition Factors 1601
Class Groupings 1603
Look-Ahead and Look-Behind Behavior 1605
Comments and Inline Modifiers 1605
Selecting the Best Condition By Using Combining Operators 1606

Tables of Perl Regular Expression (PRX) Metacharacters

General Constructs

Table A2.1 *General Constructs*

Metacharacter	Description
( )	indicates grouping.
non-metacharacter	matches a character.

Metacharacter	Description
{ } [ ] ( ) ^ $. | * + ? \ @	to match these characters, override (escape) with \.
\	overrides the next metacharacter.
\n	matches capture buffer *n*.
(?:...)	specifies a non-capturing group.

Basic Perl Metacharacters

The following table lists the metacharacters that you can use to match patterns in Perl regular expressions.

Table A2.2 *Basic Perl Metacharacters and Their Descriptions*

Metacharacter	Description
\a	matches an alarm (bell) character.
\A	matches a character only at the beginning of a string.
\b	matches a word boundary (the position between a word and a space): ■ "er\b" matches the "er" in "never" ■ "er\b" does not match the "er" in "verb"
\B	matches a non-word boundary: ■ "er\B" matches the "er" in "verb" ■ "er\B" does not match the "er" in "never"
\cA-\cZ	matches a control character. For example, \cX matches the control character control-X.
\d	matches a digit character that is equivalent to [0–9].
\D	matches any character that is not a digit.
\e	matches an escape character.
\E	specifies the end of case modification.

Metacharacter	Description
\f	matches a form feed character.
\l	specifies that the next character is lowercase.
\L	specifies that the next string of characters, up to the \E metacharacter, is lowercase.
\n	matches a newline character.
\num $num	matches capture buffer *num*, where *num* is a positive integer. Perl variable syntax ($num) is valid when referring to capture buffers, but not in other cases.
\Q	escapes (places a backslash before) all non-word characters.
\r	matches a return character.
\s	matches any whitespace character, including space, tab, form feed, and so on, and is equivalent to [\f\n\r\t\v].
\S	matches any character that is not a whitespace character [^ \f\n\r\t\v].
\t	matches a tab character.
\u	specifies that the next character is uppercase.
\U	specifies that the next string of characters, up to the \E metacharacter, is uppercase.
\v or \x0B	vertical white space.
\w	matches any word character or alphanumeric character, including the underscore.
\W	matches any non-word character or non-alphanumeric character, and excludes the underscore.
ddd	matches the octal character *ddd*.
\x*{...}*	matches up to 4 hexadecimal characters **Note:** matches a max of 4 non-zero-leading HEX values in the brace. For example, \x{000000042} is recognized as Unicode character 0x0042; \x{0030c6} is recognized as Unicode 0x30C6.. \x{0030c630b9} is recognized as invalid \x definition. Any invalid definition of \x{...} is processed as

Metacharacter	Description

	a normal character string instead of \x metacharacter. For example:
	`data null;`
	`in="ABC";`
	`out = prxchange("s/\x{0042}/\x{0043}/", -1, in);`
	`put out=;`
	`run;`
	`out=ACC`

\x*dd*	matches the hexadecimal character *dd*.
\z	matches a character only at the end of a string.
\Z	matches a character only at the end of a string or before newline at the end of a string.

Metacharacters and Replacement Strings

You can use the following metacharacters in both a regular expression and in replacement text, when you use a substitution regular expression:

- \l
- \u
- \L
- \E
- \U
- \Q

These metacharacters are useful in replacement text for controlling the case of capture buffers that are used within replacement text. For an example of how these metacharacters can be used, see “Replacing Text” on page 57

For a description of these metacharacters, see “Basic Perl Metacharacters” .

Other Quantifiers

The following table lists other quantifiers that you can use in Perl regular expressions. The descriptions of the metacharacters in the table include examples of how the metacharacters can be used.

Note: Quantifiers are used when a portion of a pattern needs to match a certain number (or numbers) of times. If there is not a quantifier the number of times to match is one.The maximun value for a quantifier is 2147483646 (which is 2^31) -1).

Table A2.3 *Other Quantifiers*

Metacharacter	Description
\	marks the next character as either a special character, a literal, a back reference, or an octal escape: ■ "\n" matches a newline character ■ "\\" matches "\" ■ "\(" matches"("
|	specifies the *or* condition when you compare alphanumeric strings. For example, the construct x|y matches either x or y: ■ "z|food" matches either "z" or "food" ■ "(z|f)ood" matches "zood" or "food"
^	matches the position at the beginning of the input string.
$	matches the position at the end of the input string.
period (.)	matches any single character except newline. To match any character including newline, use a pattern such as "[.\n]".
(pattern)	specifies grouping. Matches a pattern and creates a capture buffer for the match. To retrieve the position and length of the match that is captured, use CALL PRXPOSN. To retrieve the value of the capture buffer, use the PRXPOSN function. To match parentheses characters, use "\(" or "\)".

Greedy and Lazy Repetition Factors

Perl regular expressions support "greedy" repetition factors and "lazy" repetition factors. A repetition factor is considered greedy when the repetition factor matches a string as many times as it can when using a specific starting location. A repetition factor is considered lazy when it matches a string the minimum number of times that is needed to satisfy the match. To designate a repetition factor as lazy, add a ? to the end of the repetition factor. By default, repetition factors are considered greedy.

The following table lists the greedy repetition factors. The descriptions of the repetition factors in the table include examples of how they can be used.

Table A2.4 *Greedy Repetition Factors*

Metacharacter	Description
*	matches the preceding subexpression zero or more times: ■ zo* matches "z" and "zoo" ■ * is equivalent to {0,}
+	matches the preceding subexpression one or more times: ■ "zo+" matches "zo" and "zoo" ■ "zo+" does not match "z" ■ + is equivalent to {1,}
?	matches the preceding subexpression zero or one time: ■ "do(es)?" matches the "do" in "do" or "does" ■ ? is equivalent to {0,1}
{n}	matches *n* times.
{n,}	matches a pattern at least *n* times.
{n,m}	m and n are nonnegative integers, where n<=m. They match at least *n* and at most *m* times: ■ "o{1,3}" matches the first three o's in "fooooood" ■ "o{0,1}" is equivalent to "o?" You cannot put a space between the comma and the numbers.

The following table lists the lazy repetition metacharacters.

Table A2.5 *Lazy Repetition Factors*

Metacharacter	Description
*?	matches a pattern zero or more times.
+?	matches a pattern one or more times.
??	matches a pattern zero or one time.
{n}?	matches exactly *n* times.
{n,}?	matches a pattern at least *n* times.

Metacharacter	Description
{n,m}?	matches a pattern at least *n* times but not more than *m* times.

Class Groupings

The following table lists character class groupings. You specify these classes by enclosing characters inside brackets. These metacharacters share a set of common properties. To be successful, the character class must always match a character. The negated character class must always match a character that is not in the list of characters that are designated inside the brackets. The descriptions of the metacharacters in the table include examples of how the metacharacters can be used.

Table A2.6 *Character Class Groupings*

Metacharacter	Description
[...]	specifies a character set that matches any one of the enclosed characters: ■ "[abc]" matches the "a" in "plain"
[^...]	specifies a negative character set that matches any character that is not enclosed: ■ "[^abc]" matches the "p" in "plain"
[a-z]	specifies a range of characters that matches any character in the range: ■ "[a-z]" matches any lowercase alphabetic character in the range "a" through "z"
[^a-z]	specifies a range of characters that does not match any character in the range: ■ "[^a-z]" matches any character that is not in the range "a" through "z"
[[:alpha:]]	matches an alphabetic character.
[[:^alpha:]]	matches a nonalphabetic character.
[[:alnum:]]	matches an alphanumeric character.
[[:^alnum:]]	matches a non-alphanumeric character.
[[:ascii:]]	matches an ASCII character. Equivalent to [\0–\177].

Metacharacter	Description
[[:^ascii:]]	matches a non-ASCII character. Equivalent to [^\0–\177].
[[:blank:]]	matches a blank character.
[[:^blank:]]	matches a non-blank character.
[[:cntrl:]]	matches a control character.
[[:^cntrl:]]	matches a character that is not a control character.
[[:digit:]]	matches a digit. Equivalent to \d.
[[:^digit:]]	matches a non-digit character. Equivalent to \D.
[[:graph:]]	is a visible character, excluding the space character. Equivalent to [[:alnum:][:punct:]].
[[:^graph:]]	is not a visible character. Equivalent to [^[:alnum:][:punct:]].
[[:lower:]]	matches lowercase characters.
[[:^lower:]]	does not match lowercase characters.
[[:print:]]	matches printable characters.
[[:^print:]]	does not match printable characters.
[[:punct:]]	matches a punctuation character or a visible character that is not a space or alphanumeric.
[[:^punct:]]	does not match a punctuation character or a visible character that is not a space or alphanumeric.
[[:space:]]	matches a space. Equivalent to \s.
[[:^space:]]	does not match a space. Equivalent to \S.
[[:upper:]]	matches uppercase characters.
[[:^upper:]]	does not match uppercase characters.
[[:word:]]	matches a word. Equivalent to \w.
[[:^word:]]	does not match a word. Equivalent to \W.
[[:xdigit:]]	matches a hexadecimal character.
[[:^xdigit:]]	does not match a hexadecimal character.

Look-Ahead and Look-Behind Behavior

Look-ahead and look-behind are ways to look ahead or behind a match to see whether a particular text occurs. The text that is found with look-ahead or look-behind is not included in the match that is found. For example, if you want to find names that end with “Jr.”, but you do not want “Jr.” to be part of the match, you could use the regular expression /.*(?=Jr\.)/. For the value "John Wainright Jr.", the regular expression finds "John Wainright" as a match because it is followed by "Jr."

Table A2.7 *Look-Ahead and Look-Behind Behavior*

Metacharacter	Description
(?=...)	specifies a zero-width, positive, look-ahead assertion. For example, in the expression *regex1 (?=regex2)*, a match is found if both *regex1* and *regex2* match. *regex2* is not included in the final match.
(?!...)	specifies a zero-width, negative, look-ahead assertion. For example, in the expression *regex1 (?!regex2)*, a match is found if *regex1* matches and *regex2* does not match. *regex2* is not included in the final match.
(?<=...)	specifies a zero-width, positive, look-behind assertion. For example, in the expression *(?<=regex1) regex2*, a match is found if both *regex1* and *regex2* match. *regex1* is not included in the final match. Works with fixed-width look-behind only.
(?<!...)	specifies a zero-width, negative, look-behind assertion. Works with fixed-width look-behind only.

Comments and Inline Modifiers

The metacharacters in this table contain a question mark as the first element inside the parentheses. The characters after the question mark indicate the extension.

Table A2.8 *Comments and Inline Modifiers*

Metacharacter	Description
(?#text)	specifies a comment in which the text is ignored.

Metacharacter	Description
(?imsx)	specifies one or more embedded pattern-matching modifiers. If the pattern is case insensitive, you can use (?i) at the front of the pattern. An example is `$pattern="(?i)foobar";`. Letters that appear after a hyphen (-) turn the modifiers off.

Selecting the Best Condition By Using Combining Operators

The elementary regular expressions (for example, \a and \w) that are described in the preceding tables can match at most one substring at the given position in the input string. However, operators that perform combining in typical regular expressions combine elementary metacharacters to create more complex patterns. In an ambiguous situation, these operators can determine the best match or the worst match. The match that is the best is always chosen.

Table A2.9 *Best Match Using Combining Operators*

Metacharacter	Description
ST	in the following example, specifies that AB and A′B′, and A and A′ are substrings that can be matched by S, and that B and B′ are substrings that can be matched by T: ■ If A is a better match for S than A′, then AB is a better match than A′B′. ■ If A and A′ coincide, then AB is a better match than AB′ if B is a better match for T than B′.
S|T	specifies that when S can match, it is a better match than when only T can match. The ordering of two matches for S is the same as for S. Similarly, the ordering of two matches for T is the same as for T.
S{repeat-count}	matches as SSS . . . S (repeated as many times as necessary).
S{min,max}	matches as S{max}|S{max-1}| . . . |S{min+1}|S{min}.
S{min,max}?	matches as S{min}|S{min+1}| . . . |S{max-1}|S{max}.
S?, S*, S+	same as S{0,1}, S{0, big-number}, S{1,big-number}, respectively.

Metacharacter	Description
S??, S*?, S+	same as S{0,1}?, S{0, big-number}?, S{1,big-number}?, respectively.
(?=S), (?<=S)	considers the best match for S. (This is important only if S has capturing parentheses, and back references are used elsewhere in the whole regular expression.)
(?!S), (?<!S)	unnecessary to describe the ordering for this grouping operator because only whether S can match is important.

References

References

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Reading, USA: Addison-Wesley Publishing Co..

Abramowitz, Milton, and Irene Stegun. 1964. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables – National Bureau of Standards Applied Mathematics Series #55*. Washington, USA: U.S. Government Printing Office.

Amos, D. E., S. L. Daniel, and K. Weston. 1977. "CDC 6600 Subroutines IBESS and JBESS for Bessel Functions I(v,x) and J(v,x), x ≥ 0, v ≥ 0." *ACM Transactions on Mathematical Software* 3: 76–95.

Cheng, R. C. H. 1977. "The Generation of Gamma Variables." *Applied Statistics* 26: 71–75.

Duncan, D. B. 1955. "Multiple Range and Multiple F Tests." *Biometrics* 11: 1–42.

Dunnett, C. W. 1976. "A Multiple Comparisons Procedure for Comparing Several Treatments with a Control." *Journal of the American Statistical Association* 50: 1096–1121.

Fishman, G. S. 1976. "Sampling from the Poisson Distribution on a Computer." *Computing* 17: 145–156.

Fishman, G. S. 1978. *Principles of Discrete Event Simulation*. New York, USA: John Wiley & Sons, Inc.

Fishman, G. S., and L. R. Moore. 1982. "A Statistical Evaluation of Multiplicative Congruential Generators with Modulus (2^{31} – 1)." *Journal of the American Statistical Association* 77: 1, 29–136.

Higham, Nicholas J. 2002. *Accuracy and Stability of Numerical Algorithms*. 2d ed. Philadelphia, USA: Society of Industrial and Applied Mathematics.

Hochberg, Y., and A. C. Tamhane. 1987. *Multiple Comparison Procedures*. New York, USA: John Wiley & Sons, Inc.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 3. Sorting and Searching*. Reading, USA: Addison-Wesley.

L'Ecuyer, P. and Simard, R. 2007. "TestU01: A C Library for Empirical Testing of Random Number Generators.." *ACM Transactions on Mathematical Software 33, 4, Article 22* : 22.

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.

Matsumoto, M., and T. Nishimura 2002. “Mersenne Twister with Improved Initialization.” *ACM Transactions on Modeling and Computer Simulation 8* : xx.

Matsumoto, M. “Mersenne Twister 64bit version.” 2005. Available Mersenne Twister 64bit version.

O’Neill, Melissa E. “PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.” 2015. Available PCG.

François Panneton, Pierre L’Ecuyer, and Makoto Matsumoto 2006. “Improved Long-period Generators Based on Linear Recurrences Modulo 2.” *ACM Trans. Math. Software* : 22.

John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw 2011. “Parallel Random Numbers: As Easy as 1, 2, 3.” *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)* : 22.

Vigna, Sebastiano “XOR/rotate/shift/rotate.” 2015. Available XOR/rotate/shift/rotate.

Vincenty, T. 1975. “Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations.” *Direct and Inverse Solutions* 22: 88–93.

Williams, D. A. 1971. “A Test for Differences between Treatment Means When Several Dose Levels Are Compared with a Zero Dose Control.” *Biometrics* 27: 103–117.

Williams, D. A. 1972. “The Comparison of Several Dose Levels with a Zero Dose Control.” *Biometrics* 28: 519–531.