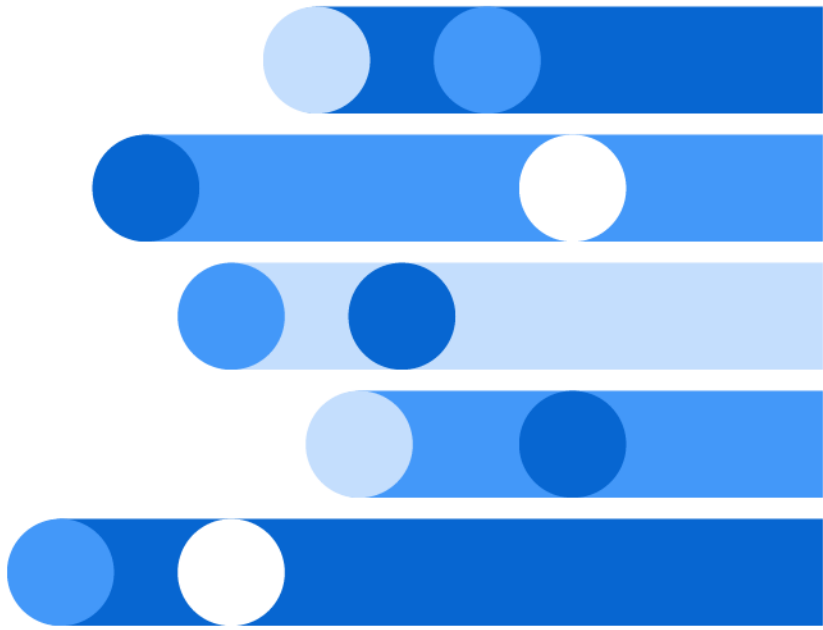
SAS® Cloud Analytic Services: CASL Reference

2026.08*



* This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.

SAS® Documentation
August 18, 2026

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2025. *SAS® Cloud Analytic Services: CASL Reference*. Cary, NC: SAS Institute Inc.

SAS® Cloud Analytic Services: CASL Reference

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

August 2026

v_002-P1:proccas

Contents

1

Introduction to CASL Programming

About the CAS Language

CASL is a language specification used by the SAS client and other clients to interact with SAS Cloud Analytic Services (CAS).

Here are characteristics of the CAS language (CASL):

- CASL is a statement-based language.
- CASL is a dynamically typed language.[1]
- The language is case insensitive.
- CASL is a scripting language with the following strengths:
  - running actions
  - working with results
  - developing analytic pipelines
  - running code in CAS with user-defined actions
- Statements can include keywords such as the names of actions and functions.
- Statements can include expressions.
- Statements are terminated with a semicolon (;).

1. Variable data types are assigned by SAS at runtime. You do not have to declare variables with a specific type before using them.

- A single PROC CAS step can contain several CASL programs.

Here are some uses for CASL:

- develop analytic pipelines
- use actions to submit requests to the CAS server to do work and then return the results
- evaluate and manipulate the results returned by an action
- create the arguments to an action
- create user-defined actions and functions

Running CASL Programs

To use the CASL language with SAS, you need the following:

- A CAS session. The CAS statement connects to an existing session on the server or starts a new session on the server. For more information about sessions, see "Sessions" in *SAS Cloud Analytic Services: Fundamentals*.

  TIP If you are not submitting CAS actions, then you do not need a CAS session.

- The CAS procedure. PROC CAS enables SAS to interpret the CASL programming statements that interact with the server.

You can submit CASL programs in the following ways:

- Through the SAS Windowing environment or SAS Studio using the CAS procedure.
- On the CAS server using server-side processing with user-defined actions. This enables you to run CASL programs from SAS, Python, Lua, or R. For more information about server-side processing, see "Writing User-Defined Actions" in *SAS Cloud Analytic Services: CASL Programmer's Guide*.

Categorized List of CAS Procedure Statements and Examples

Here is a categorized list of SAS statements related to the CAS Procedure grouped by their primary functionality:

Table 1.1 *Control Flow and Conditional Execution Statements*

Statement	Description	Example Reference
IF-ELSE and Branching		
IF on page 70	Conditionally executes a statement based on the value of an expression	Compare grades and display messages based on the results on page 71
ELSE on page 50	If the expression specified by an IF statement results in a false value, then the statement immediately after the ELSE statement is executed	Execute an alternative block when the condition is false on page 51
GOTO on page 68	Transfers execution immediately to a labeled statement	Exit a loop early based on a condition on page 69
LEAVE on page 73	Stops processing the current loop and resumes with the next statement in the sequence	Loop through values and exit iterating DO loop prematurely on page 74
CONTINUE on page 30	Stops processing the current DO loop iteration and resumes with the next iteration	Skip the PUT statement inside the loop on page 30
EXIT on page 55	Exits the current block of CASL code	
Looping and Iteration		
DO on page 38	Creates a block of code that executes as one statement	Group multiple statements within conditional blocks on page 39
DO: Iterative on page 39	Executes statements between the DO and END statements repetitively,	Loop through a range of values with specified conditions on page 42

Statement	Description	Example Reference
	based on the value of an index variable	
DO OVER on page 43	Iterates over an array, dictionary, or table	Iterate over elements in an array or list on page 44
DO UNTIL on page 46	Executes statements in a DO loop repetitively until a condition is true	Repeat a loop until a condition is met on page 47
DO WHILE on page 48	Executes statements in a DO loop repetitively while a condition is true	Repeat a loop while a condition remains true on page 48

Table 1.2 *Data Manipulation and Processing Statements*

Statement	Description	Example Reference
Data Input and Output		
OUTPUT on page 77	Sets the active output location	Display log information or results on page 78
		Generate and store results from a procedure on page 78
PRINT on page 79	Writes the values of constants, variables, and expressions to the current output location	Print one value on page 80
		Print multiple values on page 80
		Manage spacing of string variables on page 81
		Add message levels on page 81
PRINTLST on page 82	Prints to the listing file defined by the application. If the value is a list, the list will be transverse and each item will be displayed. If the item is a table, it will be printed to the current output location	Display a list of values on page 82

Statement	Description	Example Reference
FILE on page 59	The FILE statement changes the default output location for the PRINT statement. You can choose to write the output to an external file, the log, or ODS	Change output locations on page 60
Data Modification and Formatting		
FORMAT on page 62	Associates a format with a single variable or a list of variables	Apply a format on page 63
SET on page 90	Controls options that modify some behaviors of CAS actions and controls corresponding messages from the SAS log	Display the disposition in the SAS Log on page 93
		Print a warning message for duplicate keys to the SAS Log on page 93
		Print diagnostic messages to the SAS Log on page 94
Data Assignment and Handling		
Assignment on page 27	Evaluates an expression and stores the result in a variable	Define target variables and assign a result of a function to a variable on page 28
		Define a target variable with a value on page 28
DELETE on page 34	Deletes the specified value from memory	Delete different data values on page 35
UNSET on page 97	Turns off options that modify some behaviors of CAS actions and suppresses corresponding messages in the SAS log	Suppress the printing of the disposition in the SAS Log on page 99

Table 1.3 *CAS Session and Execution Control Statements*

Statement	Description	Example Reference
Execution Control		

Statement	Description	Example Reference
RUN on page 84	Executes the previously entered SAS statements	
RETURN on page 82	Returns a value from the current function	Create two functions and return values on page 83
Session and Scope Management		
SESSION on page 89	Specifies a session to use for actions invoked by the program	Print the session and the session UUID on page 89
		Connect to an existing session on page 89
LOCAL on page 75	Creates a variable that is local to a CASL function	Create local and global variables on page 76
GLOBAL on page 67	Creates a variable that can be accessed and used anywhere within a CASL program	Create global variables on page 68

Table 1.4 *Action and Function Execution Statements*

Statement	Description	Example Reference
Action Execution		
ACTION on page 25	Runs SAS Cloud Analytic Services actions	Load a table and view table information using Table action set on page 26
LOADACTIONSET on page 74	Loads a SAS Cloud Analytic Services action set	Load an action set on page 75
SAVERESULT on page 84	Saves a result table in the variable in the form specified	Save a result table generated from actions on page 87
Function Execution		
FUNCTION on page 63	Creates a new function that can be called in an expression	Create a user-defined function on page 64

Statement	Description	Example Reference
FUNCTIONLIST on page 66	Lists all CASL built-in functions and user-defined functions by name with simple descriptions	List information about the dictionary function on page 66
CALL on page 29	Calls a function with the specified argument. If the function returns a value, it is ignored	Execute a function that sorts a list of values on page 29
FNC on page 61	Lists all common functions by name and category with simple descriptions	Display all common functions and their categories on page 61

Table 1.5 *External Code and Scripting Statements*

Statement	Description	Example Reference
External Code Execution		
EXTERNALSOURCE on page 55	Enables you to embed text in the program as a block of code and then assign the code to a given variable	Save IML code in a variable and execute the code on page 57
		Use the EXTERNALSOURCE statement with the EXECUTE function on page 58
ENDEXTERNALSOURCE on page 52	Marks the end of the EXTERNALSOURCE statement	Save IML code in a variable and execute the code on page 57
Scripting Control		
SOURCE on page 95	Enables you to embed text in the program and assign it to a given variable	Embed text in a program and display it in the SAS Log on page 96
ENDSOURCE on page 54	Marks the end of the SOURCE statement	Mark the end of the SOURCE statement on page 54

Categorized List of CASL Built-in Functions and Examples

Here is a categorized list of CASL built-in functions grouped by functionality.

- Session and execution control functions on page 8 manage CAS session execution, including running actions, handling parallel sessions, and controlling execution flow.
- Data manipulation and handling functions on page 9 manage, manipulate, and process tables and their attributes.
- Encoding, decoding, and data transformation functions on page 11 perform encoding, decoding, and data format transformations.
- Environment and system functions on page 12 retrieve system-related information and manage environment variables.
- Dictionary, symbol, and variable handling functions on page 13 help define and manipulate variables, symbols, and stored values.
- CASL store and storage functions on page 14 manage CASL store objects, which can persist variables and results.
- File path and handling functions on page 15 allow reading files and retrieving path information.
- Result handling and retrieval functions on page 15 help manage and retrieve results from CAS actions.
- General utility functions on page 16 serve as general-purpose utilities within CASL.

Table 1.6 *Session and Execution Control Functions*

Function	Description	Example Reference
Managing Action Execution		
CANCELACTION on page 131	Cancels the action running on the given session.	Cancel the action running on given session on page 132
CANCELACTIONS on page 132	Cancels the action running on the given list of sessions.	Cancel the action running on the given sessions on page 133

Function	Description	Example Reference
EXECUTE on page 158	Compiles and executes the given code.	Run dynamically generated code on page 159
WAIT_FOR_NEXT_ACTION on page 224	Wait for a completed action.	Pause execution until the next action is available on page 224
		Pause execution until the next action completes on page 224
Parallel Session Management		
CREATE_PARALLEL_SESSION on page 152	Starts a session with the same identity as the calling action. You can call this function to create multiple sessions.	Create parallel sessions with all workers on page 152
		Create parallel sessions with n-workers on page 153
LIST_PARALLEL_SESSIONS on page 152	Lists parallel sessions.	List parallel sessions on page 179
TERM_PARALLEL_SESSION on page 214	Terminates a parallel session.	End a parallel session on page 214
General Execution Control		
EXIT on page 161	Exits the block of CASL code with the given status.	
SLEEP on page 190	Sleep in the operating system (OS) for the number of seconds specified.	Specify n-seconds the OS sleeps on page 190

Table 1.7 Data Manipulation and Handling Functions

Function	Description	Example Reference
Table Attribute and Row Management		

Function	Description	Example Reference
ADD_TABLE_ATTR on page 124	Adds attributes to a result table.	Add a description of data to table attributes of the result table on page 125
ADDBYGROUP on page 125	Creates a new table from a BY-group table. The BY variables are added as the first variables of each row. The attributes for BY-group processing are removed.	Create a new table from a BY-group table on page 126
ADDROW on page 127	Adds one or more rows to a table.	Add three rows to a result table on page 128
		Create a new table and add multiple rows on page 128
ADDUNIQUE on page 129	Adds a value to an array, if the value does not already exist within the array.	Add a value to an array on page 130
Table Processing and Lookup		
COMBINE_TABLES on page 142	Create a new table that has the name of the first table and contains all of the rows from all the tables. If this is a BY-group table, add the BY-group variables as the first variables of the table.	Create a new table that contains all rows from all tables on page 143
		Create a new table that contains only the rows from the tables with keys matching a pattern on page 147
		Combine multiple ByGroupSet tables on page 149
FINDTABLE on page 162	Searches the given variable for the first table it sees. This is useful when a result from an action returns a copy of a result table.	Locate and extract table data from a result object on page 162
GETCOLUMN on page 165	Extracts the column into an array.	Extract a column from a table on page 165

Function	Description	Example Reference
NEWTABLE on page 181	Creates a new table.	Create a new table that contains three rows and three columns on page 182
SORT on page 191	Returns a list sorted in ascending order.	Sort a list in ascending order on page 191
		Sort a keyed list in ascending order on page 192
SORT REV on page 192	Returns a list sorted in descending order.	Sort a list in descending order on page 193
		Sort a keyed list in descending order on page 193
Table Metadata Handling		
TABCOLUMNS on page 211	Returns the names and labels of columns from a result table into a dictionary. For each item in the dictionary, the key is the name of a column, and the value is the label of the column.	Return the names and labels of columns from a result table into a dictionary on page 211
TABTYPES on page 212	Extracts the data types from a result table into a dictionary.	Retrieve the data types from a result table on page 212

Table 1.8 *Encoding, Decoding, and Data Transformation Functions*

Function	Description	Example Reference
Base 64 Encoding/Decoding		
base64Decode on page 130	Decodes a string, integer, double array, or a binary object to Base64.	Decode the supplied base64-encoded string on page 130
base64Encode on page 131	Encodes a string, integer, double array, or a binary object to Base64.	Encode a string to Base64 on page 131

Function	Description	Example Reference
JSON and Data Conversion		
CASL2JSON on page 133	Converts the contents of a CASL dictionary to a string containing JSON formatted text.	Convert fetch action results to JSON on page 135
		Convert summary action results to JSON on page 136
		Convert Freq action results to JSON on page 137
JSON2CASL on page 175	Converts a string containing JSON formatted text to a CASL dictionary.	Convert a JSON file to a CASL dictionary on page 177
unpackData on page 219	Unpacks the string, packed by base64Encode, according to the given format.	Unpack the string packed by base64Encode on page 220

Table 1.9 *Environment and System Functions*

Function	Description	Example Reference
System and Environment Queries		
ENCODING on page 157	Returns an encoding string for the current session encoding.	Return the session encoding string on page 158
GETENV on page 166	Returns the environment variable for the given string.	Get the host name for the server running the Compute Session on page 167
MEMORYSTATS on page 180	Displays CPU and memory statistics.	Display CPU and memory statistics on page 181
TIMESTAMP on page 214	Returns the current date and time in a specified format.	Get the current date and time on page 217
		Format the results of the TIMESTAMP function on page 218

Function	Description	Example Reference
UUID on page 222	Returns the universally unique identifier (UUID) for the given session.	List the actions in a session's action queue on page 223

Table 1.10 *Dictionary, Symbol, and Variable Handling Functions*

Function	Description	Example Reference
Dictionary and Symbol Management		
DICTIONARY on page 155	Searches for a key in the given dictionary and, if found, returns its associated value.	Obtain the value associated with a key in a dictionary on page 155
SYMDEL on page 194	Deletes the specified macro variable.	Delete a macro variable from the named repository in CAS on page 194
		Delete a macro variable from the Compute Server symbol table on page 196
SYMGET on page 197	Returns the value of a macro variable.	Retrieve a macro variable from the named repository in CAS on page 198
		Retrieve a macro variable from the Compute Server symbol table on page 198
SYMPUT on page 201	Stores a value in a SAS macro variable.	Assign a value to a macro variable in CAS on page 202
		Assign a value to a macro variable on the Compute Server on page 204
SYMPUTX on page 205	Assigns a value to a macro variable, and removes both leading and trailing blanks.	Assign a value to a macro variable and remove leading and trailing blanks on page 207
		Create macro variables on the SAS Compute Server on page 209

Function	Description	Example Reference
Checking and Formatting Variables		
DIM on page 156	Retrieves the dimensions of an array or dictionary.	Retrieve the dimensions of a dictionary on page 157
FMTVAR on page 163	Creates a CASL format variable that can be used as a substitute for a SAS format.	Create a format variable on page 164
INPUT_FORMAT on page 171	Converts the string to the best numeric value.	Convert a string on page 172
isType on page 172	Returns TRUE or FALSE based on whether an expression results in a value of a specified type.	Determine whether a literal is a double on page 173
		Specify various types of expressions on page 174
LOC on page 179	Returns the row number in which the given value is found in the given column.	Retrieve a row number for a specified value on page 180

Table 1.11 *CASL Store and Storage Functions*

Function	Description	Example Reference
CASL Store Management		
CASLSTORE on page 138	Creates a caslstore value.	Load and access stored user-defined functions on page 139
DEFAULT_CASLSTORE on page 153	Sets the default caslstore.	Specify caslstore priority on page 153
UPLOAD_CASLSTORE on page 220	Uploads CASL functions from a specified CASL source code string into the specified code storage repository.	Save user-defined functions to a specified CAS library on page 221

Table 1.12 File and Path Handling Functions

Function	Description	Example Reference
File and Path Processing		
READFILE on page 184	Reads and returns the contents of the specified file reference.	Read the contents of a .sas file on page 185
READPATH on page 185	Reads and returns the contents of the specified file name.	Read the contents of a .sas file into a variable on page 186

Table 1.13 Result Handling and Retrieval Functions

Function	Description	Example Reference
Retrieving and Formatting Results		
CODETOSTATUS on page 141	Converts the action status into a dictionary.	Retrieve an action status and put the status in a dictionary on page 141
GETKEYS on page 167	Extracts the keys from the dictionary into an array.	Extract the keys from a dictionary into an array on page 168
GETRESULT on page 168	Retrieves results associated with ID or tag.	Retrieve results from an asynchronous session on page 169
GETVALUES on page 170	Extracts the values from the dictionary into an array.	Extract dictionary values into an array on page 171
RESULT_BY_COL on page 186	Creates a new table with the given columns.	Create new tables with the given columns from the result table on page 186
RESULT_BY_TYPE on page 187	Creates a new table with columns that match the type specifications.	Create new tables that include the given columns from the result table on page 188
SEND_RESPONSE on page 189	Sends the specified result back to the client.	Send the resulting objects on the server back to the client on page 190

Table 1.14 General Utility Functions

Function	Description	Example Reference
General Purpose		
CLEAR on page 140	Clears all specified variables. If no variables are specified, all variables are cleared.	Clear the specified variables on page 140
EXISTS on page 159	Determines whether a variable exists or whether a dictionary contains a specified key.	Check for an existing variable on page 160
		Check for an existing key on page 160
PUT on page 183	Formats a value with the given format and returns it as a string.	Format a value on page 184

2

CAS Procedure

Overview: CAS Procedure

About the CAS Procedure

The CAS procedure enables you to interact with SAS Cloud Analytic Services (CAS) from the SAS client by providing you a programming environment based on the CASL language specification. The programming environment enables you to run CAS actions and use the results to prepare the parameters for another action.

The CASL language is dynamically typed. The types of variables do not need to be declared prior to use. A variable can hold a value of any type. The values are automatically converted between types as necessary, but you can also manually cast from one type to another.

The CAS procedure has several features that enable you to perform the following operations:

- Run any CAS action that is supported by the server.
- Load new action sets into the server.
- Use multiple sessions to perform asynchronous execution.

- Operate on parameters and results as variables using the function expression parser.

Terminology

action
: A task that is performed by the server at the request of the user. The user sends a request to the server to execute an action. The action has a published API for input arguments. The arguments are sent as a dictionary to the server. The server executes the action, and sends back a dictionary of results along with a status to indicate whether the action has succeeded or not.

action set
: A collection of actions (tasks) that group functionality such as session management, table management, and so on.

argument
: The actual value of the parameter is passed to the function.

condition
: One or more numeric or character expressions that result in a Boolean value when evaluated

expression
: A combination of one or more values, constants, variables, operators, and functions that are evaluated using rules and precedence to compute a result.

function
: A group of statements that performs a specific task and returns a result. A function is accessed by calling the entry point of the function and providing arguments for the specified API for the function.

parameter
: A type of variable that is used in a subroutine to refer to a portion of the data that is provided as input to the subroutine. The actual information that is sent to the subroutine is referred to as an *argument*.

result table
: A transportable memory representation of a CAS table. A result table or group of result tables can be sent as a response to an action, or a result table can be created from scratch from within CASL. A result table contains rows, columns, labels, data types, and attributes.

CAS session
: A session represents a user that has logged on to the server. The session can then be used to execute actions, which produce results.

variable
: A symbolic name for a value that can be used in an expression. The value can be a list, a dictionary, or a simple data types (string, integer, or floating-point number). You can assign a value throughout a program. In CASL there are global, local, and parameter variables.

Overview of CASL Statements

CASL Statements are a series of items that include keywords, identifiers, and operators. A CASL statement can perform the following actions:

- perform variable assignments
- influence program control
- perform input and output of data
- create and call functions
- call actions and process result tables
- specify or change the behavior of a CASL program
- use expressions to build parameters for CASL statements

Syntax: CAS Procedure

Note: If a floating-point exception (FPE) is encountered, CASL prints the following message to the Log: **`CASL reset after signal`** and continues execution.

Tip: For information about the actions that you can run with the CAS procedure, see *SAS Viya Platform Actions and Action Sets by Name and Product*.

PROC CAS;

ACTION < action-set-name.*action-name* >< RESULT=<*variable*> >< STATUS=<*rc*> >< ASYNC=*name* > / <*parameters*>;

<ASSIGNMENT> *target* = *expression*;

CALL function (*argument1*, *argument2*...);

CONTINUE;

DEFAULT RESULT=*variable-expression* STATUS=*variable-expression* | CLEAR <*clear-option-1* <*clear-option-2* <...*clear-option-N*>>>;

DELETE *value*;

DESCRIBE *variable*;

DO;

... *more CASL statements* ...;

END;

DO UNTIL

condition;

... *more CASL statements* ...;

```
    END;
DO WHILE condition;
    ... more CASL statements ...;
    END;
DO [<key,>] <var> OVER <value>;
    ... more CASL statements ...;
    END;
DO <var> = <start> [TO <stop>] [BY <increment>]
    [WHILE <condition> | UNTIL <condition>];
    ... more CASL statements ...;
    END;
ENDEXTERNALSOURCE;
ENDSOURCE;
EXIT expression;
EXTERNALSOURCE variable-expression;
  <embedded-text>
  ENDEXTERNALSOURCE;
FILE fileref;
FNC <category-name>;
FORMAT variable-1...variable-N format;
FUNCTION function name ( [argument 1 [, argument 2...] ] );
    ... more CASL statements ...;
    END;
FUNCTIONLIST <name>;
    FLIST <name>;
GLOBAL variable;
GOTO label;
IF condition
    THEN statement;
    ELSE < statement>;
Label statement
LOCAL variable;
LEAVE;
LOADACTIONSET actionsetname;
NULL;
OUTPUT <ODS | LOG>;
PRINT value-1...value-N;
PRINTLIST expression;
RETURN <expression | value>;
SAVERESULT variable-name <NOREPLACE>
  <DATAOUT=<libref.> data-set-name> | <LIB=libref> |
```

<FILE=<*pathname*> | <*filename*>>
<<CASLIB=*casref*> | <CASOUT=*name*>>;

SELECT < (*select-expression*) >;

WHEN-1 < (*when-expression-1* ..., *when-expression-n* >) *statement*;
<...WHEN-n (*when-expression-1* ..., *when-expression-n*) > *statement*;

<OTHERWISE *statement*>;**END**;

SESSION *session-reference*;

SET (DISP <ECHOEXTSRC><LOGS><LOGS=DEBUG | NOTE | TRACE | WARN><STDJSON><WARNINGDUP><USEWIDTH>) ;

SOURCE *variable*; *text*; endsource;

UNSET DISP <ECHOEXTSRC><LOGS><STDJSON><USEWIDTH ><WARNINGDUP>

UPLOAD

PATH=*path-to-file*

<CASOUT={*output-table-options*>}

<IMPORTOPTIONS={FILETYPE= “BASESAS” | “CSV” | “DTA” | “ESP” | “EXCEL” | “FMT” | “EXCEL” | “HDAT” | “JMP” | “LASR” | “SPSS” | “XLSX”, *fileType-specific-parameters*}>;

RUN;

QUIT;

Statement	Task
ACTION	Runs SAS Cloud Analytic Services actions.
Assignment	Evaluates an expression and stores the result in a variable.
CALL	Calls a function with the specified argument. If the function returns a value, it is ignored.
CONTINUE	Stops processing the current DO loop iteration and resumes with the next iteration.
DEFAULT	Enables you to configure how CASL processes certain statements.
DELETE	Deletes the specified value from memory.
DESCRIBE	Enables you to view the structure and data type of CASL variables and expressions.
DO	Creates a block of code that executes as one statement.
DO	Executes statements between the DO and END statements repetitively, based on the value of an index variable.
DO OVER	Iterates over an array, dictionary, or table.

Statement	Task
DO UNTIL	Executes statements in a DO loop repetitively until a condition is true.
DO WHILE	Executes statements in a DO loop repetitively while a condition is true.
END	Ends a DO group, FUNCTION, or SELECT group processing.
EXTERNALSOURCE	Marks the end of the EXTERNALSOURCE statement.
ENDSOURCE	Marks the end of the SOURCE statement.
EXIT	Exits the current block of CASL code.
EXTERNALSOURCE	Enables you to embed text in the program and assign it to a given variable.
FILE	Enables you to specify a different output destination.
FNC	Lists all common functions by name and category with simple descriptions.
FORMAT	Associates a format with a single variable or a list of variables.
FUNCTION	Creates a new function that can be called in an expression.
FUNCTIONLIST	Lists all CASL built-in functions and user-defined functions by name with simple descriptions.
GLOBAL	Declares a variable that has global scope.
GOTO	Transfers execution immediately to a labeled statement.
IF	Conditionally executes a statement based on the value of an expression.
Label	Specifies a label target for the GOTO statement. No semicolon is required.
LEAVE	Stops processing the current loop and resumes with the next statement in the sequence.
LOADACTIONSET	Loads an action set.
LOCAL	Declares a variable that has local scope.
OUTPUT	Sets the active output location.
PROC CAS	Enables you to program and schedule SAS Cloud Analytic Services actions from the SAS client.
PRINT	Writes the values of constants, variables, and expressions to the current output location.

Statement	Task
PRINTLST	Prints to the listing file defined by the application. If the value is a list, the list is transverse and each item is displayed. If the item is a table, it is printed to the current output location.
RETURN	Returns a value from the current function.
RUN	Executes the previously entered SAS statements.
SAVERESULT	Saves a result table in the variable in the form specified.
SELECT	Executes one of several statements or groups of statements.
SESSION	Specifies a session to use for actions invoked by the program.
SET	Controls which messages are written to the SAS log by actions.
SOURCE	Enables you to embed text in the program and assign it to a given variable.
UNSET	Turns off options that modify some behaviors of CAS actions and suppresses corresponding messages in the SAS log.
UPLOAD	Transfers a file from the SAS client to the server. After data transfer, the server loads the data into an in-memory table.

PROC CAS Statement

Enables you to program and schedule SAS Cloud Analytic Services actions from the SAS client.

Tip: For information about the actions that you can run with the CAS procedure, see *SAS Viya Platform Actions and Action Sets by Name and Product*.

Syntax

PROC CAS;

Without Arguments

PROC CAS does not have any required or optional arguments. Once PROC CAS is executed, it continues running until it reaches one of the following:

- a DATA step

- another PROC step
- the QUIT statement
- an error condition that prevents the progress of the procedure.

At that point, you must execute PROC CAS again using the syntax above.

Details

Note: Global statements, SAS macro code, and RUN statements do not terminate the procedure.

ACTION Statement

Runs SAS Cloud Analytic Services actions.

Note: You can use expressions to build up parameters for ACTION statements.

See: For documentation about the actions that you can run with the ACTION statement, see: *SAS Viya Platform: System Programming Guide*, and *SAS Visual Analytics: Programming Guide*.

Example: "Example 2: Run an Action" on page 104

Syntax

<ACTION> <*action-set-name.*>*action-name* < RESULT= <*variable* > < STATUS = <*rc* > < ASYNC = *name* >
<*parameters*>;

Required Argument

<action-set-name.>action-name
specifies the action to run.

action-set-name
specifies the name of the action set that contains the action to run. Examples are **`table`**, **`simple`**, and **`network`**.

action-name
specifies the name of the action to run. Examples are **`loadTable`**, **`listActions`**, and **`serverStatus`**.

Example For example, the following statement executes the loadTable action in the table action set:

```
action table.loadTable / path="cholesterol.csv";
```

```
run;
```

In this example, PATH= is a parameter of the loadTable action.

Optional Arguments

parameter

the specified action's parameters. You can discover the parameters for an action with the help action.

Example
```
action help / action="action-name";
run;
```

RESULT=*variable-name*

stores the results of the action in a variable. You can then use the variable in other CASL statements and actions. You can use the variable with statements such as DESCRIBE and with a selected set of SAS statements within the scope of the CAS procedure invocation.

Alias R=

Interaction If you do not specify the RESULT= argument, actions that return result tables are printed to the default output destination. For actions that return other data types, the results are printed to the SAS log.

See For examples of using results in a program, see "Working with Results" in *SAS Viya Platform: System Programming Guide*.

STATUS=<*variable-name*>

stores the status code of the action in a variable. You can examine the status code to determine if an action completed with or without any errors. If you do not specify a *variable-name*, a variable named **`_status`** that contains the status codes is created.

See For information about status codes, see "Severity, Status, and Reason Codes" in *SAS Viya Platform: System Programming Guide*.

ASYNC=*result name*

enables you to submit multiple requests to the server. If your requests are in the same session, the requests are queued in the order received. If your requests are in separate sessions, the requests are executed in parallel.

Example: Loading a Table and Viewing Table Information Using Tables Action Set

This example uses the table.loadTable action that performs a server-side load.

```
/* change the host and port to match your site */
options cashost="cloud.example.com" casport=5570;

/* start a session, if you do not already have one */
```

```
*cas casauto;

proc cas;
   session casauto;

   /* Load source data (IRIS) into a table. */
   table.loadTable /                  /* #1 */
      path="iris.sashdat"
      casout={name="iris"};
run;

   table.tableInfo   / table="iris";  /* #2 */
   table.tableDetails / table="iris"; /* #3 */
quit;
```

1 Use table.loadTable action to load a table from a caslib's data source. For more information, see loadTable Action.

2 Use table.tableInfo action to view information about the table. For more information, see tableInfo Action.

3 Use the table.tableDetails action to get detailed information about a table. For more information, see tableDetails Action.

Assignment Statement

Evaluates an expression and stores the result in a variable.

Category: Local

Tip: If the variable already exists, the new assignment replaces the variable and the old value is overwritten.

Syntax

variable = *expression*;

Required Arguments

variable
names a new or existing variable.

expression
a combination of one or more values, constants, variables, operators, and functions that are evaluated using rules and precedence to compute a result. For more information about CASL Expressions, see "CASL Expressions" in *SAS Cloud Analytic Services: CASL Programmer's Guide*.

Tip expressions can contain the variable that is used on the left side of the equal sign. When a variable appears on both sides of a statement, the

original value on the right side is used to evaluate the expression, and the result is stored in the variable on the left side of the equal sign.

Details

Assignment statements evaluate the expression on the right side of the equal sign and store the result in the variable that is specified on the left side of the equal sign.

A variable can refer to a value of any data type. On one assignment statement, a value is assigned to a variable and on another assignment statement, a value of a different data type can be assigned to the same variable. If an assignment statement is **a = b = 5**, then the statement is **a = (b = 5)**, where **b = 5** is a comparison expression.

Examples

Example 1: Valid Assignment Statements

```
proc cas;
   Name = 'Jane Smith';          /* 1 */
   y = 88;                       /* 2 */
   z = min(y, 70);               /* 3 */
run;
```

1 This assignment statement defines the target variable as referring to the string value "Jane Smith".

2 This assignment statement defines the target variable with the value of "88".

3 This target statement assigns the result of the MIN function to the variable *z*.

Example 2: Using the Assignment Statement

The following example uses the assignment statement to define the target variable with a value of "1".

```
proc cas;
  x.y[2].z.a["abc"].t[10]=1;
  describe x;                    /* 1 */
run;
```

1 You can use the DESCRIBE statement to view the structure of the data.

The DESCRIBE statement shows the full structure the arrays and dictionaries. *x* is the dictionary which contains an array, *y*, with two entries. As you can see

```
a["abc"]
```

functions the same way as

```
a.abc
```

Both methods create dictionaries. The following output is printed to the SAS Log:

Output 2.1 *Log Output*

```
dictionary ( 1 entries, 1 used);
 [y] array ( 2 entries, 1 used);
  [ 2] dictionary ( 1 entries, 1 used);
   [z] dictionary ( 1 entries, 1 used);
    [a] dictionary ( 1 entries, 1 used);
     [abc] dictionary ( 1 entries, 1 used);
      [t] array ( 10 entries, 1 used);
       [10] int64_t;
```

CALL Statement

Calls a function with the specified argument. If the function returns a value, it is ignored.

Syntax

<CALL> *function* (<*argument-1*, *argument-2*...>);

Optional Arguments

function
the name of the built in or user defined function.

argument
specifies values that are passed to a function when it is called.

Example

This example uses the CALL statement with the CASL built-in SORT function.

```
proc cas;
   call sort({98, 74, 54, 18, 101, 67, 80, 90, 62});
run;
```

See Also

"FUNCTION Statement"

CONTINUE Statement

Stops processing the current DO loop iteration and resumes with the next iteration.

Requirement: The CONTINUE statement can appear only in the statement list of an iterative DO loop (for example, DO *i*=, DO WHILE, or DO UNTIL).

Syntax

CONTINUE;

Without Arguments

The CONTINUE statement has no arguments. It resumes processing statements with the next iterations of the DO loop.

Comparisons

- The CONTINUE statement stops the processing of the current iteration of a DO statement and resumes program execution with the next iteration of the current DO statement.
- The LEAVE statement stops the processing of the current DO statement and resumes program execution outside of the current DO statement.

Example

This example illustrates the use of the CONTINUE statement.

```
proc cas;
  function evaltest( x, y, t);
   if (x = y) then put "GOOD " t;        /* 1 */
    else             put "ERROR " t;
   return( t+1);
  end;
test = 1;                                /* 2 */
  do i = 1 to 10 by 1;
   continue;                             /* 3 */
   put i;
  end;
test = evaltest( i, 11,test);
   put i;
run;
```

1. If *x* equals to *y* then it prints **GOOD** to the log, otherwise it prints **ERROR** and returns the value of **t** and adds one.
2. The DO loop iterates and prints the incremented value of **test**.
3. When **test** is equal to 10, the CONTINUE statement causes execution to jump to the next iteration of the DO loop, and prevents **test** from printing.

The following lines are written to the SAS log.

Output 2.2 *Log Output*

```
GOOD 1
11
```

See Also

- DO
- "DO Statement: Iterative"
- "DO OVER Statement"
- "DO UNTIL Statement"
- "DO WHILE Statement"
- "LEAVE Statement"

DEFAULT Statement

Enables you to control how CASL returns action results and action processing status.

Syntax

Form 1: **DEFAULT** RESULT=*variable-expression*;

Form 2: **DEFAULT** STATUS=*variable-expression*;

Form 3: **DEFAULT** CLEAR <*clear-option-1* <*clear-option-2* <...*clear-option-N*>>>

Arguments

Note:

- The DEFAULT statement in Forms 1 and 2 requires at least one literal argument, either RESULT= or STATUS=. You cannot specify the DEFAULT statement alone.

However, you can include both the RESULT= and STATUS= arguments in a single DEFAULT statement:

```
default result=r1;
default status=s1;
default result=r1 status=s2;
```

Specifying `default;` results in a WARNING in the log and the DEFAULT statement is ignored.

- The DEFAULT statement in Form 3 also requires at least one argument, either the RESULT or STATUS arguments or both. However, it cannot include either argument from Forms 1 and 2 (RESULT= and STATUS=):

```
default clear result;
default clear status;
default clear result status;
```

For example, specifying

```
default clear;
```

or

```
default clear result status=s1;
```

results in a WARNING in the log and the DEFAULT statement is ignored.

CLEAR
removes the default variable assignment in which an action's result and status are saved.

(clear-options)
removes the default variable assignment for either an action's result, its status, or both its result and status.

RESULT
removes the default variable assignment where the RESULT of an action is saved.

Examples
```
default clear result;
default clear status;
default clear result status;
```

STATUS
removes the default variable assignment where the status of an action is saved.

Example
```
default clear results status;
```

Example "Example: Set and Clear the Default Result and Status Variables"

RESULT=*variable-expression*
sets the default variable where results from an action are saved. *variable-expression* is any expression that results in either a variable or data structure element into which a value can be stored.

Alias R

Example `default result=myRes;`

STATUS=*variable-expression*
sets the default variable where the status of an action is saved. *variable-expression* is any expression that results in either a variable or data structure element into which a value can be stored.

Example `default status=myStat;`

Example "Example: Set and Clear the Default Result and Status Variables"

Example: Set and Clear the Default Result and Status Variables

The following example uses the DEFAULT statement to create default values for the action status and result variables. It then uses the CLEAR statement to clear the default values.

```
proc cas;
   builtins.help status = s1 result = r1 / action = "serverStatus"; /*
1 */
run;
   default status = d["s2"] result = d["r2"];                        /*
2 */
   builtins.help / action = "serverStatus";
   print d["s2"] d["r2"];                                           /*
3 */
   describe d["s2"] d["r2"];                                        /*
4 */
run;
   default clear status result;                                     /*
5 */
   builtins.help / action = "serverStatus";
run;
quit;
```

1 Run the builtins.help action and manually specify the `status=` and `result=` variable assignments. The default for `status=` and `result=` can be simple variable names or they can be something more complex, like array elements or dictionary entries. In this example, the dictionary keys `s2` and `r2` specify entries in dictionary `d`.

2 Specify the DEFAULT statement with STATUS= and RESULT= to set their default values to `s2` and `r2`, respectively. When you specify the builtins.help action, the STATUS= and RESULT= options are automatically set to the variables that are specified for them in the DEFAULT statement. The builtins.help action also prints the result to the log.

3 Specify the PRINT statement to print the results and the status.

4 Specify the DESCRIBE statement to view the data types of the dictionary keys `s2` and `r2`.

5 Specify the DEFAULT CLEAR statement with the RESULT option and the STATUS option to clear the default values.

Output 2.3 *Output for Setting and Clearing the Default Result and Status Variables*

```
{severity=0,reason=0,,statusCode=0} {serverStatus=TRUE}
dictionary ( 4 entries, 4 used);
 [severity] int64_t;
 [reason] int64_t;
 [status] nil;
 [statusCode] int64_t;
dictionary ( 1 entries, 1 used);
 [serverStatus] boolean;
{serverStatus=TRUE}
```

DELETE Statement

Deletes the specified value from memory.

Syntax

DELETE *value*

Required Argument

value
definition of the location for the value that is to be deleted. The value can be a variable, list, array, or a dictionary.

Details

If the specified value is a variable, the variable is deleted from the memory. If the specified value is contained inside an array or dictionary, then the value is deleted from memory. If the specified value is an array, dictionary, or a list, then entire array, dictionary, or list along with the values within are deleted.

Note: When deleting values from an array, CAS does not delete the position of that value. An array with n many values becomes a sparse array with n many positions after deleting a value or multiple values.

Example

The following example uses the DELETE statement to delete different data values. Once the data is deleted, the PRINT statement returns a warning that the variable is uninitialized and is set to missing.

```
*cas casauto;

proc cas;

/* Create your sample data */

   x=25;
   arr={1,5,25};
   dict = {John=90, Mary=92, Sally=89};

   print "x=" x "arr=" arr "dict=" dict;

/* Use the DELETE statement to delete your data */

   delete x;
   print "x=" x;

   delete arr[2];
   print "arr=" arr;
   print "dimension of arr="dim(arr);

   delete arr;
   print "arr=" arr;

   delete dict.John;
   print "dict=" dict;
   print "dimension of dict=" dim(dict);

   delete dict;
   print "dict=" dict;

run;
quit;
```

Output 2.4 *Log Output*

```
x=25 arr={1,5,25} dict={John=90,Mary=92,Sally=89}
WARNING: Variable 'x' is uninitialized. It has been set to missing.
x=.
arr={1,,25}
3
WARNING: Variable 'arr' is uninitialized. It has been set to missing.
arr=.
dimension of arr=1
dict={Mary=92,Sally=89}
dimension of dict=2
WARNING: Variable 'dict' is uninitialized. It has been set to missing.
dict=.
```

DESCRIBE Statement

Enables you to view the structure and data type of CASL variables and expressions.

Interaction: When results are saved to a variable, the DESCRIBE statement writes a description of the variable to the SAS log. The description shows the expanded arrays and dictionaries in order to view the depth of the variable.

Syntax

DESCRIBE *variable-name* | *expression*;

Required Arguments

variable-name
The name of the variable that you want to view structure and data types for.

expression
A combination of one or more values, constants, variables, operators, and functions that are evaluated using rules and precedence to compute a result.

Examples

Example 1: Using the DESCRIBE Statement

The following example stores the result of the serverStatus action in the variable named *r*.

```
cas casauto;
proc cas;
  action serverStatus result=r;
  describe r;
run;
```

The DESCRIBE statement displays the result of the action from above as a list. The List contains three entries: **About** (a list), **Server** (a table), and **Nodestatus** (a table). Within the first entry in the list, **About**, there is another list named **System**.

```
dictionary ( 3 entries, 3 used);
 [About] dictionary ( 12 entries, 12 used);
  [CAS] string;
  [Version] string;
  [VersionLong] string;
  [Viya Release] string;
  [Viya Version] string;
  [Copyright] string;
  [ServerTime] string;
  [System] dictionary ( 7 entries, 7 used);
   [Hostname] string;
   [OS Name] string;
   [OS Family] string;
   [OS Release] string;
   [OS Version] string;
   [Model Number] string;
   [Linux Distribution] string;
  [license] dictionary ( 5 entries, 5 used);
   [site] string;
   [siteNum] int32_t;
   [expires] string;
   [gracePeriod] int32_t;
   [warningPeriod] int32_t;
  [CASHostAccountRequired] string;
  [Transferred] string;
  [CASCacheLocation] string;
 [server] Table ( [1] Rows [2] columns
Column Names:
 [1] nodes            [Node Count      ] (int32)
 [2] actions          [Total Actions   ] (int32)
 [nodestatus] Table[nodes] ( [6] Rows [5] columns
Column Names:
 [1] name             [Node Name       ] (char)
 [2] role             [Role            ] (char)
 [3] uptime           [Uptime (Sec)    ] (double)
 [4] running          [Running         ] (int32)
 [5] stalled          [Stalled         ] (int32)
```

Use the DESCRIBE statement to learn about the structure of the serverStatus result, and you can access that information from any CASL program.

Example 2: Producing Results Using Previously Assigned Variables

In the previous example, you stored a list with three entries in the variable *r*. You can take one of the entries and print the data.

```
cas casauto;
proc cas;
  action serverStatus result=r;
  describe r;
  print r.server[1];
run;
```

The output gives you information about the CAS server.

```
{nodes=6,actions=10}
```

Example 3: Use the DESCRIBE Statement on an Expression

You can use the DESCRIBE statement on an expression. In the example below, the DESCRIBE statement describes the expression after it is evaluated.

```
proc cas;
  describe ((1500*0.09)+1500)*1;
run;
```

The SAS log displays the data type of the evaluated expression.

Output 2.5 *SAS Log*

```
double;
```

See Also

- "CASL Data Types" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "CASL Expressions" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "CASL Result Tables" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "CASL Variables" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "Using the DESCRIBE Statement" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

DO Statement

Creates a block of code that executes as one statement.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Example: "Example 3: Creating a User-Defined Function" on page 105

Syntax

DO;

... more CASL statements ...;

END;

Details

The statements between the DO and END statements are called a DO group. The DO statement is the simplest form of DO group processing. The statements between DO and END statements are called a DO group. You can nest DO statements within DO groups. A simple DO statement is often used within an IF-THEN/ELSE statement. See "IF Statement". A DO statement is executed depending on whether the IF condition is true or false.

Example: Using the DO Statement

In this simple DO group, the statements between DO and END are performed only when YEARS is greater than 7. If YEARS is less than or equal to 7, statements in the DO group after the ELSE statement execute.

```
proc cas;
  years = 5;
  if years>7 then
    do;
     months = years*12;
     put years= months=;
    end;
  else;
    do;
     yrsleft=7-years;
    end;
 print yrsleft;
run;
```

Output 2.6 *Log Output*

```
2
```

See Also

- "CONTINUE Statement"
- "END Statement"
- "LEAVE Statement"

DO Statement: Iterative

Executes statements between the DO and END statements repetitively, based on the value of an index variable.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Tips: The order of the optional TO and BY clauses can be reversed.

When you use more than one specification, each one is evaluated before its execution.

Syntax

DO <*variable*> = <*start*> TO<*stop*><BY *increment*> <WHILE (*condition*) | UNTIL (*condition*)>;

... *more CASL statements* ...;

END;

Optional Arguments

variable
names a variable whose value governs execution of the DO group.

start
specifies the initial value of the variable.

When both start and stop are present, execution continues (based on the value of increment) until the value of index-variable passes the value of stop. When only start and increment are present, execution continues (based on the value of increment) until a statement directs execution out of the loop, or until a WHILE or UNTIL expression that is specified in the DO statement is satisfied. If neither stop nor increment is specified, the group executes according to the value of start. The value of stop is evaluated before the first execution of the loop.

stop
specifies the ending value of the index variable.

BY *increment*
specifies a positive or negative number (or an expression that yields a number) to control the incrementing of index-variable.

The value of increment is evaluated before the execution of the loop. Any changes to the increment that are made within the DO group do not affect the number of iterations. If no increment is specified, the index variable is increased by 1. When increment is positive, start must be the lower bound and stop, if present, must be the upper bound for the loop. If increment is negative, start must be the upper bound and stop, if present, must be the lower bound for the loop

WHILE (*condition*) | UNTIL(*condition*)
evaluates, either before or after execution of the DO group, any condition that you specify. Enclose the condition in parentheses.

A WHILE expression is evaluated before each execution of the loop, so that the statements inside the group are executed repetitively while the expression is true. An UNTIL expression is evaluated after each execution of the loop, so that

the statements inside the group are executed repetitively until the expression is true.

Restriction A WHILE or UNTIL specification affects only the last item in the clause in which it is located.

Details

CASL supports several statements for DO group processing.

The following table lists the different kinds of DO statements available in CASL:

Statement	Description	Syntax
"DO Statement: Iterative" (p. 39)	The DO statement, the simplest form of DO group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements. For more information about the IF-THEN/ELSE statement, see IF-THEN/ELSE Statement.	**DO** <*variable*> = <*start*> TO <*stop*><BY *increment*> <WHILE (*condition*) | UNTIL (*condition*)>; … *more CASL statements* …; **END**;
"DO OVER Statement" (p. 43)	The DO OVER statement iterates over an array, dictionary, or table.	**DO** <*key*>, <*var*> **OVER** <*value*>; ... *more CASL statements ...;* **END**;
"DO UNTIL Statement" (p. 46)	The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.	**DO UNTIL** (*condition*); ... *more CASL statements ...;* **END**;
"DO WHILE Statement" (p. 48)	The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.	**DO WHILE** (*condition*); ... *more CASL statements ...;* **END**;

You can use the LEAVE statement to stop processing the current iteration and resume with the next statement in the sequence. You can use the "CONTINUE Statement" to stop processing the current iteration and resume with the next iteration.

Example: Using the Do Loop Statement

This statement identifies the loop variable as `i`.

```
proc cas;
   print "***********************************";
   print(note) "Results for do i=1 to 6 by 3:";
   do i=1 to 6 by 1.5;
      print "I=" i;
   end;
   print "***********************************";
   print(note) "Results for do i=1 to 6 while( i < 3):";
   do i=1 to 6 while( i < 3);
      print "I=" i;
   end;
   print "***********************************";
   print(note) "Results for do i=6 to 1 by -1.25 until ( i < 4):";
   do i=6 to 1 by -1.25 until ( i < 4);
      print "I=" i;
   end;
   print "***********************************";
run;
quit;
```

Output 2.7 *Log Output*

```
*********************************
NOTE: Results for do i=1 to 6 by 3:
I=1
I=2.5
I=4
I=5.5
*********************************
NOTE: Results for do i=1 to 6 while( i < 3):
I=1
I=2
*********************************
NOTE: Results for do i=6 to 1 by -1 until ( i < 4):
I=6
I=4.75
I=3.5
*********************************
```

See Also

- "CONTINUE Statement"

- "END Statement"
- "LEAVE Statement"

DO OVER Statement

Iterates over an array, dictionary, or table.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Example: "Example 4: Using Run-time Math Functions with Inline Operator" on page 106

Syntax

DO <*key*>, <*var*> **OVER** <*value*>;

... more CASL statements ...;

END;

Optional Arguments

key
specifies the variable that is assigned the key value for a dictionary element.

var
specifies the variable that is assigned the value associated with the key for an array or dictionary element.

value
specifies the array, dictionary, or table.

Details

CASL supports several statements for DO group processing.

The following table lists the different kinds of DO statements available in CASL:

Statement	Description	Syntax
"DO Statement: Iterative" (p. 39)	The DO statement, the simplest form of DO group processing, designates a group of statements to be executed as a unit, usually as a part of IF-	**DO** <*variable*> = <*start*> TO <*stop*><BY *increment*> <WHILE (*condition*) | UNTIL (*condition*)>; … *more CASL statements* …; **END**;

Statement	Description	Syntax
	THEN/ELSE statements. For more information about the IF-THEN/ELSE statement, see IF-THEN/ELSE Statement.	
"DO OVER Statement" (p. 43)	The DO OVER statement iterates over an array, dictionary, or table.	**DO** <*key*>, <*var*> **OVER** <*value*>; *... more CASL statements ...;* **END**;
"DO UNTIL Statement" (p. 46)	The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.	**DO UNTIL** (*condition*); *... more CASL statements ...;* **END**;
"DO WHILE Statement" (p. 48)	The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.	**DO WHILE** (*condition*); *... more CASL statements ...;* **END**;

You can use the LEAVE statement to stop processing the current iteration and resume with the next statement in the sequence. You can use the "CONTINUE Statement" to stop processing the current iteration and resume with the next iteration.

Example: Using the DO OVER Statement

The following example takes the results of an action, iterates over the results to insert the name of variables with 20 or fewer missing values into an array. That array can then be used as input to another action. For the complete example, see "Example" in *SAS Viya Platform: System Programming Guide*.

```
cas casauto;
proc cas;
action simple.summary result=CPSSum                    /*1*/
  table={caslib="casuser", name="phoneSubs"};
run;
proc cas;
```

```
  cinfo = findTable(CPSSum);                              /* 2 */
   nmissVar = {"Country Name", "Indicator Name"};         /* 3 */
 do col over cInfo;                                       /* 4 */
 if (col.NMiss <20) then do;                              /* 5 */
   nmissVar= nmissVar + col.Column;                       /* 6 */
 end;
 end;
print nmissVar;                                           /* 7 */
run;
```

1 The Summary action creates descriptive statistics such as sum, means, and nmiss. The RESULT= option saves the results in a variable named CPSSum. For more information, see summary Action.

2 Cinfo is the name of the variable that contains the results of the FINDTABLE function. FINDTABLE is a built-in function that searches for a value in a table. CPSSum is the variable that holds the results of the Summary action. For more information, see FINDTABLE Function.

3 NmissVar is the name of the variable that will hold the results of the program. The brackets { } indicate that a list is being made. “Country Name” and “Indicator Name” are columns to be included in the array. The body of the DO OVER loop contains code that uses the ASSIGNMENT statement and + operator to add values to the nmissVar array. For more information, see ASSIGNMENT Statement.

4 The DO OVER statement iterates over the results stored in the variable cInfo. Col is the name of the index for the array. For more information, see DO OVER Statement.

5 The IF-THEN DO statement finds columns that have a value of NMiss greater than twenty. For more information, see IF-THEN/ELSE Statement.

6 The columns are then added to the array in the variable nmissVar.

7 The PRINT statement prints the variable nmissVar to the SAS log. For more information, see PRINT Statement.

Output 2.8 *SAS Log*

```
{Country Name,Indicator Name,1990,1995,1996,1997,1998,1999,2000,
2001,2002,2003,2004,2005,2006,2007,2008,2009,2010,2011,2012,2013,2014}
```

See Also

- “CONTINUE Statement”
- “END Statement”
- “LEAVE Statement”

DO UNTIL Statement

Executes statements in a DO loop repetitively until a condition is true.

Requirement: An END statement must follow a DO statement and all of the DO group processing statements.

Syntax

DO UNTIL (*condition*);

... more CASL statements ...;

END;

Required Argument

condition

specifies one numeric or character expression evaluation that results in a Boolean result. The result is used to decide to which code path to execute next. The condition is executed at the bottom of the loop. This means that the body of the loop always executes at least once. When the expression evaluate is true, it stops the iteration of the loop.

Details

CASL supports several statements for DO group processing.

The following table lists the different kinds of DO statements available in CASL:

Statement	Description	Syntax
"DO Statement: Iterative" (p. 39)	The DO statement, the simplest form of DO group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements. For more information about the IF-THEN/ELSE statement, see IF-THEN/ELSE Statement.	**DO** <*variable*> = <*start*> TO <*stop*><BY *increment*> <WHILE (*condition*) | UNTIL (*condition*)>; … *more CASL statements* …; **END**;

Statement	Description	Syntax
"DO OVER Statement" (p. 43)	The DO OVER statement iterates over an array, dictionary, or table.	**DO** <*key*>, <*var*> **OVER** <*value*>; *... more CASL statements ...;* **END**;
"DO UNTIL Statement" (p. 46)	The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.	**DO UNTIL** (*condition*); *... more CASL statements ...;* **END**;
"DO WHILE Statement" (p. 48)	The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.	**DO WHILE** (*condition*); *... more CASL statements ...;* **END**;

You can use the LEAVE statement to stop processing the current iteration and resume with the next statement in the sequence. You can use the "CONTINUE Statement" to stop processing the current iteration and resume with the next iteration.

Example: Using a DO UNTIL Statement to Repeat a Loop

This statement repeats the loop until **x** is greater 10. The expression *x>10* is evaluated at the bottom of the loop. There are eleven iterations in all (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10).

```
proc cas;
   x=0;
   do until (x>10);
   print x;
   x = x+1;
   end;
run;
```

Output 2.9 *Output*

```
0
1
2
3
4
5
6
7
8
9
10
```

See Also

- "CONTINUE Statement"
- "END Statement"
- "LEAVE Statement"

DO WHILE Statement

Executes statements in a DO loop repetitively while a condition is true.

Restriction: An END statement must follow a DO statement and all of the DO group processing statements.

See: "DO Statement: Iterative" on page 39

Syntax

DO WHILE (*condition*);

... more CASL statements ...;

END;

Required Argument

condition

specifies one numeric or character expression evaluation that results in a Boolean result. The result is used to decide to which code path to execute next. The condition is evaluated prior to executing the body of the loop. This means the body of the loop may not execute.

Details

CASL supports several statements for DO group processing.

The following table lists the different kinds of DO statements available in CASL:

Statement	Description	Syntax
"DO Statement: Iterative" (p. 39)	The DO statement, the simplest form of DO group processing, designates a group of statements to be executed as a unit, usually as a part of IF-THEN/ELSE statements. For more information about the IF-THEN/ELSE statement, see IF-THEN/ELSE Statement.	**DO** <*variable*> = <*start*> TO <*stop*><BY *increment*> <WHILE (*condition*) | UNTIL (*condition*)>; … *more CASL statements* …; **END**;
"DO OVER Statement" (p. 43)	The DO OVER statement iterates over an array, dictionary, or table.	**DO** <*key*>, <*var*> **OVER** <*value*>; *... more CASL statements ...;* **END**;
"DO UNTIL Statement" (p. 46)	The DO UNTIL statement executes statements in a DO loop repetitively until a condition is true, checking the condition after each iteration of the DO loop.	**DO UNTIL** (*condition*); *... more CASL statements ...;* **END**;
"DO WHILE Statement" (p. 48)	The DO WHILE statement executes statements in a DO loop repetitively while a condition is true, checking the condition before each iteration of the DO loop.	**DO WHILE** (*condition*); *... more CASL statements ...;* **END**;

You can use the LEAVE statement to stop processing the current iteration and resume with the next statement in the sequence. You can use the "CONTINUE Statement" to stop processing the current iteration and resume with the next iteration.

Example: Using a DO WHILE Statement

These statements repeat the loop while **x** is less than ten. The expression *x < 10* is evaluated at the top of the loop. There are ten iterations in all (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

```
proc cas;
   x=0;
   do while (x < 10);
   print x;
   x = x + 1;
   end;
run;
```

Output 2.10 *Output*

```
0
1
2
3
4
5
6
7
8
9
```

See Also

- "CONTINUE Statement"
- "END Statement"
- "LEAVE Statement"

ELSE Statement

If the expression specified by an IF statement results in a false value, the statement immediately after the ELSE statement is executed.

See: "IF Statement"

Syntax

ELSE *statement*;

Example

In this simple DO group, the statements between DO and END are performed only when YEARS is greater than 7. If YEARS is less than or equal to 7, statements in the DO group after the ELSE statement execute.

```
proc cas;
  years = 5;
  if years>7 then
    do;
     months = years*12;
     put years= months=;
    end;
  else;
    do;
     yrsleft=7-years;
    end;
 print yrsleft;
run;
```

Output 2.11 *Log Output*

```
2
```

END Statement

Ends a DO group, FUNCTION, or SELECT group processing.

Requirement: The END statement must be the last statement in a DO group, FUNCTION, or a SELECT group.

Syntax

END;

Without Arguments

Use the END statement to end DO group, FUNCTION, or SELECT group processing.

See Also

Statements

- “DO Statement”

- "DO Statement: Iterative"
- "DO OVER Statement"
- "DO UNTIL Statement"
- "DO WHILE Statement"
- "FUNCTION Statement"
- "SELECT Statement"

ENDEXTERNALSOURCE Statement

Marks the end of the EXTERNALSOURCE statement.

Syntax

ENDEXTERNALSOURCE;

Without Arguments

There are no arguments or parameters for the ENDEXTERNALSOURCE statement.

Details

- When an ENDEXTERNALSOURCE statement is used within PROC CAS to terminate an EXTERNALSOURCE text block, it must appear on a line by itself and be followed by a semicolon statement terminator. See "Example: Using the EXTERNALSOURCE Statement".
- Any text following the semicolon on the same line as the ENDEXTERNALSOURCE statement is ignored. See "Example: Using the EXTERNALSOURCE Statement".
- When an EXTERNALSOURCE statement is processed by either the EXECUTE function or the runCasl Action, the EXTERNALSOURCE statement and ENDEXTERNALSOURCE statements do not need to be on a separate line from the embedded text. Both statements must be terminated with a semicolon. See "Example 2: Using the EXTERNALSOURCE Statement with the EXECUTE Function".

Example: Using the EXTERNALSOURCE Statement

The following example uses the EXTERNALSOURCE statement to save IML code in a variable and then executes the IML code using the iml action.

```
proc cas;
   action loadactionset / actionset='iml';              /* 1 */
   set echoextsrc;                                      /* 2 */
   externalsource pgm;                                  /* 3 */
      a = 123;
      x =/* embedded comment */ 3;                      /* 4 */
      y=a-x;
   endexternalsource; print "This text is ignored";     /* 5 */
   print "This statement is part of regular CASL code"; /* 6 */
   run;
   iml / code=pgm;                                      /* 7 */
   run;
quit;
```

1 Load the iml action set.

2 Specify the ECHOEXTSRC option in the SET statement to enable SAS to echo the embedded text to the SAS log.

3 Specify the EXTERNALSOURCE statement followed by the variable name `pgm` and a semicolon. Specify this statement on a line by itself.

4 Specify the embedded text that you want to save into the variable `pgm`. Notice that you do not have to surround the embedded text in quotation marks or end the statements in the embedded text with a semicolon.

5 Specify the ENDEXTERNALSOURCE statement on a new line (by itself) to terminate the code block. The ENDEXTERNALSOURCE statement must be followed immediately by a semicolon. The text that is specified after the semicolon and on the same line as the ENDEXTERNALSOURCE statement is ignored.

6 Specify additional CASL code on a new line after the ENDEXTERNALSOURCE statement.

7 Specify the variable `pgm` in the Code= parameter of the iml action to execute the code contained in the variable `pgm`.

Output 2.12 *Log Output for Using the EXTERNALSOURCE Statement*

```
82   proc cas;
83      action loadactionset / actionset='iml';
84      set echoextsrc;
85      externalsource pgm;
86         a = 123;
87         x =/* embedded comment */ 3;
88         y=a-x;
89      endexternalsource; print "This text is ignored";      /*  */
90      print "This statement is part of regular CASL code";  /*  */
91      run;
NOTE: Active Session now CASAUTO.
NOTE: Added action set 'iml'.
{actionset=iml}
This statement is part of regular CASL code
92      iml / code=pgm;                  /*  */
93      run;
94   quit;
NOTE: PROCEDURE CAS used (Total process time):
      real time           0.22 seconds
      cpu time            0.00 seconds
```

See Also

- "SOURCE Statement"
- "EXTERNALSOURCE Statement"

ENDSOURCE Statement

Marks the end of the SOURCE statement.

Restriction: SOURCE and ENDSOURCE statements cannot be nested.

Syntax

ENDSOURCE;

Without Arguments

There are no arguments or parameters for the ENDSOURCE statement.

Details

When ENDEXTERNALSOURCE is used as a top-level statement within PROC CAS to terminate an EXTERNALSOURCE text block, it must appear on a line by itself and be followed by a semi-colon statement terminator. Any text following the semi-colon on the same line is ignored.

When an ENDEXTERNALSOURCE statement is processed either by the EXECUTE function or on a CAS server, when running the sccasl.runCasl action, the statement does not need to be on a separate line but must be followed by a semi-colon. Any text following the semi-colon is interpreted as CAS language code.

Example

```
proc cas;
  source pgm;
    v1='"weight"'n/"hei'ght"n;
    if missing('ag"e'n) then v2=0;
      else v2=v1/'ag"e'n;
  endsource;
run;
  print pgm;
run;
quit;
```

Output 2.13 *Log Output*

```
v1='"weight"'n / "hei'ght"n; if missing('ag"e'n) then v2=0;
else v2=v1/'ag"e'n;
```

See Also

"SOURCE Statement"

EXIT Statement

Exits the current block of CASL code.

Syntax

EXIT *expression*;

Required Argument

expression
a combination of one or more values, constants, variables, operators, and functions that are evaluated using rules and precedence to compute a result.

EXTERNALSOURCE Statement

Enables you to embed text in the program as a block of code and then assign the code to a given variable.

Restriction: EXTERNALSOURCE and ENDSOURCE statements cannot be nested.

Requirement: The "ENDEXTERNALSOURCE Statement" is required to terminate the code block.

See: "ENDEXTERNALSOURCE Statement"

Syntax

EXTERNALSOURCE *variable-expression*;
<*embedded-text*>
ENDEXTERNALSOURCE;

Required Arguments

variable-expression
specifies the variable to assign the embedded text to, or an expression yielding the variable to which the text is assigned.

ENDEXTERNALSOURCE;
specifies the end of the EXTERNALSOURCE statement.

Optional Argument

embedded-text
specifies a string representation of the value.

Details

The EXTERNALSOURCE statement preserves both the content and the formatting of the embedded text. Use of the EXTERNALSOURCE statement is necessary when the embedded text is code that is written in a programming language that is sensitive to white space, like Python.

The statement is preferable to embedding code in a quoted text string, which does not preserve line breaks and might require the insertion of additional white space.

Use of the EXTERNALSOURCE statement is not limited to languages like Python, but can also be used for other languages within SAS Viya, such as IML, CMP, DS2, DATA step, SQL, and CASL.

Here are a few more considerations for the EXTERNALSOURCE statement:

- The EXTERNALSOURCE statement can be preferable to the SOURCE statement, which might require the insertion of additional white space in some cases.
- Unless used with the EXECUTE function, the EXTERNALSOURCE statement, as the opening statement, must appear on a line by itself and be followed by a semicolon. See “Example 1: Using the EXTERNALSOURCE Statement”.
- The embedded text must start on a new line following the EXTERNALSOURCE statement and semicolon. The embedded text does not need to be contained within quotation marks or end in a semicolon. See “Example 1: Using the EXTERNALSOURCE Statement”.
- After you specify the embedded text, you must specify an ENDEXTERNALSOURCE statement to terminate the code block. The ENDEXTERNALSOURCE statement must be specified on a new line by itself. Any text that is specified after `ENDEXTERNALSOURCE;` and on the same line as the ENDEXTERNALSOURCE statement is ignored. See “Example 1: Using the EXTERNALSOURCE Statement”.
- When an EXTERNALSOURCE statement is processed by either the EXECUTE function or the runCasl Action, the EXTERNALSOURCE statement and ENDEXTERNALSOURCE statements do not need to be on a separate line from the embedded text. However, the statements do need to be terminated with a semicolon. See “Example 1: Using the EXTERNALSOURCE Statement”.

Comparisons

The EXTERNALSOURCE statement is like the SOURCE statement, except for in the following cases:

- In the EXTERNALSOURCE statement, white space and line breaks are preserved.
- The embedded text is saved without modification to the specified string. The text is saved *as is* without the need to modify or add formatting. In the SOURCE statement, the embedded text does not preserve line breaks and might require the insertion of additional white space or other formatting.
- Macro processing is not supported with the EXTERNALSOURCE statement nor are references to macros, macro variables, and macro functions. Macro processing is supported in the SOURCE statement.
- By default, the embedded text in the EXTERNALSOURCE statement code block is not echoed to the SAS log. The embedded text in the SOURCE statement code block is echoed to the SAS log by default. See "Example 1: Using the EXTERNALSOURCE Statement".

  Note: You can specify the ECHOEXTSRC option in the SET statement to enable SAS to echo the embedded text in the EXTERNALSOURCE statement code block to the SAS log. For an example, see "Example 1: Using the EXTERNALSOURCE Statement".

Examples

Example 1: Using the EXTERNALSOURCE Statement

The following example uses the EXTERNALSOURCE statement to save IML code in a variable and then executes the IML code using the iml action.

```
proc cas;
   action loadactionset / actionset='iml';              /* 1 */
   set echoextsrc;                                      /* 2 */
   externalsource pgm;                                  /* 3 */
      a = 123;
      x =/* embedded comment */ 3;                      /* 4 */
      y=a-x;
   endexternalsource; print "This text is ignored";     /* 5 */
   print "This statement is part of regular CASL code"; /* 6 */
   run;
   iml / code=pgm;                                      /* 7 */
   run;
quit;
```

1 Load the iml action set.

2 Specify the ECHOEXTSRC option in the SET statement to enable SAS to echo the embedded text to the SAS log.

3 Specify the EXTERNALSOURCE statement followed by the variable name pgm and a semicolon. Specify this statement on a line by itself.

4 Specify the embedded text that you want to save into the variable pgm. Notice that you do not have to surround the embedded text in quotation marks or end the statements in the embedded text with a semicolon.

5 Specify the ENDEXTERNALSOURCE statement on a new line (by itself) to terminate the code block. The ENDEXTERNALSOURCE statement must be followed immediately by a semicolon. The text that is specified after the semicolon and on the same line as the ENDEXTERNALSOURCE statement is ignored.

6 Specify additional CASL code on a new line after the ENDEXTERNALSOURCE statement.

7 Specify the variable pgm in the Code= parameter of the iml action to execute the code contained in the variable pgm.

Output 2.14 *Log Output for Using the EXTERNALSOURCE Statement*

```
82   proc cas;
83      action loadactionset / actionset='iml';
84      set echoextsrc;
85      externalsource pgm;
86         a = 123;
87         x =/* embedded comment */ 3;
88         y=a-x;
89      endexternalsource; print "This text is ignored";      /*  */
90      print "This statement is part of regular CASL code";  /*  */
91      run;
NOTE: Active Session now CASAUTO.
NOTE: Added action set 'iml'.
{actionset=iml}
This statement is part of regular CASL code
92      iml / code=pgm;                  /*  */
93      run;
94   quit;
NOTE: PROCEDURE CAS used (Total process time):
      real time           0.22 seconds
      cpu time            0.00 seconds
```

Example 2: Using the EXTERNALSOURCE Statement with the EXECUTE Function

The following example shows how to use the EXTERNALSOURCE statement with the EXECUTE function.

```
proc cas;
   execute( "print 'this is my program'; externalsource mystring; 10*5
= 50; i*2=x endexternalsource;" );
run;
   print mystring;
run;
quit;
```

Output 2.15 *Log Output for the EXTERNALSOURCE Statement Used with the EXECUTE Function*

```
12
13    proc cas;
14       execute( "print 'this is my program'; externalsource mystring; 10*5 =
50; i*2=x endexternalsource;" );
15    run;
this is my program
16       print mystring;
17    run;
10*5 = 50; i*2=x
18    quit;
```

See Also

- "ENDEXTERNALSOURCE Statement"
- "SOURCE Statement"

FILE Statement

The FILE statement changes the default output location for the PRINT statement. You can choose to write the output to an external file, the log, or ODS.

Default: Defined by the application.

Example: "Example 5: Training a Decision Tree Model and Score Data" on page 108

Syntax

FILE *file specification*;

Required Argument

file specification

file specification can take any of the following values:

name "path to file"

assigns a user-supplied name to an external file at the specified location. This name can then be used to redirect the output of the PRINT statement on page 79 to the specified external file. Once a name is assigned, it can be reused in future FILE statements without the need to re-specify the file path, as long as the PROC CAS session remains active.

ODS

a predefined system name that directs tabular output to the Output Delivery System (ODS) and non-tabular data to the SAS log. This is the default

behavior for the PRINT statement on page 79 when no other file specification is provided.

LOG
a predefined fileref referencing the SAS log that directs all PRINT statement output, regardless of its data type, to the SAS log.

Example: Use the FILE Statement to Change Output Locations

```
cas casauto;
proc cas;
   /* Create the string variable myText */
   myText="A line of text.";                        /*1*/

   /* Print myText to various locations */
   file myExternalFile "~/myExternalFile.txt";      /*2*/
   print(Note) "Writing text to file:";
   print myText;                                    /*3*/
   file log;                                        /*4*/
   print(Note) "Writing text to Log:";
   print(debug) myText;                             /*5*/
   file ods;                                        /*6*/
   print(Note) "Writing text to ODS:";
   print(debug) myText;                             /*7*/
   print " ";
   print " ";

   /* Assign a result table to variable myTable */
   table.fileinfo result=r /
      caslib="samples";
   myTable=r.FileInfo;                              /*8*/

   /* Print myTable to various locations */
   file myExternalFile;                             /*9*/
   print(Note) "Writing table to file:";
   print myTable;                                   /*10*/
   file log;                                        /*11*/
   print(Note) "Writing table to Log:";
   print(debug)  myTable;                           /*12*/
   file ods;                                        /*13*/
   print(Note) "Writing table to ODS:";
   print myTable;                                   /*14*/
run;
```

1 Creates a string variable named `myText` containing some constant text.

2 Assigns the name `myExternalFile` to "`~/myExternalFile.txt`", and then sets the active output location to myExternalFile.

3 Prints the value of `myText`.

4 Sets the active output location to log.

5 Prints the value of `myText`.

6 Sets the active output location to ods.

7 Prints the value of `myText`.

8 Creates a result table type variable named `myTable` containing a list of files.

9 Sets the active output location to `myExternalFile`.

10 Prints the value of `myTable`.

11 Sets the active output location to log.

12 Prints the value of `myTable`.

13 Sets the active output location to ods.

14 Prints the value of `myTable`.

See Also

- "OUTPUT Statement"
- PRINT Statement on page 79
- fileInfo Action

FNC Statement

Lists all common functions by name and category with simple descriptions.

Syntax

FNC *category-name*;

Required Argument

category-name
specifies the name of the category to display functions from. For a list of category names, see "Example: Display All Common Functions and Their Categories".

Example: Display All Common Functions and Their Categories

```
proc cas;
   fnc cate;
run;
```

Output 2.16 *SAS Log*

```
bitwise
char
combinatorial
datetime
distance
financial
math
probability
quantile
random
special
statistics
trig
```

See Also

- "CASL Built-In Functions"
- "Common Functions"
- "FUNCTION Statement"
- "FUNCTIONLIST Statement"

FORMAT Statement

Associates a format with a single variable or a list of variables.

Tip: You can specify multiple lists of variables and formats.

Syntax

FORMAT *variable-1...variable-N format*;

Details

Removing a Format

To remove a format, specify a list of variables with no format specification at the end. In the following example, *Todaysdate*, *Tdate*, *Tod*, and *DateT* are variables that currently have a format associated with each variable. Since there is no format specification in the FORMAT statement below, this will remove the associated formats from the variables.

```
format todaysdate tdate tod datet;
```

Applying a Format to a Column Within A Result Table

You can apply a format to a column within a result table. In the following example, the result table contains three columns, *Cylinders*, *Wheels* and *MSRP*, that needs to be formatted. *Cylinders* and *Wheels* are formatted with Best5. and the *MSRP* column is formatted with Best8.

> **TIP** Verify that at the end of each format there is a period, ".", following the format name.

```
format table "cylinders" "wheels" best5. "msrp" best8.;
```

Example: Applying a Format

The following examples applies the format Date12. to the variables *Todaysdate*, *Tdate*, and *DateT* along with the format Time12. to the variable *Tod*.

```
proc cas;
   format todaysdate tdate datet date5. tod TIME12.;
run;
```

FUNCTION Statement

Creates a new function that can be called in an expression.

Default:	The default return value will be zero unless the return statement is specified.
Requirement:	An END or ENDSUB statement must follow a FUNCTION statement.
Note:	The END and ENDSUB statements can be used interchangeably to terminate a FUNCTION definition. ENDSUB is provided for compatibility with FCMP-style function definitions and SOURCE blocks that are stored using functions such as UPLOAD_CASLSTORE.
Example:	"Example 3: Creating a User-Defined Function" on page 105

Syntax

FUNCTION *function-name* (<*parameter–1*><, *parameter–2*><, ..., *parameter–n*>);
   statements;
END | ENDSUB;

Required Arguments

function-name
   specifies a user-defined function name.

statement
is any series of CASL statement.

Optional Argument

parameter
specifies a variable name that holds an argument value passed to the function when it is called.

Examples

Example 1: Creating a User-Defined Function

This example defines a new function and executes the function using the IF-ELSE/THEN, DO, and PRINT statements. This example also uses an internal function, *factorial*, to define and execute the newly created function *x*.

Program

```
proc cas;                                                    /* 1 */
   function factorial(x);                                    /* 2 */
     if (x < 1.0) then return(x);                            /* 3 */
     else do;
        return exp(lgamma (x+1));
     end;                                                    /* 4 */
   end;
run;
   do i = 1 to 9;                                            /* 5 */
      x = factorial(i);
      print "   Factorial (" i ") = " put(x,best9.);         /* 6 */
   end;
   x = factorial(75);
   print "Factorial of 75 = " x;
run;
   x = factorial(75.0);
   print "Factorial of 75.0 = " x;
run;
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The FUNCTION statement creates a new function that takes one argument.

3 This function uses the **lgamma** and **exp** internal functions to calculate the factorial of a number. The IF statement asks whether the value of *x* is less than 1.0. The ELSE statement asks to return the run-time math expression function lgamma to be multiplied by the value of *x* and added 1.

4 The END statement ends the function processing.

Note: ENDSUB can also be used to terminate a FUNCTION definition. ENDSUB behaves the same as END when terminating a FUNCTION definition. Existing programs that use END continue to work unchanged.

5 The DO statement calls the function with arguments 1–9.

6 The PRINT statement prints the results.

Example Code 2.1 *Log Output: Define a New Function*

```
Factorial (1 ) =         1
Factorial (2 ) =         2
Factorial (3 ) =         6
Factorial (4 ) =        24
Factorial (5 ) =       120
Factorial (6 ) =       720
Factorial (7 ) =      5040
Factorial (8 ) =     40320
Factorial (9 ) =    362880
Factorial of 75= 2.480914E109
Factorial of 75.0= 2.480914E109
```

Example 2: Creating a User-Defined Function with ENDSUB Statement

This example defines a user-defined function named `add` that accepts two arguments and returns their sum. The function definition is terminated with the ENDSUB statement, and the function is then called to calculate and display a result. The example demonstrates that ENDSUB can be used as an alternative to END when defining a CASL function.

```
proc cas;                          /* 1 */
   function add(a,b);              /* 2 */
      return(a+b);                 /* 3 */
   endsub;                         /* 4 */
run;

   x=add(2,3);                     /* 5 */
   print x;                        /* 6 */
run;
```

1 The PROC CAS statement starts a CAS session and enables execution of CASL statements.

2 The FUNCTION statement defines a user-defined function named `add` that accepts two parameters, `a` and `b`.

3 The RETURN statement returns the sum of the two parameter values.

4 The ENDSUB statement terminates the function definition. ENDSUB can be used as an alternative to END.

5 The function is called with the arguments `2` and `3`, and the returned value is assigned to variable `x`.

6 The PRINT statement displays the value of x.

Example Code 2.2 *Log Output*

```
5
```

See Also

- "CALL Statement"
- "END Statement"
- "FUNCTIONLIST Statement"

FUNCTIONLIST Statement

Lists all CASL built-in functions and user defined functions by name with simple descriptions.

Syntax

FUNCTIONLIST <*name*>;
FLIST <*name*>;

Optional Argument

If a name is not specified, then available functions are printed to the SAS log.

name
specifies the name of the function to list information about.

You can use the following names to view which functions are available to you:

- **user** specifies user-defined functions
- <*name*> specifies imported function by extension name

Example

This example uses the FUNCTIONLIST statement to list information about the dictionary function.

```
proc cas;
   functionlist dictionary;
run;
```

The following output is printed to the SAS Log:

Output 2.17 *Log Output*

```
NOTE: dictionary      : Search for a value in a dictionary given the key
NOTE:                 :     value = dictionary( dictionary, string, recurse);
```

GLOBAL Statement

Creates a variable that can be accessed and used anywhere within a CASL program.

Syntax

GLOBAL *variable*;

Required Argument

variable
specifies a variable name.

Details

Variables have local or global scope. Local variables exist only within the function where they were created. Global variables can be accessed and used anywhere within a CASL program. You can use the LOCAL statement to explicitly create a variable that is local to a function, and the GLOBAL statement to explicitly create a global variable.

CAUTION
If you do not use the LOCAL statement to create a local variable in a function, the variable is implicitly global. If a global variable of the same name exists, the global variable is referenced. It is a best practice to use the LOCAL statement when creating variables inside a function.

- Global variables can be accessed and used anywhere within a CASL program.
- To explicitly create a global variable, use the GLOBAL statement.
- A variable is implicitly global if it is created outside of a function.
- A variable is implicitly global if it is created inside of a function, but is not a control loop variable or created by the LOCAL statement
- Global variables persist until the QUIT statement for PROC CAS is submitted.
- The LOCAL statement explicitly created a variable that is local to the function where it is created. Variables created using the LOCAL statement exist only within that function.

- Loop iteration variables used within a function implicitly have local scope and exist only within that function.
- Local variables persist until the function returns.

Example: Specifying a Global Variable

The following example creates two global variables, both named x. Because the variable created inside the function is created with global scope, it overwrites variables of the same name.

```
proc cas;
   function myFunction( y );
      global x;                                              /* 1 */
      print "NOTE: When myFunction begins execution, x=" x " and y=" y;
      x=y;
   end func;
run;
   x=5;                                                      /* 2 */
   print "NOTE: Before myFunction execution, x=" x;
   myFunction(2);
   print "NOTE: After myFunction execution, x=" x;
run;
quit;
```

1 The GLOBAL statement creates a variable named x that has global scope.

2 The x variable created outside of the function is also global.

Output 2.18 *SAS Log*

```
NOTE: Before myFunction execution, x=5
NOTE: When myFunction begins execution, x=5  and y=2
NOTE: After myFunction execution, x=2
```

See Also

- "Variable Scope" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "LOCAL Statement"

GOTO Statement

Transfers execution immediately to a labeled statement.

Syntax

GOTO *label*;

Required Argument

label
specifies a statement label that identifies the GOTO destination. Statement labels are defined by using the Label statement.

Details

The destination label for the GOTO statement must be within the same CASL method. You must specify the label argument or an error will occur. Statement labels are defined by using the Label statement.

Comparisons

GOTO statements can often be replaced by DO-END and IF-THEN/ELSE programming logic.

Example: Using the GOTO Statement

In the following example, x is initialized to zero. The DO WHILE loop executes, incrementing x by one each time it loops. When x=5, the program executes the GOTO statement and exits the DO WHILE loop. The GOTO statement enables the DO WHILE loop to end prematurely (before the x<10 condition is met). The GOTO statement refers to the null label, `done`.

```
proc cas;
   x=0;
   do while (x < 10);
      print x;
      x = x + 1;
      if (x=5) then do;
         put "x now equals 5 x= " x;
         goto done;
      end;
   end;                                          /* 1 */
   done:                                         /* 2 */
   print "finished, x should equal 5 x= " x;     /* 3 */
run;
quit;
```

1 End the IF statement.

2 End the DO WHILE loop.

3 Specify a null label that GOTO can refer to.

See Also

- "DO Statement"
- "DO Statement: Iterative"
- "DO OVER Statement"
- "DO UNTIL Statement"
- "DO WHILE Statement"
- "IF Statement"
- "Label Statement"
- "LEAVE Statement"

IF Statement

Conditionally executes a statement based on the value of an expression.

Note: If the expression results in a false value and there is an ELSE statement, the statement immediately after the ELSE statement is executed.

Tip: For greater efficiency, construct your IF-THEN/ELSE statement with conditions of decreasing probability.

See: "ELSE Statement" on page 50

Example: "Example 3: Creating a User-Defined Function" on page 105

Syntax

IF *condition* THEN *statement*;

ELSE *statement*;

Required Arguments

THEN
If the expression results in a true value, the statement immediately after the THEN argument is executed.

condition
determines which statement executes next.

statement
specifies any CASL statement.

Details

CAS evaluates the expression in an IF-THEN statement to produce a result that is either nonzero, zero, or missing. A nonzero value causes the expression to be true. A result of zero or missing causes the expression to be false.

If the expression results in a true value, the statement immediately after the THEN argument is executed. If the expression results in a false value and there is an ELSE statement, the statement immediately after the ELSE is executed. If there is no ELSE argument, the statement after the IF statement is executed.

Note: You can use an IF expression to select between two values based on whether a conditional evaluates to true or false. In addition, IF expressions can be nested to select between many values for a multi-way decision.

Comparisons

Use a SELECT group rather than a series of IF-THEN statements when you have a long series of mutually exclusive conditions. For more information, see SELECT Statement.

Example: Specifying IF-THEN/ELSE Statements

In this example, the IF-THEN/ELSE statements to specify whether Grade1, Grade2, and Grade3 are greater than or equal to the class average which is 80.

```
proc cas;
  grade1=90;
  grade2=89;
  grade3=74;
  if (grade1>=80) then put 'Grade 1 is equal to or above class
average.';
   else put 'Grade 1 is less than 80';
  if (grade2>=80) then put 'Grade 2 is equal to or above class
average.';
   else put 'Grade 2 is less than 80;
  if (grade3>=80) then put 'Grade 3 is equal to or above class
average.';
   else put 'Grade 3 is less than 80';
run;
quit;
```

Output 2.19 *SAS Log*

```
Grade 1 is equal to or above class average.
Grade 2 is equal to or above class average.
Grade 3 is less than 80
```

See Also

- "DO Statement"
- "DO Statement: Iterative"
- "DO OVER Statement"
- "DO UNTIL Statement"
- "DO WHILE Statement"
- SELECT Statement

Label Statement

Specifies a label target for the GOTO statement. No semicolon is required.

Supports: GOTO Statement

Syntax

label:*statement*;

Required Arguments

label
specifies any identifier, which is followed by a colon (:). You must specify the label argument.

Note: If the label contains non-Latin characters, you must enclose it in double quotation marks.

statement
specifies any executable statement, including a null statement (;). You must specify the statement argument.

Details

A label associates an identifier with a given statement so that the statement can be referred to by other statements, such as GOTO and LEAVE. You can have multiple labels for a statement.

Example

```
proc cas;
   flag=false;
   if flag then goto test;
   print (Note) "Your test ran successfully.";
   goto finish;
   test:print (Warn)"Your test did not run successfully.";
   finish:;
   end;
run;
```

Output 2.20 *SAS Log*

```
NOTE: Your test ran successfully.
```

See Also

- "GOTO Statement"
- "LEAVE Statement"

LEAVE Statement

Stops processing the current loop and resumes with the next statement in the sequence.

Tip: You can use the LEAVE statement to exit a DO loop or SELECT group prematurely based on a condition.

Syntax

LEAVE;

Details

You can use the LEAVE statement to exit an iterating DO loop prematurely. You can use the LEAVE statement either on its own or use it based on a condition (for example, in conjunction with an IF statement). When the LEAVE statement is encountered from within a DO loop, the loop iteration stops there and control returns from the loop to the first statement after the loop.

Note: The non-iterative DO statement in PROC CAS creates a block of code that executes as one statement. The LEAVE statement has no effect on programs that are created with the DO statement. The LEAVE statement affects the "DO Statement: Iterative".

Example: LEAVE Statement

This example illustrates the LEAVE statement.

```
proc cas;
   function evaltest(x, y);
   if (x == y) then put "GOOD ";
     else             put "ERROR ";
     end;
   do i = 1 to 10 by 1;
   if (i == 5) then leave;
   put i;
     end;
   test = evaltest( i, 5,test);
   put i;
run;
```

The following lines are written to the SAS log:

Output 2.21 *SAS Log*

```
1
2
3
4
GOOD
5
```

LOADACTIONSET Statement

Loads a SAS Cloud Analytic Services action set.

Requirement: The ability to load action sets into your session is subject to access controls.

Example: "Example 5: Training a Decision Tree Model and Score Data" on page 108

Syntax

LOADACTIONSET "*action-set-name*";

Required Argument

action-set-name
specifies the name of a SAS Cloud Analytic Services action set.

Example

This example uses the LOADACTIONSET statement to load the freqTab action set.

```
cas casauto;
proc cas;
   loadactionset "freqTab";
run;
```

The following outprint is printed to the SAS Log:

Output 2.22 *Log Output*

```
NOTE: Added action set 'freqTab'.
```

LOCAL Statement

Creates a variable that is local to a CASL function.

Syntax

LOCAL *variable*;

Required Argument

variable
specifies a variable name.

Details

Variables have local or global scope. Local variables exist only within the function where they were created. Global variables can be accessed and used anywhere within a CASL program. You can use the LOCAL statement to explicitly create a variable that is local to a function, and the GLOBAL statement to explicitly create a global variable.

CAUTION

If you do not use the LOCAL statement to create a local variable in a function, the variable is implicitly global. If a global variable of the same name exists, the global variable is referenced. It is a best practice to use the LOCAL statement when creating variables inside a function.

- Global variables can be accessed and used anywhere within a CASL program.
- To explicitly create a global variable, use the GLOBAL statement.
- A variable is implicitly global if it is created outside of a function.
- A variable is implicitly global if it is created inside of a function, but is not a control loop variable or created by the LOCAL statement
- Global variables persist until the QUIT statement for PROC CAS is submitted.
- The LOCAL statement explicitly created a variable that is local to the function where it is created. Variables created using the LOCAL statement exist only within that function.
- Loop iteration variables used within a function implicitly have local scope and exist only within that function.
- Local variables persist until the function returns.

Example: Specifying a Local Variable

The following example creates both a local and global variable named x. Because the variable created inside the function is created with local scope, it does not overwrite any other variables of the same name.

```
proc cas;
   function myFunction( y );
      local x;                    /* 1 */
      print "NOTE: When myFunction begins execution, x=" x " and y=" y;
      x=y;
      print "NOTE: After the myFunction assignment statement, x=" x "
and y=" y;
   end func;
run;
   x=5;                           /* 2 */
   print "NOTE: Before myFunction execution, x=" x;
   myFunction(2);
   print "NOTE: After myFunction execution, x=" x;
run;
quit;
```

1 The LOCAL statement creates a variable named x that has local scope.

2 The x variable created outside of the function is global.

Output 2.23 *SAS Log*

```
NOTE: Before myFunction execution, x=5
NOTE: When myFunction begins execution, x=  and y=2
NOTE: After the myFunction assignment statement, x=2  and y=2
NOTE: After myFunction execution, x=5
```

See Also

- "Variable Scope" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "GLOBAL Statement" on page 67

Null Statement

A statement that performs no operation.

Syntax

;

Details

The Null statement consists solely of a semicolon. It is a statement that performs no operation.

OUTPUT Statement

Sets the active output location.

Category: Data Input and Output

Default: ODS

Syntax

OUTPUT <ODS | LOG>;

Optional Arguments

<ODS>
specifies that the output is sent to an ODS destination.

<LOG>
specifies that the output is sent to the SAS log.

Examples

Example 1: Active Output Set to SAS Log

Use the OUTPUT statement with the LOG option to set the active output to the SAS Log.

```
cas casauto;

proc casutil;
   load data=sashelp.iris
   casout="iris"
   outcaslib="casuser"
   replace;
run;

proc cas;
  output log;
  table.columnInfo / table="iris";
run;
```

Output 2.24 *SAS Log*

Column	Label	ID	Type
Species	Iris Species	1	char
SepalLength	Sepal Length (mm)	2	double
SepalWidth	Sepal Width (mm)	3	double
PetalLength	Petal Length (mm)	4	double
PetalWidth	Petal Width (mm)	5	double

RawLength	FormattedLength	NFL	NFD
15	15	0	0
8	12	0	0
8	12	0	0
8	12	0	0
8	12	0	0

Example 2: Active Output Set to ODS

Use the OUTPUT statement with the ODS option to set the active output to the generate results in an HTML table.

```
cas casauto;

proc casutil;
   load data=sashelp.iris
```

```
   casout="iris"
   outcaslib="casuser"
   replace;
run;

proc cas;
  output ods;
  table.columnInfo / table="iris";
run;
```

Output 2.25 *ODS HTML Output*

Column Information for IRIS in Caslib casdata							
Column	Label	Id	Type	Length	Formatted Length	Format Width	Format Decimal
Species	Iris Species	1	char	15	15	0	0
SepalLength	Sepal Length (mm)	2	double	8	12	0	0
SepalWidth	Sepal Width (mm)	3	double	8	12	0	0
PetalLength	Petal Length (mm)	4	double	8	12	0	0
PetalWidth	Petal Width (mm)	5	double	8	12	0	0

See Also

"PRINT Statement"

PRINT Statement

Writes the values of constants, variables, and expressions to the current output location.

Note: When printing, the PRINT statement does not add spaces between two variables that contain a string. Add spaces as needed.

Example: "Example 4: Using Run-time Math Functions with Inline Operator" on page 106

Syntax

PRINT <(*level*)>*value-1*<*value-2*>...<*value-N*>;

Optional Arguments

(level)

specifies the diagnostic level that is associated with a log event from an action. When leveled messages are written to the log, their appearance is properly formatted based on the level.

level, from lowest to highest, can be one of the following:

TRACE
specifies a trace level message.

DEBUG
specifies a debug level message.

INFO
specifies an info level message.

NOTE
specifies a note level message.

WARN
specifies a warn level message.

ERROR
specifies an error level message.

Default TRACE

Example "Example 4: Adding Message Levels " on page 81

value
can be a constant, variable, or expression.

Tip You can include more than one value in your PRINT statement.

Examples

Example 1: Print One Value

The following example uses the PRINT statement to print one variable, *Grades*.

```
proc cas;
   x={96,98,82,83,87,82,99,99,100};
   y={85,90,87,85,92};
   z=x+y;
   Grades=sort(z);
   print Grades;
run;
```

Output 2.26 *SAS Log*

```
{82,82,83,85,85,87,87,90,92,96,98,99,99,100}
```

Example 2: Print Multiple Values

The following example uses the PRINT statement to print multiple variables, *x*, *y*, *z*, and *Grades*.

```
proc cas;
   x={96,98,82,83,87,82,99,99,100};
```

```
   y={85,90,87,85,92};
   z=x+y;
   Grades=sort(z);
  print x y z Grades;
run;
```

In the following SAS log, the order of the printed log is in the same order of the variables in the PRINT statement.

Output 2.27 *SAS Log*

```
{96,98,82,83,87,82,99,99,100} {85,90,87,85,92}
{96,98,82,83,87,82,99,99,100,85,90,87,85,92}
{82,82,83,85,85,87,87,90,92,96,98,99,99,100}
```

Example 3: Manage Spacing of String Variables

The PRINT statement does not automatically add spaces between values. The following example uses the PRINT statement to print some text and the values of variables *a*, *b*, *c*, and *d*. In the second PRINT statement in the example, add spaces as necessary to format the result.

```
proc cas;
   a="is";
   b="desired";
   c="text";
   d=".";
   print "This" a "my" b c d;
   print "This " a " my " b " " c d;
run;
```

In the following SAS log, the first line corresponds to the first PRINT statement, and the second line corresponds to the second PRINT statement.

Output 2.28 *SAS Log*

```
Thisismydesiredtext.
This is my desired text.
```

Example 4: Adding Message Levels

The following example uses the PRINT statement to add message levels to the SAS log.

```
proc cas;
   print;
   print "Plain text";
   print (trace) "TRACE level text";
   print (debug) "DEBUG level text";
   print (note) "NOTE level text";
   print (warn) "WARNING level text";
   print (error) "ERROR level text";
quit;
```

Output 2.29 *SAS Log*

```
Plain text
TRACE level text
      DEBUG level text
NOTE: NOTE level text
WARNING: WARNING level text
ERROR: ERROR level text
```

PRINTLST Statement

Prints to the listing file defined by the application. If the value is a list, the list will be transverse and each item will be displayed. If the item is a table, it will be printed to the current output location.

Alias: plist

Syntax

PRINTLST *expression*;

Required Argument

expression
A combination of one or more values, constants, variables, operators, and functions that are evaluated using rules and precedence to compute a result.

Example

```
proc cas;
   printlst {98, 74, 54, 18, 101, 67, 80, 90, 62};
run;
```

RETURN Statement

Returns a value from the current function.

Example: “Example 5: Training a Decision Tree Model and Score Data” on page 108

Syntax

RETURN *expression | value*;

Required Arguments

expression
The result of the expression is the value returned by the function to the calling environment (or to the caller).

value
specifies the value that is returned from a function.

Example: Using the RETURN Statement

The following example details how to create two functions to determine whether or not the class and the student are on track to complete a course successfully.

```
proc cas;
   grade={80,85,90,86,89,90,75,76,88,70,67};
   classavg=81;
   function evaltest (grade, classavg, student);          /* 1 */
   if (classavg<=67) then put "Class is ON TRACK";
    else put "Class is NOT ON TRACK";
   return (grade+1);                                      /* 2 */
end;
   function grade(classavg);
   return(67);                                            /* 3 */
end;
   grade=grade(1)+1;                                      /* 4 */
   test=evaltest(grade, 5, test);
    if (grade>=classavg) then put "Student is ON TRACK";
     else put "Student is NOT ON TRACK";
   put grade;
run;
```

1 The user-defined function Evaltest determines whether or not the class is on track. In this example, the class average is 81 and the class is on track because the class average is greater than 67.

2 The RETURN statement determines the value of the student's grade. The current student has a grade of 67, and all students get an additional point added to their grade. Therefore, the returned value is 68.

3 The user-defined function Grade determines the student's final grade and whether the student is on track to successfully complete the course. The RETURN statement uses the value of 67 to determine whether the student is on track. Since the value 67 is not equal to or greater than 81, the student is not on track.

4 In order to determine whether another class or student is on track, simply replace the value of 67 in the first IF statement and the second RETURN statement.

Output 2.30 *SAS Log*

```
Class is ON TRACK
Student is NOT ON TRACK
68
```

RUN Statement

Executes the previously entered SAS statements.

Syntax

RUN;

Without Arguments

The RUN statement executes the previously entered SAS statements.

Details

Although the RUN statement is not required between steps in a SAS program, using it creates a step boundary and can make the SAS log easier to read. CASL code is not executed until a RUN statement is read or until the PROC is terminated by another PROC, DATA, or QUIT statement. If a RUN statement is encountered, the code runs and more code will be accepted after the RUN statement. Any variables created in a preceding RUN block remain in scope for subsequent RUN blocks. After PROC CAS is terminated by a PROC, DATA, or QUIT statement, the variables are no longer in scope.

Note: Do not use the RUN statement within the DO loop. It will cause the DO loop to break and result in an error.

SAVERESULT Statement

Saves a result table in the variable in the form specified.

Restriction: When running server-side only, the SAVERESULT statement only saves an in-memory CAS table. All other formats are invalid.

Note: A column that contains more than 32,767 characters is too large to save in a data set.

Example: "Example 5: Training a Decision Tree Model and Score Data" on page 108

Syntax

Form 1: **SAVERESULT** *variable-name* <DATAOUT=<*libref.*>*data-set-name* | LIB=*libref*> <REPLACE | NOREPLACE>;

Form 2: **SAVERESULT** *variable-name* <CSV="*file-name*" | FILE=*output-file*> <REPLACE | NOREPLACE>;

Form 3: **SAVERESULT** *variable-name* <CASLIB=*caslib-reference-name*> <CASOUT="*table-name*"> <REPLACE | NOREPLACE>;

Required Argument

variable-name
specifies a variable that includes one or more result tables.

Optional Arguments

CASLIB=*caslib-reference-name*
specifies which caslib to use.

CASOUT="*table-name*"
specifies to transfer the result table to the server and store it as an in-memory table. By default, the active caslib is used but it can be overridden with the CASLIB= argument.

CSV="*file-name*"
specifies the CSV file name. The best practice is to specify a fully qualified path. If you do not specify a fully qualified path, then SAS attempts to create the file in the current working directory for the SAS process.

CSV files are saved with the session encoding. However, the encoding defaults to UTF-8 when you upload the file with the UPLOAD statement. To upload the saved file, you must provide the encoding that the file was created with.

Use the ENCODING= IMPORTOPTION parameter to specify the encoding. For information about the IMPORTOPTIONS option, see IMPORTOPTIONS=.

You can use the ENCODING function with the ENCODING= parameter to dynamically provide the CSV file encoding.

DATAOUT=<*libref.*>*data-set-name*
specifies the SAS library and data set name. The default libref is Work.

FILE="*external-file*" | *fileref*
specifies the external file or file reference to use.

"external-file"
specifies the file path. The best practice is to specify a fully qualified path. If you do not specify a fully qualified path, then SAS attempts to create the file in the current working directory for the SAS process.

fileref
specifies a SAS file reference.

LIB=*libref*
specifies which SAS library to use. The specified libref must be a V9 engine libref.

REPLACE | NOREPLACE
specifies to overwrite existing files or to prevent overwriting existing files. By default, SAS data sets, external files, and in-memory tables are not overwritten. Specify REPLACE to overwrite existing files.

Default NOREPLACE

Details

To replace the loaded table in CAS, specify the REPLACE option or drop the loaded table from CAS before executing the SAVERESULT statement. The following table identifies the output file type and the default REPLACE option for each file type.

Note: If DATAOUT= is not specified, the SAS data set is saved as Work.*result-table-name*.

Table 2.1 *SAVERESULT Output File Type and Default REPLACE Values*

Option	Output File Type	Default REPLACE Value
`casout=table-name`	CAS in-memory table	NOREPLACE
`CSV=path-name`	CSV	REPLACE
`dataout=libname.name`	SAS7bdat	REPLACE
`lib=libname`	SAS7bdat	REPLACE
`file=path-name`	Text	REPLACE

Examples

Example 1: Using the SAVERESULT Statement: SAS Data Set

This example program shows how the SAVERESULT statement is used to save a result table generated from actions.

```
proc cas;
  session casauto;
  output log;
  table.loadTable / path="iris.sashdat";                    /
*#1*/
  simple.summary result=iris / table={name="iris"};         /
*#2*/
  table01=findtable(iris);                                  /
*#3*/
  saveresult table01 dataout=work.resultIris;               /
*#4*/
run;
```

1 Use the table.loadTable action to load *Iris* from a caslib's data source.

2 Use the simple.Summary action to generate descriptive statistics of the *Iris* table.

3 *Table01* is a new table that stores the result of the function, FINDTABLE.

4 The SAVERESULT statement saves *Table01* as a SAS data set, ResultIris, in the Work library.

Example 2: Using the SAVERESULT Statement: CSV Files

Use the SAVERESULT statement to save your results from an action to CSV files.

```
proc cas;
  session casauto;
  output log;
  table.loadTable / path="iris.sashdat";
  simple.summary result=iris / table={name="iris"};
  tableIris=findtable(iris);
  saveresult tableIris csv="sum.csv";
run;
```

The following note is printed to the SAS log.

Output 2.31 *SAS Log*

```
NOTE: The CSV file sum.csv has been created.
```

See Also

"ACTION Statement"

SELECT Statement

Executes one of several statements or groups of statements.

Restriction: An END statement must follow a SELECT statement.

Note: SELECT statements can be nested.

Syntax

SELECT <(*select-expression*)>;

WHEN-1 <(*when-expression-1* ..., *when-expression-n*>) *statement*;
<...WHEN-n (*when-expression-1* ..., *when-expression-n*) > *statement*;

<OTHERWISE *statement*>;

END;

Required Arguments

(*select-expression*)
specifies an expression that evaluates to a single value.

(*when-expression*)
specifies any SAS expression, including a compound expression. SELECT requires you to specify at least one *when-expression*.

Note: You can separate multiple *when-expressions* with a comma and that is equivalent to separating them with the logical operator OR.

statement
specifies any executable SAS statement, including DO, SELECT, and null statements. You must specify the *statement* argument.

Details

Using WHEN Statements in a SELECT Group

The SELECT statement begins a SELECT group. SELECT groups contain WHEN statements that identify CAS statements that are executed when a particular condition is true. Use at least one WHEN statement in a SELECT group. An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. An END statement ends a SELECT group.

Note: SELECT statements can be nested.

Example: Using the SELECT statement

Use the SELECT statement to execute several statements. In the example below, the SELECT statement executes two WHEN expressions and one OTHERWISE statement.

```
proc cas;
  a=10;
  select (a);
    when (10) put a;
    when (20) put a=a*2;
      otherwise;
  end;
run;
```

See Also

- "DO Statement"
- "END Statement"
- "IF Statement"

SESSION Statement

Specifies a session to use for actions invoked by the program.

Example: "Example 5: Training a Decision Tree Model and Score Data" on page 108

Syntax

Form 1: **SESSION** *session-reference*;

Required Argument

session-reference
specifies the name of an existing session.

Examples

Example 1: Print the Session and the Session UUID

This example prints session UUID for the *casauto* session.

```
options cashost="cloud.example.com" casport=5570 casuser="sasdemo";
```

```
cas casauto;
proc cas;
    session casauto;
    action sessionid result=uuid;
    print uuid;
run;
```

Output 2.32 *Log Output*

```
{CASAUTO:Tue Sep 13 11:16:18 2016=4ebd26ad-9dc4-cf4f-a108-4f497d06a23f}
```

Example 2: Connect to an Existing Session

This example connects to an existing session using the UUID. UUID is session specific.

```
options cashost="cloud.example.com" casport=5570 casuser="sasdemo";
cas casauto uuid="4ebd26ad-9dc4-cf4f-a108-4f497d06a23f";
proc cas;
    session casauto;
    session.sessionstatus result=s;
    put s;
run;
```

SET Statement

Controls options that modify some behaviors of CAS actions and controls corresponding messages from the SAS log.

Syntax

SET DISP <ECHOEXTSRC><LOGS><LOGS=DEBUG | NOTE | TRACE | WARN><STDJSON><WARNINGDUP><USEWIDTH>

Arguments

The SET statement requires at least one of the following arguments:

DISP
specifies printing the disposition returned from a CAS action in the SAS log.

Alias DISPOSITION

ECHOEXTSRC
allows the embedded text in the EXTERNALSOURCE statement code block to be echoed to the SAS log.

Example "Example 1: Using the EXTERNALSOURCE Statement".

LOGS
specifies printing messages from CAS in the SAS log. The default value is TRACE.

Default TRACE

Tips The LOGS option without arguments is equivalent to LOGS=TRACE option. For information bout specifying additional types of log messages, see the LOGS= on page 91 option.

You can suppress the printing of messages from CAS to the log by specifying the LOGS option in the UNSET statement.

See For information about the UNSET statement, see "UNSET Statement".

LOGS=
enables you to specify any one of the following options to control messages to the log:

DEBUG
prints all log messages except those that are displayed using the TRACE option.

NOTE
prints all log messages except those that are displayed using the DEBUG and TRACE options.

Example "Example 3: Print Diagnostic Messages to the SAS Log".

TRACE
prints all log messages to the log.

Example "Example 3: Print Diagnostic Messages to the SAS Log".

WARN
prints all log messages except those that are displayed using the DEBUG, NOTE, and TRACE options.

STDJSON
specifies that the CASL2JSON function produces only text that is conformant with the JSON standard and that the JSON2CASL function accepts only text that is conformant with the JSON standard.

Setting STDJSON affects only the environment in which CASL code is executing, whether it is executing on the SAS Compute Server (SAS client) or on the CAS server.

- For backward compatibility with existing code, the STDJSON argument is unset by default so that variances from the standard are allowed by these functions. Setting the STDJSON argument in a program that is running on the SAS client affects only the behavior of the functions on the SAS client.

  To set the STDJSON option on the SAS client, specify `set stdjson;` in the PROC CAS code block:

```
proc cas;
   set stdjson;
```

```
run; quit;
```

- Setting the STDJSON argument in a program that is running on the CAS server affects only the behavior of the functions on the CAS server.

  To set the STDJSON option on the CAS server, ensure that `set stdjson;` appears in the code submitted to the sccasl.runCasl action.

```
proc cas;
   source pgm;
      set stdjson;
      < more-statements >;
   endsource;
   sccasl.runCasl / code=pgm;
 run;
```

The STDJSON setting can locally override the value of the CASL_STD_JSON environment variable. For more information about setting environment variables in configuration instances, see "Edit Server Configuration Instances" in *SAS Environment Manager: User's Guide*.

Default STDJSON is unset.

See "Edit Configuration Instances" in *SAS Viya Platform: Programming Run-Time Servers*

"CASL2JSON Function" on page 133

"JSON2CASL Function" on page 175

USEWIDTH

specifies the use of the defined column width of character string values in a table. Values with widths less than the defined width of the column are padded with blanks to the full defined column width. USEWIDTH affects only fixed-length character variables within a result table. Varying-length character variables are not affected.

Default USEWIDTH is unset.

Tip You can specify that all values are padded with blanks to the maximum measured width of all values in the column by using the UNSET statement.

WARNINGDUP

specifies printing a warning message in the SAS log when two or more response dictionary entries that are returned from a CAS action have the same key. For the runCasl action, the SEND_RESPONSE function can be used by CASL code to send one or more response dictionaries from the action to the client. When the responses contain duplicate named values, the last value received from the CAS action in a response becomes the value for that key.

Default WARNINGDUP is unset.

Tip You can specify that no warnings for duplicate response dictionary entries are printed by specifying the UNSET statement.

Example "Example 2: Print a Warning Message for Duplicate Keys to the SAS Log "

Examples

Example 1: Display the Disposition in the SAS Log

You can use the SET statement to turn on options. `SET DISP` is used to display the disposition in the SAS log.

```
proc cas;
   set disposition;                        /* 1 */
   builtins.help / action="serverStatus";
run;
   builtins.help / action="nonExistent";   /* 2 */
run;
   unset disposition;                      /* 3 */
   builtins.help / action="nonExistent";
run;
quit;
```

1 Specify the SET DISPOSITION statement to print the disposition to the log. The action runs successfully, so the following disposition is printed to the log: `NOTE: Disposition: Severity Normal`.

2 Because the SET DISPOSITION statement is still active and the `nonExistent` action is in error, the following disposition is printed to the log: `ERROR: Disposition: Severity Error`.

3 Specify the DISPOSITION option in the UNSET statement to stop printing disposition messages to the log.

Output 2.33 *Log Output for Display the Disposition in the SAS Log*

```
{serverStatus=TRUE}
ERROR: Action 'nonExistent' was not found.
ERROR: The action stopped due to errors.
ERROR: Disposition: Severity Error
ERROR: Action 'nonExistent' was not found.
ERROR: The action stopped due to errors.
```

Example 2: Print a Warning Message for Duplicate Keys to the SAS Log

The following example uses the WARNINGDUP option in the SET statement to print a warning message to the SAS log, because two response dictionary entries return the same key.

```
proc cas;
   source serversource;
      resp1={"t"=99};
      send_response(resp1);
```

```
      resp2={"t"=v};
      send_response(resp2);
      exit( {severity=0,reason=0,statusCode=0} );
   endsource;
   set warningdup;                                  /* 1 */
   sccasl.runCasl result = r status = sc  /
      code=serversource, vars={v=1};
   print r;
run;
quit;
```

1 Because the `respl` variable is duplicated in the program assignment, the SET WARNINGDUP statement prints a warning message to the log.

Note: The EXIT function exits the block of CASL code with the given status.

Output 2.34 *Log Output for Print a Warning Message for Duplicate Keys to the Log*

```
This is WARNING: The key 't' has a duplicate in the results.
{t=1}
```

Example 3: Print Diagnostic Messages to the SAS Log

The following example uses the SET LOGS statement, the SET LOGS= statement, and the UNSET LOGS statement to control messaging to the SAS log. The UNSET statement suppresses printing messages from CAS to the SAS log.

```
proc cas;
   source serversource;
       print (trace) "This is a trace-level message.";
       print (debug) "This is a debug-level message";
       print  (note) "This is a note-level message";
       print  (warn) "This is a warning-level message";
       print (error) "This is an error-level message";
       exit( {severity=0,reason=0,statusCode=0} );
   endsource;
   print "Specify LOGS (with no arguments) to print all messages to
the log";
   set logs;       /* 1 */
   sccasl.runCasl result = r
                  status = sc / code=serversource;
run;
   print "Specify LOGS=NOTE to print all messages except DEBUG and
TRACE to the log";
   set logs=note;  /* 2 */
   sccasl.runCasl result = r
                  status = sc / code=serversource;
run;
   print "Specify UNSET LOGS to suppress all messages to the log";
   unset logs;     /* 3 */
   sccasl.runCasl result = r
                  status = sc / code=serversource;
run;
```

```
quit;
```

1 Print all messages from CAS to the SAS log. The default value is TRACE.

2 Print all log messages except those that are displayed using the DEBUG option and the TRACE option.

3 Specify the UNSET statement to suppress printing messages from CAS in the SAS log.

Output 2.35 *Log Output for Print Diagnostic Messages to the Log*

```
Specify LOGS (with no arguments) to print all messages to the log
This is a trace-level message.
This is a debug-level message
NOTE: This is a note-level message
WARNING: This is a warning-level message
ERROR: This is an error-level message
Specify LOGS=NOTE to print all messages except DEBUG and TRACE to the log
NOTE: This is a note-level message
WARNING: This is a warning-level message
ERROR: This is an error-level message
Specify UNSET LOGS to suppress all messages to the log
```

See Also

- "UNSET Statement"

SOURCE Statement

Enables you to embed text in the program and assign it to a given variable.

Restriction: SOURCE and ENDSOURCE statements cannot be nested.

Note: A function definition in a SOURCE block can be terminated with either the END statement or the ENDSUB statement. For functions that are defined within SOURCE...ENDSOURCE blocks, SAS recommends using ENDSUB because it improves compatibility with FCMP-style function definitions and CASL source code that is stored by using functions such as UPLOAD_CASLSTORE.

Syntax

SOURCE *variable*;

<*text*>

ENDSOURCE;

Required Argument

variable
The variable to assign the text to.

Optional Argument

text
specifies a string representation of the value for the variable.

Examples

Example 1: Using the SOURCE Statement

The following example uses the SOURCE statement to embed text in the program. The embedded text is displayed in the SAS log when you use the PRINT statement. The embedded text is assigned to the *pgm* variable.

```
proc cas;
  source pgm;
    v1='"weight"'n/"hei'ght"n;
    if missing('ag"e'n) then v2=0;
      else v2=v1/'ag"e'n;
  endsource;
run;
  print pgm;
run;
quit;
```

Output 2.36 *SAS Log*

```
v1='"weight"'n / "hei'ght"n; if missing('ag"e'n) then v2=0;
else v2=v1/'ag"e'n;
```

Example 2: Using the SOURCE Statement with ENDSUB Statement

This example defines a CASL function named add within a SOURCE block and uses the ENDSUB statement to terminate the function definition. The UPLOAD_CASLSTORE function then saves the SOURCE block to a CASL store named `my_caslstore` in the casuser caslib.

```
proc cas;
   source mymodule;
      function add(a,b);
         return(a+b);
      endsub;
   endsource;

   upload_caslstore(
      {caslib="casuser",
       name="my_caslstore",
```

```
        replace=true},
      mymodule);
run;
```

Example Code 2.3 *Log Output*

```
NOTE: The table my_caslstore loaded into casuser with 1 observation and 6 variables.
```

See Also

"EXTERNALSOURCE Statement"

UNSET Statement

Turns off options that modify some behaviors of CAS actions and suppresses corresponding messages in the SAS log.

Tip: You can set which messages are written to the SAS log by using the SET statement.

Syntax

UNSET DISP <ECHOEXTSRC><LOGS><STDJSON><USEWIDTH><WARNINGDUP>;

Arguments

DISPOSITION
specifies printing the disposition returned from a CAS action to the SAS log.

Alias DISP

Example "Example: Using the UNSET Statement"

ECHOEXTSRC
prevents SAS from echoing the lines gathered by the EXTERNALSOURCE statement to the SAS log.

LOGS
specifies printing messages from CAS to the SAS log.

STDJSON
specifies that the CASL2JSON function can produce output text that is not strictly conformant with the JSON standard and that the JSON2CASL function can accept text that is not strictly conformant with the JSON standard.

For backward compatibility with existing code, the STDJSON argument is unset by default so that variances from the standard are allowed by these functions. Unsetting STDJSON affects only the environment in which CASL code is

executing, whether it is executing on the SAS Compute Server (SAS client) or the CAS server.

- Unsetting the STDJSON argument in a program that is running on the SAS client affects only the behavior of the functions on the SAS client.

  To unset the STDJSON option on the SAS client, specify `unset stdjson;` in the PROC CAS code block:

```
proc cas;
   unset stdjson;
run; quit;
```

- Unsetting the STDJSON argument in a program that is running on the CAS server affects only the behavior of the functions on the CAS server.

  To unset the STDJSON option on the CAS server, ensure that `unset stdjson` appears in the code submitted to the sccasl.runCasl action.

```
proc cas;
   source pgm;
      unset stdjson;
      < more-statements >;
   endsource;
   sccasl.runCasl / code=pgm;
 run;
```

The STDJSON setting can locally override the value of the CASL_STD_JSON environment variable. For more information about setting environment variables in configuration instances, see "Edit Server Configuration Instances" in *SAS Environment Manager: User's Guide*.

Default STDJSON is unset.

See "Edit Configuration Instances" in *SAS Viya Platform: Programming Run-Time Servers*

"CASL2JSON Function" on page 133

"JSON2CASL Function" on page 175

USEWIDTH

specifies defining the width of string columns in a table based on the maximum width that is used for each character value, instead of basing the width on a globally defined width for all values. USEWIDTH affects only fixed-length character variables within a result table. Varying-length character variables are not affected.

Default USEWIDTH is unset.

Tip You can specify that all values are padded with blanks to the full defined width of the column using the SET statement.

WARNINGDUP

specifies that no warnings for duplicate response dictionary entries are printed.

Example: Using the UNSET Statement

The UNSET statement turns off options. The `unset disposition` statement suppresses the printing of the disposition in the SAS log.

```
proc cas;
   session casauto;
   unset disposition;
   setsessopt / caslib="casuser";
run;
```

See Also

- "SET Statement" on page 90

UPLOAD Statement

Transfers a file from the SAS client to the server. After data transfer, the server loads the data into an in-memory table.

Restriction: When uploading CSV files, you must specify the CSV file encoding. See the IMPORTOPTIONS= option for more information.

Notes: The file received by the server is stored as temporary until the file is loaded to an in-memory table. Once it is loaded, then the file is removed.

Uploaded files

Syntax

UPLOAD

PATH=*path-to-file*

<CASOUT={*output-table-options*}>

<IMPORTOPTIONS={FILETYPE="BASESAS" | "CSV" | "DTA" | "ESP" | "EXCEL" | "FMT" | "EXCEL" | "HDAT" | "JMP" | "LASR" | "SPSS" | "XLSX"<, *fileType-specific-parameters*>}>;

Required Argument

PATH=*path-to-file*

specifies the path to the file to upload. The path must be fully qualified from a directory that the SAS client can access.

Optional Arguments

CASOUT= *output-table-options*
specifies the settings for a basic output table. The following parameters can be specified:

- CASLIB= "*string*"
- INDEXVARS={ "*variable-name-1*" <, "*variable-name-2,* " ...>}
- LABEL= "*string*"
- NAME= "*table-name*"
- PROMOTE=TRUE | FALSE
- REPLACE=TRUE | FALSE
- REPLICATION=*integer*

IMPORTOPTIONS={"BASESAS" | "CSV" | "DTA" | "ESP" | "EXCEL" | "FMT" | "HDAT" | "JMP" | "LASR" | "SPSS" | "XLSX"<, *fileType-specific-parameters*>}
specifies the file format and options. The value that you specify for the fileType parameter determines the other parameters that apply. For more information about the fileType parameter, see "importOptions" in the Upload action.

CSV files are saved with the session encoding. However, the encoding defaults to UTF-8 when you upload the file with the UPLOAD statement. To upload the saved file, you must provide the encoding that the file was created with.

You can dynamically pass the session encoding to the ENCODING= parameter by using the ENCODING function. The following IMPORTOPTION specification for CSV files uses the ENCODING function to return the session encoding string and pass it to the ENCODING= parameter: `importoptions={filetype="CSV", encoding=encoding()};`

Example: Using the UPLOAD Statement

The following example shows you to use the UPLOAD statement to transfer a CSV file from the SAS client to the server and perform basic programming tasks on the server-side in-memory table.

```
proc cas;
   session casauto;
   upload path="your-file-path\titanic3.csv"
      importoptions={filetype="CSV", encoding=encoding()}
      casout={name="Titanic01"};                                    /
*#1*/
run;
   table.tableInfo;                                                 /
*#2*/
run;
proc cas;
   simple.Summary result=Titanic02 / table={name="Titanic01"};      /
*#3*/
   print Titanic02;
```

```
   Titanic03=findTable(Titanic02);                                    /
*#4*/
   saveresult Titanic03 dataout=work.Titanic03;                       /
*#5*/
run;
```

1 Use the UPLOAD statement to transfer Titanic3.csv from the SAS client to the server. The IMPORTOPTION specification for the CSV file uses the ENCODING function to return the session encoding string and pass it to the ENCODING= parameter. The CASOUT option uses the name parameter to create a new table name.

2 Use the table.Tableinfo action to view the table you uploaded to the server. For more information, see tableInfo Action.

3 Use the simple.Summary action to generate descriptive statistics of numeric variables on the *Titanic01* table and save the results to *Titanic02*. For more information, see summary Action.

4 *Titanic03* is a new table which stores the result of the function, FINDTABLE. For more information, see FINDTABLE Function.

5 Use the SAVERESULT statement to save *Titanic03* as a SAS data set to the work directory. For more information, see SAVERESULT Statement.

Usage: CAS Procedure

Using: CAS Procedure

The PROC CAS step begins the interactive procedure, and it does not terminate until one of the following occurs:

- a DATA step
- another PROC step
- the QUIT statement
- an error condition that prevents the progress of the procedure.

Global statements, SAS macro code, and RUN statements do not terminate the CAS procedure.

Results: CAS Procedure

Result Table

When a CAS action returns a result, the result data type is usually a dictionary that contains one or more result tables. The result table is the primary means that a CAS action uses to return information to CASL. In addition to rows and columns, the result table also contains labels and data types. Any operation on a result table might create a new result table, and CASL includes syntax for creating your own result tables.

CASL offers a variety of operations on the CAS result table:

- You can reference particular row and column values within a result table.
- You can extract a row, column names, and types into a dictionary.
- You can subset a table by listing the rows and columns to be kept in the new table.
- You can use WHERE expression processing to create a new table with rows that match the WHERE expression.
- You can use the COMPUTE clause to create computed columns or to create an array with computed values from each row.
- You can iterate through the result that you receive from the submission of an action.

CASL enables you to create your own result table from the existing result table. It also enables you to create a subset or combination of other result tables.

See Also

"CASL Result Tables" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

Examples: CAS Procedure

Example 1: Setup Program for PROC CAS

Some of the examples in this book require that you download data and load the data into CAS. The following example uses the HTTP procedure and the PROC CAS UPLOAD statement to access data and load it into CAS.

Program

```
cas casauto;

%let data='http://support.sas.com/documentation/onlinedoc/viya/
exampledatasets/iris.csv';
filename t temp;
proc http method="get" url=&data. out=t;
run;
%let temppath = %sysfunc(quote(%sysfunc(pathname(t))));

proc cas;
   upload path=&temppath.
        casOut={
        name="iris"
        replace=True
       }
     importOptions={fileType="csv"}
;
run;
```

Example Code 2.4 *SAS Log*

```
NOTE: Cloud Analytic Services made the uploaded file available as table IRIS in
caslib CASUSER(sasdemo).
NOTE: The table IRIS has been created in caslib CASUSER(sasdemo) from binary data
uploaded to Cloud Analytic Services.
{caslib=CASUSER(sasdemo),tableName=IRIS}
```

Example 2: Run an Action

This example runs the specified action, in this case, the listNodes action and displays the contents of the table to the Output Delivery System (ODS).

Program

```
proc cas;                                  /* 1 */
  action listnodes result=res;             /* 2 */
  print res;                               /* 3 */
run;
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The ACTION statement runs the specified action, in this case, the listNodes action. The results of the listNodes action are stored in the variable named *Res*.

3 The PRINT statement displays the contents of the variable *Res*. The listNodes action returns a table and the PRINT statement supplies the table to the Output Delivery System (ODS) for display.

The following output shows sample results for eight machines.

Output 2.37 *Output: Partial Output for listNodes Action*

res: Results from builtins.listNodes

Name	Role	Connected	IP Address
cloud.example.com	controller	Yes	192.0.2.0
cloud.example.com	controller	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0
cloud.example.com	worker	Yes	192.0.2.0

Example 3: Creating a User-Defined Function

This example defines a new function and executes the function using the IF-ELSE/THEN, DO, and PRINT statements. This example also uses an internal function, *factorial*, to define and execute the newly created function *x*.

Program

```
proc cas;                                                    /*1*/
   function factorial(x);                                    /*2*/
     if (x < 1.0) then return(x);                            /*3*/
     else do;
        return exp(lgamma (x+1));
     end;                                                    /*4*/
   end;
run;
   do i = 1 to 9;                                            /*5*/
```

```
      x = factorial(i);
      print "   Factorial (" i ") = " put(x,best9.);   /* 6 */
   end;
   x = factorial(75);
   print "Factorial of 75 = " x;
run;
   x = factorial(75.0);
   print "Factorial of 75.0 = " x;
run;
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The FUNCTION statement creates a new function that takes one argument.

3 This function uses the **lgamma** and **exp** internal functions to calculate the factorial of a number. The IF statement asks whether the value of *x* is less than 1.0. The ELSE statement asks to return the run-time math expression function lgamma to be multiplied by the value of *x* and added 1.

4 The END statement ends the function processing.

5 The DO statement calls the function with arguments 1–9.

6 The PRINT statement prints the results.

Example Code 2.5 *Log Output: Define a New Function*

```
Factorial (1 ) =         1
Factorial (2 ) =         2
Factorial (3 ) =         6
Factorial (4 ) =        24
Factorial (5 ) =       120
Factorial (6 ) =       720
Factorial (7 ) =      5040
Factorial (8 ) =     40320
Factorial (9 ) =    362880
Factorial of 75= 2.480914E109
Factoiral of 75.0= 2.480914E109
```

Example 4: Using Run-time Math Functions with Inline Operator

Program

This example creates a new function and uses built-in run-time math functions to calculate the probability of the number of people in the same room who share the same birthday. This example uses common functions *lgamma* and *exp* to calculate

the factorial of a number. This example also uses the inline operator, which causes the given function to be called without a stack frame enabling the function to access the local variables in the current variable. The syntax, `inline.function-name`, calls the exact code within the code so that it runs during compilation.

```
proc cas;                                                    /* 1 */
   function SharedFeature (feature,number);                  /* 2 */
   p = exp(lgamma (feature+1)- lgamma                        /* 3 */
             (feature-number+1)-number*log(feature));
   return (1-p);
   end;
run;
   do n over {3 10 22 23 50 75};                             /* 4 */
   p =inline.SharedFeature(365,n);                           /* 5 */
   print "Chance at least 1 out of " put(n,best3.)
  " share same birthday = " put(p,best8.4);                  /* 6 */
   p =inline.SharedFeature(365,3);
   print "Chance at least 2 out of " put(n,best3.)
  " share same birthday = " put(p,best8.4);
end;
run;
quit;
```

1 The PROC CAS statement prepares the SAS client to execute statements that are unique to the CAS procedure and a selected subset of CASL statements.

2 The FUNCTION statement creates a new function. This function contains two arguments and calculates the probability that at least two items share the same feature given *number* of items and *features*.

3 This function uses the **`lgamma`**, **`log`**, and **`exp`** internal functions to calculate the probability.

4 The DO statement iterates over a list.

5 The variable *p* is defined here. You can use the inline operator to call the **`SharedFeature`** function to calculate the probability that at least two people have the same birthday, given the number of people in the room.

6 The PRINT statement prints the results. The PUT function applies the BEST. format.

Note: Functions can be recursive and they can execute any CASL statement. Functions can also be re-defined.

Example Code 2.6 *Log: Probability of 1 and 2 People Sharing the Same Birthday in the Same Room*

```
Chance at least 1 out of 3  share same birthday = 0.0082041659
Chance of at least 2 out of 3  share same birthday=0.0082041659
Chance at least 1 out of 10  share same birthday = 0.1169481777
Chance of at least 2 out of 10  share same birthday=0.0082041659
Chance at least 1 out of 22  share same birthday = 0.4756953077
Chance of at least 2 out of 22  share same birthday=0.0082041659
Chance at least 1 out of 23  share same birthday = 0.5072972343
Chance of at least 2 out of 23  share same birthday=0.0082041659
Chance at least 1 out of 50  share same birthday = 0.9703735796
Chance of at least 2 out of 50  share same birthday=0.0082041659
Chance at least 1 out of 75  share same birthday = 0.9997198782
Chance of at least 2 out of 75  share same birthday=0.0082041659
```

Example 5: Training a Decision Tree Model and Score Data

This example shows you how to train a decision tree model and score data generated by CASL actions.

Program

```
cas casauto;
libname mycas cas sessref=casauto;
data mycas.golf;                                    /*1*/
   format outlook     $8.;
   format temperature best10.;
   format humidity    best10.;
   format windy       $5.;
   format golf        $10.;
   input outlook $ 1-8 temperature humidity windy $ 16 - 21 golf $22 -
32;
datalines;
sunny    85 85 false Don't Play
sunny    80 90 true  Don't Play
overcast 83 78 false Play
rain     70 96 false Play
rain     68 80 false Play
rain     65 70 true  Don't Play
overcast 64 65 true  Play
sunny    72 95 false Don't Play
sunny    69 70 false Play
rain     75 80 false Play
sunny    75 70 true  Play
overcast 72 90 true  Play
overcast 81 75 false Play
```

```
rain     71 80 true  Don't Play
;
run;
filename score temp;                              /* 2 */
proc cas;
 decisionTree.dtreeTrain result=r /               /* 3 */
  table={name = "golf"}
   inputs={"outlook", "windy", "humidity", "temperature"}
   target="golf"
   maxlevel =4
   maxbranch=2
   nbins    =5
   binorder =1
   varImp   =true
   code={labelid=999, comment=true, tabForm=true};
run;
   print r['ModelInfo'];
   print r['DTreeVarImpInfo'];
run;
   saveresult r['CodeGen'] file=score;            /* 4 */
run;
quit;
data mycas.more_golf;                             /* 5 */
   format outlook     $8.;
   format temperature best10.;
   format humidity    best10.;
   format windy       $5.;
   *format golf       $10.;
   input outlook $ 1-8 temperature humidity windy $ 16 - 21 /* golf
$22 - 32 */;
datalines;
sunny    75 85 true
overcast 83 78 false
sunny    68 80 false
;
run;
proc cas;
  code = readfile("score");                       /* 6 */
 dscode =
   "data more_golf_scored;
     set more_golf;"
     || code ||
"run;";
run;
 dataStep.runCode / code = dscode;                /* 7 */
run;
 table.fetch / table = "more_golf_scored";        /* 8 */
run;
quit;

libname casuser cas caslib="casuser";
data casuser.golf;
  set mycas.golf;
run;

data casuser.more_golf;
```

```
  set mycas.more_golf;
run;
```

1 Generate, save, and run DATA step score code using CASL. The DATA step in CAS operates on CAS tables. Input and output data sets must use the CAS LIBNAME engine. The engine enables the DATA step to fetch column metadata for CAS tables when compiling the program. Data sets that use other engines must be loaded into CAS, or a caslib must be defined for that data source.

2 The FILENAME statement associates a SAS fileref with an external file. In this example, the FILENAME statement associates *score_golf.sas* with the *score* file, which is external.

3 The decision tree action set provides actions that can generate DATA step scoring code for decision tree models. The dtreeTrain action trains a decision tree. For more information about the syntax of dtreeTrain, see "Trains decision tree" in *SAS Visual Analytics: Programming Guide*

4 The SAVERESULT statement saves the DATA step score code to a SAS data set.

5 DATA step creates new data to score. Notice the commented out code in the DATA step, *golf* has been left out of the DATA step.

6 Use Readfile to read the contents of the file. The Readfile function reads the contents of the file given into the variable as a string.

7 The dataStep.runCode action is scoring new observations. The scored data includes a prediction for playing golf and a predicted probability.

8 Fetch rows from a table. Use the format parameter to apply formats to the variables. Use sortBy to specify the variables and variable settings for sorting results. Use the parameter to specify the ordinal position of the last row to return.

Output 2.38 *HTML Output: Trained Decision Tree*

rModelInfo: Results from decisionTree.dtreeTrain

Decision Tree for GOLF	
Number of Tree Nodes	3.000000
Max Number of Branches	2.000000
Number of Levels	2.000000
Number of Leaves	2.000000
Number of Bins	5.000000
Minimum Size of Leaves	5.000000
Maximum Size of Leaves	9.000000
Number of Variables	4.000000
Confidence Level for Pruning	0.250000
Number of Observations Used	14.000000
Misclassification Error (%)	28.571429

rDTreeVarImpInfo: Results from decisionTree.dtreeTrain

Variable Importance in Decision Tree Related Analytics			
Variable Name	Importance	Std Dev.	Count
outlook	0.9175	0	1.0000

Output 2.39 *HTML Output: New Data Set*

Results from dataStep.runCode

CAS Library	Name	Number of Rows	Number of Columns
CASUSER(casuser)	more_golf	3	4

Output CAS Tables			
CAS Library	Name	Number of Rows	Number of Columns
CASUSER(casuser)	more_golf_scored	3	7

Output 2.40 *HTML Output: Table Fetch Action*

Results from table.fetch

Selected Rows from Table MORE_GOLF_SCORED							
Index	outlook	temperature	humidity	windy	_leaf_id_	DT_golf	DT_golf_PredP
1	sunny	75	85	true	2	Don't Play	0.6
2	overcast	83	78	false	1	Play	0.7777777778
3	sunny	68	80	false	2	Don't Play	0.6

Example 6: Subsetting a Computed Column

This example uses CAS actions to retrieve rows from a SAS data set, computes a ratio for all rows, subsets the results, and saves the table to disk. The example uses CAS statements, actions, and the CASUTIL procedure.

Program

```
%let data='http://support.sas.com/documentation/onlinedoc/viya/
exampledatasets/cars.csv';
filename t temp;
proc http method="get" url=&data.
out=t;                                        /*1*/
run;
%let temppath = %sysfunc(quote(%sysfunc(pathname(t))));

proc cas;
   upload path=&temppath.
        casOut={
        name="cars"
        replace=True
       }
     importOptions={fileType="csv"}
;
run;
proc cas;
  table.recordCount
result=count /                                          /*2*/
  table="cars";
run;
table.fetch
result=fetchr/                                                 /*3*/
  table="cars"
   to=findtable(count)[1,1];
run;
lowMSRP=sort(findtable(fetchr), "MSRP")[1:5,
{"MODEL","MAKE","MSRP"}];      /*4*/
print lowMSRP;
run;
```

```
computedcolumn=findtable(fetchr).
   /*5*/
    compute({"ratio","Invoice/MSRP",best5.3},Invoice/MSRP)
    [,{"MODEL", "MAKE","MSRP","ratio"}];
subset= sort(computedcolumn,"ratio")
[1:5];                                  /*6*/
print subset;
run;
nrows=findtable(count)[1,1];
print"total rows:" nrows;
run;
saveresult lowMSRP
dataout=sasuser.lowMSRP;                                    /*7*/
saveresult subset dataout=sasuser.subset;
run;
quit;
```

1 Use the HTTP procedure and the PROC CAS UPLOAD statement to access the CARS data set and load it into CAS.

2 The recordCount action shows the number of rows in the CARS table and saves the record count in the variable *count*.

3 The fetch action fetches rows from the table CARS.

4 The assignment statement defines the variable *lowMSRP*. The variable *lowMSRP* contains five vehicles with the lowest MSRP.

5 The assignment statement defines the variable *computedcolumn*. The **compute** function calculates an invoice to MSRP ratio for all rows.

6 The assignment statement defines the variable *subset*. The **sort** function sorts and subsets the five lowest ratios. The total number of rows is displayed in the SAS log.

7 The SAVERESULT statement saves the CAS tables to disk. The output tables are saved as files.

Output 2.41 *LowMSRP Result Table*

lowMSRP: Results

Model	Make	MSRP
Rio 4dr manual	Kia	$10,280
Accent 2dr hatch	Hyundai	$10,539
Echo 2dr manual	Toyota	$10,760
Ion1 4dr	Saturn	$10,995
Rio 4dr auto	Kia	$11,155

Output 2.42 *Subset Result Table*

subset: Results

Model	Make	MSRP	Invoice/MSRP
911 Carrera 4S coupe 2dr (convert)	Porsche	$84,165	0.858
LX 470	Lexus	$64,800	0.871
SC 430 convertible 2dr	Lexus	$63,200	0.871
LS 430 4dr	Lexus	$55,750	0.871
GS 430 4dr	Lexus	$48,450	0.872

3

CASL Built-In Functions

Overview of CASL Functions

The CASL interface provides you with run-time support to create and manage values that the routine might return. All CASL functions work on the client-side, unless otherwise specified. All server-side functions are noted in the category. Functions can be defined in many ways:

- CASL supports built-in functions that provide run-time support for your CASL program.
- The FUNCTION statement can be used to create new functions, using the CASL syntax, to be called in expressions. For more information, see FUNCTION Statement.
- CASL also supports common SAS functions.

For a list of supported common SAS functions, see Common Functions.

Many CASL statements and functions can be executed client-side. Because these functions do not require interaction with the CAS server, they run as CASL scripts without the need to establish a CAS session. Functions that interact with CAS tables or perform distributed computations require an active CAS session to ensure

they are executed in the correct environment. Built-in CASL functions categorized as server-side functions on page 121 always require a CAS session to execute.

General CASL Built-In Function Syntax

The general CASL built-in function syntax has the following form:

function-name (*argument*, ...<*argument*>);

function-name
name of the CASL built-in function.

argument
a variable name, a constant, an expression, or another function. The number and type of arguments that CASL built-in functions allow are described with individual functions. Not all functions contain arguments. Multiple arguments are separated by a comma.

Note: If the value of an argument is invalid, an error occurs and the function's return expression is set to a missing or null value.

Function Exception Handling

You can define exception-handler functions. All of the exception-handler functions take exactly one parameter, which is a dictionary of information. Within the dictionary the context of the exception and parameters are specified.

This is an example of a handler for a system exception such as a floating-point error.

```
function myfphandler(env) do;
put "error detected, using_MISSING_instead";
setexceptvalue(_MISSING_);
resume;
end do;
on_fpexception call fphandler;
```

The example above is telling the handler when you find an error use "_MISSING_". If SAS finds an error or a floating-point error in your data, then it displays MISSING every time there is an error. Otherwise, it continues with the value that it is supposed to display.

Function Categories

Functions can be categorized by the types of values that they operate on. Each CASL function belongs to one of the following categories:

Table 3.1 *Function Category Descriptions*

Category	Description
Array	functions that operate on a named aggregate collection of homogenous data See Array for a list of functions.
Control	functions that specify how the program maintains program flow See Control for a list of functions.
Dictionary	functions that operate on a dictionary See Dictionary for a list of functions.
File IO	functions that are used to input and output data or results from the client or server side See File IO for a list of functions.
Function IO	functions that return information about the input and output values such as name, type, length, informats, formats, labels, and other value information. See Function IO for a list of functions.
Formatting	functions that modify the data type of a variable See Formatting for a list of functions.
Macros	functions that enable you to use macro variables within CASL See Macros for a list of functions.
Result Tables	functions that enable you to work with result tables See Result Tables for a list of functions.
Server-side	functions that can be used on the server side or during asynchronous sessions See Server-side for a list of functions.

Category	Description
Status	functions that operate on the execution state or process flow of a program at any time See Status for a list of functions.
Variable Information	functions that operate on variables and return names, types, lengths, informats, labels, and other variable information See Variable Information for a list of functions.

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

Category	Language Elements	Description
Array	ADDUNIQUE Function (p. 129)	Adds a value to an array, if the value does not already exist within the array.
	DIM Function (p. 156)	Retrieves the dimensions of an array or dictionary.
	SORT Function (p. 191)	Returns a list sorted in ascending order.
	SORT REV Function (p. 192)	Returns a list sorted in descending order.
	UNIQUE Function (p. 218)	Searches an array to determine if a value exists in the array.
Control	CANCELACTION Function (p. 131)	Cancels the action running on the given session.
	EXECUTE Function (p. 158)	Compiles and executes the given code.
	EXIT Function (p. 161)	Exits the block of CASL code with the given status.
	GETENV Function (p. 166)	Returns the environment variable for the given string.
	SLEEP Function (p. 190)	Sleep in the operating system (OS) for the number of seconds specified.
	TIMESTAMP Function (p. 214)	Returns the current date and time in a specified format.
	UUID Function (p. 222)	Returns the universally unique identifier (UUID) for the given session.
Dictionary	CASL2JSON Function (p. 133)	Converts the contents of a CASL dictionary to a string containing JSON formatted text.
	DICTIONARY Function (p. 155)	Searches for a key in the given dictionary and, if found, returns its associated value.
	DIM Function (p. 156)	Retrieves the dimensions of an array or dictionary.

Category	Language Elements	Description
	EXISTS Function (p. 159)	Determines whether a variable exists or whether a dictionary contains a specified key.
	GETKEYS Function (p. 167)	Extracts the keys from the dictionary into an array.
	GETVALUES Function (p. 170)	Extracts the values from the dictionary into an array.
	JSON2CASL Function (p. 175)	Converts a string containing JSON formatted text to a CASL dictionary.
File IO	READFILE Function (p. 184)	Reads and returns the contents of the specified file reference.
	READPATH Function (p. 185)	Reads and returns the contents of the specified filename.
Formatting	FMTVAR Function (p. 163)	Creates a CASL format variable that can be used as a substitute for a SAS format.
	INPUT_FORMAT Function (p. 171)	Converts the string to the best numeric value.
	PUT Function (p. 183)	Formats a value with the given format and returns it as a string.
Function IO	CLEAR Function (p. 140)	Clears all specified variables. If no variables are specified, all variables are cleared.
	EXISTS Function (p. 159)	Determines whether a variable exists or whether a dictionary contains a specified key.
Macros	SYMDEL Function (p. 194)	Deletes the specified macro variable.
	SYMGET Function (p. 197)	Returns the value of a macro variable.
	SYMPUT Function (p. 201)	Stores a value in a SAS macro variable.
	SYMPUTX Function (p. 205)	Assigns a value to a macro variable, and removes both leading and trailing blanks.
Result Tables	ADD_TABLE_ATTR Function (p. 124)	Adds attributes to a result table.
	ADDBYGROUP Function (p. 125)	Creates a new table from a BY-group table. The BY variables are added as the first variables of each row. The attributes for BY-group processing are removed.
	ADDROW Function (p. 127)	Adds one or more rows to a table.
	COMBINE_TABLES Function (p. 142)	Create a new table that has the name of the first table and contains all of the rows from all the tables. If this is a BY-group table, add the BY-group variables as the first variables of the table.

Category	Language Elements	Description
	FINDTABLE Function (p. 162)	Searches the given variable for the first table it sees. This is useful when a result from an action returns a copy of a result table.
	GETCOLUMN Function (p. 165)	Extracts the column into an array.
	GETRESULT Function (p. 168)	Retrieves results associated with ID or tag.
	LOC Function (p. 179)	Returns the row number in which the given value is found in the given column.
	NEWTABLE Function (p. 181)	Creates a new table.
	RESULT_BY_COL Function (p. 186)	Creates a new table with the given columns.
	RESULT_BY_TYPE Function (p. 187)	Creates a new table with columns that match the type specifications.
	TABCOLUMNS Function (p. 211)	Returns the names and labels of columns from a result table into a dictionary. For each item in the dictionary, the key is the name of a column, and the value is the label of the column.
	TABTYPES Function (p. 212)	Extracts the data types from a result table into a dictionary.
Server-side	CANCELACTIONS Function (p. 132)	Cancels the action running on the given list of sessions.
	CREATE_PARALLEL_SESSION Function (p. 152)	Starts a session with the same identity as the calling action. You can call this function to create multiple sessions.
	LIST_PARALLEL_SESSIONS Function (p. 179)	Lists parallel sessions.
	SEND_RESPONSE Function (p. 189)	Sends the specified result back to the client.
	TERM_PARALLEL_SESSION Function (p. 214)	Terminates a parallel session.
	WAIT_FOR_NEXT_ACTION Function (p. 224)	Wait for a completed action.
Status	CODETOSTATUS Function (p. 141)	Converts the action status into a dictionary.
	GETENV Function (p. 166)	Returns the environment variable for the given string.
	MEMORYSTATS Function (p. 180)	Displays CPU and memory statistics.

Category	Language Elements	Description
	SLEEP Function (p. 190)	Sleep in the operating system (OS) for the number of seconds specified.
	UUID Function (p. 222)	Returns the universally unique identifier (UUID) for the given session.
Variable Information	isType Function (p. 172)	Returns TRUE or FALSE based on whether an expression results in a value of a specified type.

Dictionary

ACTIONQ Function

Displays the action queue.

Syntax

Form 1: **ACTIONQ**(<"*session-reference* ">, <"yes">);

Without Arguments

You can specify the ACTIONQ function without arguments. If no arguments are specified, then the function returns the current, active session.

Optional Arguments

session-reference
specifies the name of the session in which actions are queued.

Note: If the session name is not provided then the current, active session is used.

Example "Example 1: Displaying the Action Queue" on page 123

yes
specifies the connection ID and the sequence number are displayed in the output.

Default no

Example "Example 2: Displaying the Action Queue with the Yes Argument" on page 123

Details

CASL enables you to submit an action asynchronously to the server. If the action is executing for the session in which it was submitted, then the action is placed in the action queue. You can submit as many actions as you need, but only one action executes at a time. The remaining actions execute in the order in which they were placed in the queue.

Examples

Example 1: Displaying the Action Queue

```
cas mySession1;
proc cas;
   session mysession1;
      listsession async="action1"; run;
      listsession async="action2"; run;
      listsession async="action3"; run;
      listsession async="action4"; run;
      print actionq("mysession1");  run;
quit;
```

Output 3.1 *Action Queue Results*

Timestamp	Action	Action Parameter Count	Action Tag	Action is Active
07/23/24 16	listsession	0	action1	Running
07/23/24 16	listsession	0	action2	Active
07/23/24 16	listsession	0	action3	Active
07/23/24 16	listsession	0	action4	Active

Note: The ASYNC= option enables you to submit multiple requests to the server. If your requests are in the same session, the requests are queued in the order received. If your requests are in separate sessions, the requests are executed in parallel.

Example 2: Displaying the Action Queue with the Yes Argument

```
cas mySession1;
proc cas;
   session mysession1;
      listsession async="action1"; run;
      listsession async="action2"; run;
```

```
        listsession async="action3"; run;
        listsession async="action4"; run;
        print actionq("mysession1", "yes");  run;
quit;
```

Output 3.2 *Action Queue Results*

Connection ID	Sequence Number	Timestamp	Action	Action Parameter Count	Action Tag	Action is Active
12195	93	07/23/24 16	listsession	0	action1	Running
12195	94	07/23/24 16	listsession	0	action2	Active
12195	95	07/23/24 16	listsession	0	action3	Active
12195	96	07/23/24 16	listsession	0	action4	Active

ADD_TABLE_ATTR Function

Adds attributes to a result table.

Category: Result Tables

Syntax

ADD_TABLE_ATTR(*table-name*, *dictionary-1*<, *dictionary-2*, ..., *dictionary-n*>)

Required Arguments

table-name
specifies the name of the table that the attributes are added to.

dictionary
specifies the dictionary that contains the attribute key-value pairs.

See "CASL Dictionaries" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

Example The following example specifies a dictionary literal:

```
add_table_attr(table,{attrA="valueA"})
```

Details

Result Tables

Result table attributes are any named key with an associated value that can be set on a CAS result table. You can use the CASL PRINT statement to view the table attributes. For an example that illustrates printing result table attributes, see "Result Table Properties" in *SAS Cloud Analytic Services: CASL Programmer's Guide*.

Return Values

The following are the possible returned data values for the ADD_TABLE_ATTR function:

0 The ADD_TABLE_ATTR function succeeded.

>0 The ADD_TABLE_ATTR function failed to add a table attribute.

Example

This example uses the ADD_TABLE_ATTR function to add a description of the data to the table attributes of the result table named `myTable`.

```
proc cas;
   /* Create a result table to work with*/
   myTable=newtable("myTable" /* Table Name */
                  ,{"Name","G1","G2"} /* Column names in order */
                  ,{"Varchar","int64","int64"} /* Column data types in
order */
                  ,{"Able" 74 84}    /* Row 1 values */
                  ,{"Baker" 74 65}   /* Row 2 values */
                  ,{"Charlie" 80 90} /* Row 3 values */);
   describe myTable;
   print;
   print(note) "Existing table attributes (none): " myTable.attrs;

   /*Modify table attributes - add a descripton of the data*/
   add_table_attr(myTable,{Description="Chemistry 101 Class Grades"});
   print;
   print(NOTE) "            New table attributes: " myTable.attrs;
run;
quit;
```

The program writes the following output to the log:

```
The program writes the following output to the log:
NOTE: The result table 'myTable' has been created.
[myTable] Table ( [3] Rows [3] columns
Column Names:
[1] Name             [Name            ] (varchar)
[2] G1               [G1              ] (int64)
[3] G2               [G2              ] (int64)
NOTE: Existing table attributes (none): {}
NOTE:             New table attributes: {Description=Chemistry 101 Class Grades}
96    quit;
```

ADDBYGROUP Function

Creates a new table from a BY-group table. The BY variables are added as the first variables of each row. The attributes for BY-group processing are removed.

Category: Result Tables

Returned data type: If the creation of a new table was not successful, the returned value is 0.

Syntax

ADDBYGROUP (*result-table-name*);

Required Argument

result-table-name
specifies the name of the BY-group table.

Example: Using ADDBYGROUP Function

Use the simple.Summary action to generate simple statistics for the *Iris* data set and use the groupBy option to create a BY-group table. Use the ADDBYGROUP function to create a new table from a BY-group table.

```
cas casauto;

proc casutil;
   load data=sashelp.iris
   casout="iris"
   outcaslib="casuser"
   replace;
run;

proc cas;
  simple.Summary result=summTab / table={name="iris"
groupBy="Species"};
run;
  print summTab;
run;
  NewTab=addbygroup(summTab."ByGroup1.Summary"n);
run;
  print NewTab;
run;
```

Output 3.3 *Results from Simple.Summary: summTab Results*

summTab: Results from simple.summary

Species=Setosa

Descriptive Statistics for IRIS

Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss	Skewness	Kurtosis
SepalLength	43.0000	58.0000	50	2503	50.0600	3.5249	0.4985	12.4249	7.0413	608.82	125909	100.42	<.0001	0	0.1201	-0.2527
SepalWidth	23.0000	44.0000	50	1714	34.2800	3.7906	0.5361	14.3690	11.0579	704.08	59460	63.95	<.0001	0	0.04117	0.9547
PetalLength	10.0000	19.0000	50	731	14.6200	1.7366	0.2456	3.0159	11.8785	147.78	10835	59.53	<.0001	0	0.1064	1.0216
PetalWidth	1.0000	6.0000	50	123	2.4600	1.0539	0.1490	1.1106	42.8397	54.4200	357.00	16.51	<.0001	0	1.2539	1.7191

summTab: Results from simple.summary

Species=Versicolor

Descriptive Statistics for IRIS																
Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss	Skewness	Kurtosis
SepalLength	49.0000	70.0000	50	2968	59.3600	5.1617	0.7300	26.6433	8.6956	1305.52	177486	81.32	<.0001	0	0.1054	-0.5330
SepalWidth	20.0000	34.0000	50	1385	27.7000	3.1380	0.4438	9.8469	11.3285	482.50	38847	62.42	<.0001	0	-0.3628	-0.3662
PetalLength	30.0000	51.0000	50	2130	42.6000	4.6991	0.6646	22.0816	11.0308	1082.00	91820	64.10	<.0001	0	-0.6065	0.04790
PetalWidth	10.0000	18.0000	50	663	13.2600	1.9775	0.2797	3.9106	14.9135	191.62	8983.00	47.41	<.0001	0	-0.03118	-0.4101

summTab: Results from simple.summary

Species=Virginica

Descriptive Statistics for IRIS																
Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss	Skewness	Kurtosis
SepalLength	49.0000	79.0000	50	3294	65.8800	6.3588	0.8993	40.4343	9.6521	1981.28	218990	73.26	<.0001	0	0.1180	0.03290
SepalWidth	22.0000	38.0000	50	1487	29.7400	3.2250	0.4561	10.4004	10.8439	509.62	44733	65.21	<.0001	0	0.3659	0.7061
PetalLength	45.0000	69.0000	50	2776	55.5200	5.5189	0.7805	30.4588	9.9405	1492.48	155616	71.13	<.0001	0	0.5494	-0.1538
PetalWidth	14.0000	25.0000	50	1013	20.2600	2.7465	0.3884	7.5433	13.5563	369.62	20893	52.16	<.0001	0	-0.1295	-0.6023

Output 3.4 *Results from ADDBYGROUP function: NewTab Results*

NewTab: Results from simple.summary

Species=Setosa

Descriptive Statistics for IRIS																
Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss	Skewness	Kurtosis
SepalLength	43.0000	58.0000	50	2503	50.0600	3.5249	0.4985	12.4249	7.0413	608.82	125909	100.42	<.0001	0	0.1201	-0.2527
SepalWidth	23.0000	44.0000	50	1714	34.2800	3.7906	0.5361	14.3690	11.0579	704.08	59460	63.95	<.0001	0	0.04117	0.9547
PetalLength	10.0000	19.0000	50	731	14.6200	1.7366	0.2456	3.0159	11.8785	147.78	10835	59.53	<.0001	0	0.1064	1.0216
PetalWidth	1.0000	6.0000	50	123	2.4600	1.0539	0.1490	1.1106	42.8397	54.4200	357.00	16.51	<.0001	0	1.2539	1.7191

ADDROW Function

Adds one or more rows to a table.

Category: Result Tables

Returned data type: CAS table

Syntax

ADDROW (*table*, *row-1*<, *row-2*, ... *row-N*>);

Required Argument

table

specifies the table to which rows are added. This argument can be any expression that results in a table.

Optional Argument

row-1, row-2, ..., row-N
specifies one or more lists of column values to add as rows to the table. Each row argument can be any expression that results in a list of values. The lists are treated as arrays and values within the lists are assigned to columns in the table row in increasing array index order.

Note Adding multiple rows in one function call is more efficient that specifying multiple function calls (one call for each row).

Details

Return Values

The following are the possible returned data values for the ADDROW function:

Table 3.2 *Return Values for the ADDROW Function*

Return	Description
1	the ADDROW function succeeded.
CAS table	the ADDROW function failed to add the row or rows to the table.

Examples

Example 1: Add Three Rows to the Result Table

The rows *job.result.tableName*, *job.result.caslib*, and *job.status.severity* are added to the table *results*.

```
addrow(results,{job.result.tableName, job.result.caslib,
job.status.severity});
```

Example 2: Create a New Table and Add Multiple Rows

This example shows how to create a table by using the NEWTABLE function and then add multiple rows to the table by using the ADDROW function. The program uses expressions as arguments for the row values in both the NEWTABLE and ADDROW functions. The example also shows how to efficiently add multiple rows in a single ADDROW function.

```
proc cas;

table_array[1] = newtable("Table1",{"x","y","z"},
{"double","double","double"});
```

```
table_array[2] = newtable("Table2",{"x","y","z"},
{"double","double","double"});

row_array = {
   { 1, 2, 3 },
   { 4, 5, 6 },
   { "x"=7, "z"=8, "y"=9 },
   { "z"=10, "y"=11, "x"=12 },
   { "y"=13, "z"=14, "x"=15 },
   { "x"=16, "z"=17, "y"=18 }
};

addrow( table_array[1], row_array[1] );
addrow( table_array[1], row_array[2] );
addrow( table_array[1], row_array[3] );

addrow( table_array[2], row_array[4] );
addrow( table_array[2], row_array[5] );
addrow( table_array[2], row_array[6] );

addrow( table_array[2], row_array[1], row_array[2], row_array[3] );

describe table_array[1];
print table_array[1];
run;

describe table_array[2];
print table_array[2];
run;

quit;
```

ADDUNIQUE Function

Adds a value to an array, if the value does not already exist within the array.

Category: Array

Returned data type: If the function successfully added the value to the array, then the Boolean value TRUE (1) is returned. Otherwise, FALSE is returned.

Syntax

ADDUNIQUE (*array*, *value*);

Required Arguments

array
specifies a countable number of values.

value
specifies a character or numeric constant, variable, or expression. If value is numeric, SAS converts the value to a character string using the BEST. format.

Example

This example uses the ADDUNIQUE function to add the value 4 to the array.

```
proc cas;
   a=addunique({1, 2, 3, 5}, 4);
   print a;
run;
```

```
1
```

base64Decode Function

Decodes a string, integer, double array, or a binary object to Base64.

Returned data type: string

Details

The BASE64DECODE function ignores all white space within the specified string.

Example

This example decodes the supplied base64-encoded string.

```
proc cas;
   a=base64Decode("SGVsbG8gV29ybGQhAA==");
   print a;
run;
```

Output 3.5 *Log Output*

```
[13] "0x48656c6c6f20576f726c642100"
```

See Also

- “base64Encode Function”

- "unpackData Function"

base64Encode Function

Encodes a string, integer, double array, or a binary object to Base64.

Syntax

base64Encode("*name*")

Required Argument

name
specifies the name of the variable or expression where the string, integer, double array, or a binary object needs to be encoded.

Example

This example uses the base64Encode function to encode a string to Base64.

```
proc cas;
   a=base64Encode("Hello World!");
   print a;
run;
```

Output 3.6 *Log Output*

```
SGVsbG8gV29ybGQhAA==
```

See Also

- "base64Decode Function"
- "unpackData Function"

CANCELACTION Function

Cancels the action running on the given session.

Category: Control

Interaction: This function works client-side as well.

Returned data type: If the function successfully canceled the action, then the Boolean value TRUE is returned. Otherwise, FALSE is returned.

Syntax

CANCELACTION (“*session-reference*”);

Required Argument

session-reference
specifies the session name on which the action is running.

Example

This example uses the CANCELACTION function to cancel the action running on the given session `mySession`.

```
proc cas;
    cancelaction("mySession");
run;
```

CANCELACTIONS Function

Cancels the action running on the given list of sessions.

Category: Server-side

Interaction: This function works client-side as well.

Syntax

CANCELACTIONS ({*list-of-session-references*});

Required Argument

session-reference
specifies the list of sessions on which the actions are running. You can specify multiple sessions.

Example

This example uses the CANCELACTION function to cancel the action running on the given sessions `casauto` and `mySession`.

```
proc cas;
    cancelactions({"casauto", "mySession"});
run;
```

CASL2JSON Function

Converts the contents of a CASL dictionary to a string containing JSON formatted text.

Category: Dictionary

Interaction: See "Interactions" on page 133.

Syntax

CASL2JSON(*dictionary*)

Required Argument

dictionary
specifies the dictionary to convert.

Details

Return Values

The following are the possible returned data values for the CASL2JSON function:

string a string containing JSON formatted text

FALSE the dictionary failed to be converted to a JSON string.

Interactions

Setting the `CASL_STD_JSON` environment variable and the STDJSON option affect whether the text that is generated by the CASL2JSON function conforms to the JSON standard.

By default, and for backward compatibility of existing code, the text generated by the CASL2JSON function might not conform to the JSON standard. Text that does not conform to the JSON standard might be problematic when processed by third-party applications. You can manage this non-conformant behavior in the following ways:

- by setting the STDJSON option in a CASL session. This is a local setting that affects only the environment in which the CASL program is executing, whether execution is on the SAS Compute Server (SAS client) or the CAS server.
  - To set the STDJSON option on the SAS client, specify `set stdjson` or `unset stdjson` in a PROC CAS code block:

    ```
    proc cas;
       set stdjson;
    run; quit;
    ```

    Setting the STDJSON argument in a program that is running on the SAS client affects only the behavior of the functions on the SAS client.
  - To set the STDJSON option on the CAS server, ensure that `set stdjson` or `unset stdjson` appears in the code submitted to the sccasl.runCasl action.

    Setting the STDJSON argument in a program that is running on the CAS server affects only the behavior of the functions on the CAS server.
- by setting the `CASL_STD_JSON` environment variable. The environment variable can be set on the SAS Compute Server (SAS client) or on the CAS server. Setting the environment variable in either of these environments has a global effect and applies to all CASL programs submitted for execution on each server.
  - To set the `CASL_STD_JSON` environment variable on the SAS Compute Server, add the environment variable to the `sas.compute.server: startup_commands` configuration instance for the Compute service in SAS Environment Manager.

    ```
    # to enable the JSON standard
          export CASL_STD_JSON = true
    # to disable the JSON standard
          export CASL_STD_JSON = false
    ```

    Setting the environment variable in the SAS Compute Server configuration instance (`sas.compute.server: startup_commands`) affects only CASL code that is running on the SAS client.
  - To set the `CASL_STD_JSON` environment variable on the CAS server, add the environment variable to the `sas.cas.instance.config: settings` configuration instance for the `cas-shared-default` service in SAS Environment Manager.

    ```
    # to enable the JSON standard
          export CASL_STD_JSON = true
    # to disable the JSON standard
          export CASL_STD_JSON = false
    ```

    Setting environment variable in the CAS server configuration instance (`sas.cas.instance.config: settings`) affects only the CASL code that is running on the CAS server.

  For more information about setting environment variables in configuration instances, see "Edit Server Configuration Instances" in *SAS Environment Manager: User's Guide*.

For more information about JSON, see RFC 8259.

When executing the CASL2JSON function and the JSON2CASL function in the same program and using the output of one function as the input to the other function, the STDJSON option should be set consistently:

- If the STDJSON option is *not set* when running the CASL2JSON function, then it should not be set when running the JSON2CASL function.
- If the STDJSON option *is* set when running the JSON2CASL function, then it should be set when running the CASL2JSON function.

Examples

Example 1: Convert Fetch Action Results to JSON

The following example converts the results of the Fetch action into JSON. The Fetch action results are stored as a dictionary in the variable FetchRes. The JSON is stored as a string in the variable **json1**.

```
cas casauto;                                  /* 1 */
proc casutil;                                 /* 2 */
   load data=sashelp.cars(obs=3
        keep=Make Model DriveTrain)
        casout="cars" outcaslib="casuser"
   replace;
run;

proc cas;
   table.fetch result=FetchRes /              /* 3 */
     table={name="cars",
            caslib="casuser"};
;
   describe FetchRes;                         /* 4 */
   json1 = casl2json(FetchRes);               /* 5 */
   describe json1;                            /* 6 */
   print json1;                               /* 7 */
run;
```

1 If necessary, create session CASAUTO.

2 Use PROC CASUTIL to transfer the SAS data set to CAS.

3 The Fetch action results are stored in the variable FetchRes as a dictionary.

4 The DESCRIBE statement prints the structure of the variable FetchRes to the SAS log.

5 The CASL2JSON function converts the summary results in the variable FetchRes to JSON. The JSON is saved in the variable Json1.

6 The DESCRIBE statement prints the structure of the variable Json1 to the SAS log.

7 The PRINT statement writes the contents of the variable Json1 to the SAS log. The variable Json1 contains the results from the Fetch action that are converted to JSON.

Example Code 3.1 *Fetch Action Results Converted to JSON*

```
 { "_cas_table_.Fetch" :  { "attributes" :  { "name" : "Fetch","label" : "Selected
Rows from Table CARS" }  , "metadata" :  [  {
"name" : "_Index_","type" : "int64" }  ,  { "name" : "Make","type" : "string" }  ,
{ "name" : "Model","type" : "string" }  ,  {
"name" : "DriveTrain","type" : "string" }  ]  , "rows" :  [  [ "1" , "Acura" , "
MDX           " , "All  " ]  ,  [ "2" , "Acura" ,
" RSX Type S 2dr" , "Front" ]  ,  [ "3" , "Acura" , " TSX 4dr       " ,
"Front" ]  ]  }  }
```

Example 2: Convert Summary Action Results to JSON

The following example converts the results of the Summary action into JSON. The Summary action results are stored as a dictionary in the variable SummRes. The JSON is stored as a string in the variable **json2**.

```
cas casauto;                                    /*1*/
proc casutil;                                   /*2*/
   load data=sashelp.cars
      casout="cars" outcaslib="casuser"
   replace;
run;

proc cas;
   simple.summary result=SummRes /              /*3*/
      table={name="cars",caslib="casuser"}
      ,inputs={"MPG_City","MPG_Highway"}
      ,subSet={"NMISS","MIN", "MEAN", "MAX"}
    ;
   describe SummRes;                            /*4*/
   json2 = casl2json(SummRes);                  /*5*/
   describe json2;                              /*6*/
   print json2;                                 /*7*/
run;
```

1 If necessary, create session CASAUTO.

2 Use PROC CASUTIL to transfer the SAS data set to CAS.

Use the CAS engine to transfer data to the CAS server.

3 The Summary action results are saved in the variable SummRes as a dictionary.

4 The DESCRIBE statement prints the structure of the variable SummRes to the SAS log.

5 The CASL2JSON function converts the summary results in the variable SummRes to JSON. The JSON is saved in the variable Json2.

6 The DESCRIBE statement prints the structure of the variable Json2 to the SAS log.

7 The PRINT statement writes the contents of the variable Json2 to the SAS log. The variable Json2 contains the results from the Summary action that are converted to JSON.

Example Code 3.2 *Summary Action Results Converted to JSON*

```
 { "_cas_table_.Summary" :  { "attributes" :  { "name" : "Summary","label" :
"Descriptive Statistics for CARS" }  , "metadata" :  [
 { "name" : "Column","label" : "Analysis Variable","type" : "string" }  ,
{ "name" : "Min","label" : "Minimum","format" :
"D8.4","type" : "double" }  ,  { "name" : "Max","label" : "Maximum","format" :
"D8.4","type" : "double" }  ,  { "name" :
"NMiss","label" : "N Miss","format" : "BEST10.","type" : "double" }  ,  { "name" :
"Mean","label" : "Mean","format" : "D8.4","type"
: "double" }  ]  , "rows" :  [  [ "MPG_City   " , "10" , "60" , "0" ,
"20.060747664" ]  ,  [ "MPG_Highway" , "12" , "66" , "0" ,
"26.843457944" ]  ]  }  }
```

Example 3: Convert Freq Action Results to JSON

The following example converts the results of the Freq action into JSON. The Freq action results are stored as a dictionary in the variable FreqRes. The converted JSON is stored as a string in the variable **json3**.

```
cas casauto;                                        /*1*/
proc casutil;                                       /*2*/
   load data=sashelp.iris
      casout="iris" outcaslib="casuser"
   replace;
run;


proc cas;
   simple.freq result=FreqRes /
      inputs={"Species"},
      table={name="iris", caslib="casuser"};        /*3*/
   describe FreqRes;                                /*4*/
   json3 = casl2json(FreqRes);                      /*5*/
   describe json3;                                  /*6*/
   print json3;                                     /*7*/
run;
```

1 If necessary, create session CASAUTO.

2 Use PROC CASUTIL to transfer the SAS data set to CAS.

3 The Freq action results are a dictionary saved in the variable FreqRes.

4 The DESCRIBE statement prints the structure of the variable FreqRes to the SAS log.

5 The CASL2JSON function converts the dictionary in the variable FreqRes to JSON. The JSON string is saved in the variable Json3.

6 The DESCRIBE statement prints the structure of the variable Json3 to the SAS log.

7 The PRINT statement writes the contents of the variable Json3 to the SAS log. The variable Json3 contains the results from the Freq action that are converted to JSON.

Example Code 3.3 *Freq Action Results Converted to JSON*


```
 { "_cas_table_.Frequency" :  { "attributes" :  { "name" : "Frequency","label" :
"Frequency for IRIS" }  , "metadata" :  [  {
"name" : "Column","label" : "Analysis Variable","type" : "varchar" }  ,  { "name" :
"CharVar","label" : "Character Value","type" :
"varchar" }  ,  { "name" : "FmtVar","label" : "Formatted Value","type" :
"varchar" }  ,  { "name" : "Level","label" :
"Level","format" : "BEST12.","type" : "int64" }  ,  { "name" : "Frequency","label" :
"Frequency","format" : "BEST12.","type" :
"double" }  ]  , "rows" :  [  [ "Species" , "Setosa    " , "Setosa    " , "1" ,
"50" ]  ,  [ "Species" , "Versicolor" ,
"Versicolor" , "2" , "50" ]  ,  [ "Species" , "Virginica " , "Virginica " , "3" ,
"50" ]  ]  }  }
```


See Also

- "SET Statement" on page 90
- "UNSET Statement" on page 97
- "CAS LIBNAME Engine " in *SAS Cloud Analytic Services: User's Guide*
- "DESCRIBE Statement" on page 36
- "Fetch rows" in *SAS Viya Platform: System Programming Guide*
- "Frequency" in *SAS Visual Analytics: Programming Guide*
- "JSON2CASL Function" on page 175
- "PRINT Statement" on page 79
- "Summary" in *SAS Visual Analytics: Programming Guide*

CASLSTORE Function

Creates a caslstore value.

Syntax

Form 1: **CASLSTORE**({ CASLIB= *caslib-reference-name*, NAME= *caslstore-table-name*} <, { CASLIB=*caslib-reference-name-2*, NAME= *caslstore-table-name-2*}... , {CASLIB= *caslib-reference-name-n*, NAME=*caslstore-table-name-n*}>);

Form 2: **CASLSTORE**({PATH=*pathname*} <, {PATH=*pathname-2*}, ..., {PATH=*pathname-n*}>);

Required Arguments

caslib-reference-name
specifies the caslib that the caslstore is stored in.

caslstore-table-name
specifies the name of the caslstore table.

pathname
specifies the path that the caslstore is stored in.

Details

A caslstore is a CASL code repository for storing user-defined functions on the CAS server. The CASLSTORE function makes the user-defined functions that have been uploaded to CAS using the UPLOAD_CASLSTORE function available for use in your program. The caslstore does not persist after your program runs and must be defined inside every program that uses your user-defined functions each time the program is executed.

Example

```
proc cas;
   source funcs;
      function c2f(c);
          /*(°Cx9/5)+32*/
          return ((c*9/5)+32);
      end;
      function f2c(f);
          /* (°F-32)x5/9 */
          return ((f-32)*5/9);
      end;
   endsource;
   upload_caslstore({caslib="casuser",
name="store1", replace=true}, funcs);
run;

proc cas;
  cs = caslstore({caslib="casuser", name="store1"});
  DegF=37.0;
  DegC=37.0;
  f=round(c2f(DegC),.1);
  c=round(f2c(DegF),.1);
  print "NOTE: " degF "°F is " c "°C.";
  print "NOTE: " degC "°C is " f "°F.";
run;
```

Example Code 3.4 *Log Output:*

```
NOTE: 37 °F is 2.8 °C.
NOTE: 37 °C is 98.6 °F.
```

See Also

- "DEFAULT_CASLSTORE Function"
- "UPLOAD_CASLSTORE Function"

CLEAR Function

Clears all specified variables. If no variables are specified, all variables are cleared.

Category: Function IO

Syntax

CLEAR (<*variable-1*<, *variable-2* ..., *variable-n*>>);

Optional Argument

variable
a string that specifies a variable name.

Details

Return Values

The following are the possible returned data values for the CLEAR function:

0 The variables were cleared.

Example

This example uses the CLEAR function to clear the specified variables `a` and `b`.

```
proc cas;
   a=1;
   b=2;
   c=clear("a", "b");
   print c;
run;
```

```
0
```

CODETOSTATUS Function

Converts the action status into a dictionary.

Category: Status

Returned data type: Returns the value of 0 for failure.

Syntax

CODETOSTATUS (*status_code*);

Required Argument

status_code
The number that represents the status from the server.

Details

The details that are returned in the dictionary are values that are derived from the status code. The following information is included in the dictionary:

Status
CAS returns the status as a number.

0
failure

Group
The message file where the status is defined. The number is between 0–1000.

Number
The number of the message file.

Msg
The status code is converted to a string and used in the message file.

Example: Using CODETOSTATUS Function

The following example shows you how to use the CODETOSTATUS function to retrieve an action status and put the status in a dictionary. The CODETOSTATUS function requires a status code to retrieve the information. The information that the status code retrieves is status, group, message number, and message.

```
proc cas;
```

```
  x=codetostatus(5280316);
  describe x;
  print put(x, hex8.);
run;
```

The following is printed to the SAS log:

Example Code 3.5 *SAS Log*

```
dictionary ( 4 entries, 4 used);
[status] int64_t;
[group] int64_t;
[number] int64_t;
[msg] string;
{status=90BFC13C,group=00000210,number=0000013C,msg=The action has been cancelled.}
```

See Also

"EXIT Function"

COMBINE_TABLES Function

Create a new table that has the name of the first table and contains all of the rows from all the tables. If this is a BY-group table, add the BY-group variables as the first variables of the table.

Category: Result Tables

Syntax

COMBINE_TABLES (*list-of-tables*, <*"key-pattern"*>);

Required Argument

list-of-tables
The list of tables to combine.

Optional Argument

key-pattern
matches against the name of the table that are sent to the *list-of-tables* arguments. If a key pattern is specified, then the key pattern keeps only the tables whose keys match the pattern within the tables' keys.

Note: ByGroupInfo tables are ignored.

Details

Return Values

The following are the possible returned data values for the COMBINE_TABLES function:

CAS result table	If the function is successful, the returned data type is the CAS result table.
0	If the function is not successful, the returned data type of 0 is too many arguments.
3	If the function is not successful, the returned data type of 3 is unable to combine tables.

Examples

Example 1: Using the COMBINE_TABLES Function

Use the simple.mdSummary action to generate descriptive summary statistics for the columns in the cars data set and use the groupBy parameter to save the results to separate tables by the unique combinations of the variables Origin and Type. Use the COMBINE_TABLES function to create a new table that contains all of the rows from all the tables.

```
libname casuser cas caslib="casuser";
data casuser.cars;                                    /* 1 */
     set sashelp.cars;
run;

cas casauto;

proc cas;
   simple.mdSummary result=summTab /                  /* 2 */
      table={caslib="casuser",
             name="cars",
             groupBy={"origin", "type"}
      };
   describe summTab;                                  /* 3 */
   title "Results of simple.mdSummary action";
   print summTab;                                     /* 4 */
run;
   combTable = combine_tables(summTab);               /* 5 */
   title "Results of combine_tables function";
   print combTable;                                   /* 6 */
   saveresult combTable dataout=work.combTable;       /* 7 */
run;
quit;

title "Combined Table";
proc print data=work.combTable;                       /* 8 */
```

```
run;
```

1. Load the cars table from SASHELP into the Casuser library.
2. Use simple.mdSummary to compute summary statistics by each unique combination of the group by variables Origin and Type, and store the result in the variable named summTab.
3. Use the DESCRIBE statement to view the data type and values of summTab. In the log, the output of the DESCRIBE statement shows that summTab is a dictionary with 16 entries where each entry is a table. The first table is the ByGroupInfo table.
4. Use the PRINT statement to view the computed group by tables. The results show 15 tables containing summary statistics by the unique combinations of Origin and Type.
5. Use the COMBINE_TABLES function with the summTab variable as the argument to combine the tables. Save the result as combTable.
6. Use the PRINT statement to view combTable. The result shows that all 15 tables have been combined except for the first table in the summTab dictionary where the key contains ByGroupInfo.
7. Use saveresult to save combTable as work.combTable.
8. Use PROC PRINT on work.combTable to print the combined table containing all rows from all tables in summTab (excluding the ByGroupInfo table), which now includes columns for Origin and Type.

Output 3.7 *Partial Log Output from the DESCRIBE Statement*

```
dictionary ( 16 entries, 16 used);
 [ByGroupInfo] Table ( [15] Rows [5] columns
Column Names:
 [1] Origin           [                ] (char)
 [2] Origin_f         [                ] (char)
 [3] Type             [                ] (char)
 [4] Type_f           [                ] (char)
 [5] _key_            [                ] (varchar)
 [ByGroup1.MDSummary] Table[MDSummary] ( [10] Rows [15] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] Min              [Minimum         ] (double) [D8.4]
 [3] Max              [Maximum         ] (double) [D8.4]
 [4] N                [N               ] (double) [BEST10.]
 [5] NMiss            [N Miss          ] (double) [BEST10.]
 [6] Mean             [Mean            ] (double) [D8.4]
 [7] Sum              [Sum             ] (double) [BEST10.]
 [8] Std              [Std Dev         ] (double) [D8.4]
 [9] StdErr           [Std Error       ] (double) [D8.4]
 [10] Var              [Variance        ] (double) [D8.4]
 [11] USS              [USS             ] (double) [D8.4]
 [12] CSS              [Corrected SS    ] (double) [D8.4]
 [13] CV               [Coeff of Variation] (double) [D8.4]
 [14] TValue           [t Value         ] (double) [7.2]
 [15] ProbT            [Pr > |t|        ] (double) [PVALUE6.4]
 [ByGroup2.MDSummary] Table[MDSummary] ( [10] Rows [15] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] Min              [Minimum         ] (double) [D8.4]
 [3] Max              [Maximum         ] (double) [D8.4]
 [4] N                [N               ] (double) [BEST10.]
 [5] NMiss            [N Miss          ] (double) [BEST10.]
 [6] Mean             [Mean            ] (double) [D8.4]
 [7] Sum              [Sum             ] (double) [BEST10.]
 [8] Std              [Std Dev         ] (double) [D8.4]
 [9] StdErr           [Std Error       ] (double) [D8.4]
 [10] Var              [Variance        ] (double) [D8.4]
 [11] USS              [USS             ] (double) [D8.4]
 [12] CSS              [Corrected SS    ] (double) [D8.4]
 [13] CV               [Coeff of Variation] (double) [D8.4]
 [14] TValue           [t Value         ] (double) [7.2]
 [15] ProbT            [Pr > |t|        ] (double) [PVALUE6.4]
.
.
.
.
.
```

Output 3.8 *Partial Results from the First PRINT Statement*

Results of simple.mdSummary action

summTab: Results from simple.mdSummary

Origin=Asia Type=Hybrid

Descriptive Statistics for CARS

Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss
MSRP	19110	20510	3	59760	19920	725.47	418.85	526300	3.6419	1052600	1.1915E9	47.56	0.0004	0
Invoice	17911	18926	3	55288	18429	507.85	293.21	257908	2.7556	515817	1.0194E9	62.85	0.0003	0
EngineSize	1.4000	2.0000	3	4.9	1.6333	0.3215	0.1856	0.1033	19.6809	0.2067	8.2100	8.80	0.0127	0
Cylinders	3.0000	4.0000	3	11	3.6667	0.5774	0.3333	0.3333	15.7459	0.6667	41.0000	11.00	0.0082	0
Horsepower	73.0000	110.00	3	276	92.0000	18.5203	10.6927	343.00	20.1307	686.00	26078	8.60	0.0132	0
MPG_City	46.0000	60.0000	3	165	55.0000	7.8102	4.5092	61.0000	14.2005	122.00	9197.00	12.20	0.0067	0
MPG_Highway	51.0000	66.0000	3	168	56.0000	8.6603	5.0000	75.0000	15.4647	150.00	9558.00	11.20	0.0079	0
Weight	1850.00	2890.00	3	7472	2490.67	560.43	323.56	314081	22.5012	628163	19238424	7.70	0.0165	0
Wheelbase	95.0000	106.00	3	304	101.33	5.6862	3.2830	32.3333	5.6114	64.6667	30870	30.87	0.0010	0
Length	155.00	175.00	3	505	168.33	11.5470	6.6667	133.33	6.8596	266.67	85275	25.25	0.0016	0

Results of simple.mdSummary action

summTab: Results from simple.mdSummary

Origin=Asia Type=Sedan

Descriptive Statistics for CARS

Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss

Output 3.9 *Partial Results from the Second PRINT Statement*

Results of combine_tables function

combTable: Results from simple.mdSummary

Origin=Asia Type=Hybrid

MDSummary

Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss
MSRP	19110	20510	3	59760	19920	725.47	418.85	526300	3.6419	1052600	1.1915E9	47.56	0.0004	0
Invoice	17911	18926	3	55288	18429	507.85	293.21	257908	2.7556	515817	1.0194E9	62.85	0.0003	0
EngineSize	1.4000	2.0000	3	4.9	1.6333	0.3215	0.1856	0.1033	19.6809	0.2067	8.2100	8.80	0.0127	0
Cylinders	3.0000	4.0000	3	11	3.6667	0.5774	0.3333	0.3333	15.7459	0.6667	41.0000	11.00	0.0082	0
Horsepower	73.0000	110.00	3	276	92.0000	18.5203	10.6927	343.00	20.1307	686.00	26078	8.60	0.0132	0
MPG_City	46.0000	60.0000	3	165	55.0000	7.8102	4.5092	61.0000	14.2005	122.00	9197.00	12.20	0.0067	0
MPG_Highway	51.0000	66.0000	3	168	56.0000	8.6603	5.0000	75.0000	15.4647	150.00	9558.00	11.20	0.0079	0
Weight	1850.00	2890.00	3	7472	2490.67	560.43	323.56	314081	22.5012	628163	19238424	7.70	0.0165	0
Wheelbase	95.0000	106.00	3	304	101.33	5.6862	3.2830	32.3333	5.6114	64.6667	30870	30.87	0.0010	0
Length	155.00	175.00	3	505	168.33	11.5470	6.6667	133.33	6.8596	266.67	85275	25.25	0.0016	0
MSRP	1028[illegible]	[illegible]750	94	2139813	22764	9613.14	991.52	92412548	42.2297	8.5944E9	5.731E10	22.96	<.0001	0

Output 3.10 *Partial Results from the PRINT Procedure*

Combined Table

Obs	Origin	Type	Column	Min	Max	N	NMiss	Mean	Sum	Std	StdErr	Var	USS	CSS	CV	TValue	ProbT
1	Asia	Hybrid	MSRP	19110	20510	3	0	19920	59760	725.47	418.85	526300	1.1915E9	1052600	3.6419	47.56	0.0004
2	Asia	Hybrid	Invoice	17911	18926	3	0	18429	55288	507.85	293.21	257908	1.0194E9	515817	2.7556	62.85	0.0003
3	Asia	Hybrid	EngineSize	1.4000	2.0000	3	0	1.6333	4.9	0.3215	0.1856	0.1033	8.2100	0.2067	19.6809	8.80	0.0127
4	Asia	Hybrid	Cylinders	3.0000	4.0000	3	0	3.6667	11	0.5774	0.3333	0.3333	41.0000	0.6667	15.7459	11.00	0.0082
5	Asia	Hybrid	Horsepower	73.0000	110.00	3	0	92.0000	276	18.5203	10.6927	343.00	26078	686.00	20.1307	8.60	0.0132
6	Asia	Hybrid	MPG_City	46.0000	60.0000	3	0	55.0000	165	7.8102	4.5092	61.0000	9197.00	122.00	14.2005	12.20	0.0067
7	Asia	Hybrid	MPG_Highway	51.0000	66.0000	3	0	56.0000	168	8.6603	5.0000	75.0000	9558.00	150.00	15.4647	11.20	0.0079
8	Asia	Hybrid	Weight	1850.00	2890.00	3	0	2490.67	7472	560.43	323.56	314081	19238424	628163	22.5012	7.70	0.0165
9	Asia	Hybrid	Wheelbase	95.0000	106.00	3	0	101.33	304	5.6862	3.2830	32.3333	30870	64.6667	5.6114	30.87	0.0010
10	Asia	Hybrid	Length	155.00	175.00	3	0	168.33	505	11.5470	6.6667	133.33	85275	266.67	6.8596	25.25	0.0016
11	Asia	Sedan	MSRP	10280	55750	94	0	22764	2139813	9613.14	991.52	92412548	5.731E10	8.5944E9	42.2297	22.96	<.0001
12	Asia	Sedan	Invoice	9875.00	48583	94	0	20788	1954101	8363.51	862.63	69948245	4.713E10	6.5052E9	40.2318	24.10	<.0001

Example 2: Using the COMBINE_TABLES Function and Specifying a Key Pattern

Use the simple.mdSummary action to generate descriptive summary statistics for the columns in the cars data set and use the groupBy parameter to save the results to separate tables by the unique combinations of the variables Origin and Type. Use the COMBINE_TABLES function to create a new table that contains only the rows from the tables with keys matching the pattern "2".

```
libname casuser cas caslib="casuser";

data casuser.cars;                                          /* 1 */
     set sashelp.cars;
run;

cas casauto;

proc cas;
   simple.mdSummary result=summTab /                        /* 2 */
      table={caslib="casuser",
             name="cars",
             groupBy={"origin", "type"}
      };
   describe summTab;                                        /* 3 */
   title "Results of the simple.mdSummary action";
   print summTab;                                           /* 4 */
run;
   combTable = combine_tables(summTab, "2");                /* 5 */
   title "Results of the combine_tables function";
   print combTable;                                         /* 6 */
   saveresult combTable dataout=work.combTable;             /* 7 */
run;
quit;

title "Combined Table";
proc print data=work.combTable;                             /* 8 */
run;
```

1 Load the cars table from SASHELP into the Casuser library.

2 Use simple.mdSummary to compute summary statistics by each unique combination of the group by variables Origin and Type, and store the result in the variable named summTab.

3 Use the DESCRIBE statement to view the data type and values of summTab. In the log, the output of the DESCRIBE statement shows that summTab is a dictionary with 16 entries where each entry is a table. The first table is the ByGroupInfo table.

4 Use the PRINT statement to view the computed group by tables. The results show 15 tables containing summary statistics by the unique combinations of Origin and Type.

5 Use the COMBINE_TABLES function with the summTab variable as the argument to combine the tables. The second argument is specified as the key pattern "2" so that it keeps only the tables whose keys match the pattern "2" within the tables' keys. This includes the tables that have the keys 2 and 12 (ByGroup2.MDSummary and ByGroup12.MDSummary). ByGroupInfo tables are ignored when a key pattern is specified.

6 Use the PRINT statement to view combTable. The result shows that the two tables whose keys contain 2 and 12 have been combined.

7 Use saveresult to save combTable as work.combTable.

8 Use PROC PRINT on work.combTable to print the combined table, which includes columns for Origin and Type and the rows from the tables with the keys 2 and 12 (ByGroup2.MDSummary and ByGroup12.MDSummary).

Output 3.11 *Partial Results from the First PRINT Statement*

Results of the simple.mdSummary action

summTab: Results from simple.mdSummary

Origin=Asia Type=Hybrid

Descriptive Statistics for CARS

Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss
MSRP	19110	20510	3	59760	19920	725.47	418.85	526300	3.6419	1052600	1.1915E9	47.56	0.0004	0
Invoice	17911	18926	3	55288	18429	507.85	293.21	257908	2.7556	515817	1.0194E9	62.85	0.0003	0
EngineSize	1.4000	2.0000	3	4.9	1.6333	0.3215	0.1856	0.1033	19.6809	0.2067	8.2100	8.80	0.0127	0
Cylinders	3.0000	4.0000	3	11	3.6667	0.5774	0.3333	0.3333	15.7459	0.6667	41.0000	11.00	0.0082	0
Horsepower	73.0000	110.00	3	276	92.0000	18.5203	10.6927	343.00	20.1307	686.00	26078	8.60	0.0132	0
MPG_City	46.0000	60.0000	3	165	55.0000	7.8102	4.5092	61.0000	14.2005	122.00	9197.00	12.20	0.0067	0
MPG_Highway	51.0000	66.0000	3	168	56.0000	8.6603	5.0000	75.0000	15.4647	150.00	9558.00	11.20	0.0079	0
Weight	1850.00	2890.00	3	7472	2490.67	560.43	323.56	314081	22.5012	628163	19238424	7.70	0.0165	0
Wheelbase	95.0000	106.00	3	304	101.33	5.6862	3.2830	32.3333	5.6114	64.6667	30870	30.87	0.0010	0
Length	155.00	175.00	3	505	168.33	11.5470	6.6667	133.33	6.8596	266.67	85275	25.25	0.0016	0

Results of the simple.mdSummary action

summTab: Results from simple.mdSummary

Origin=Asia Type=Sedan

Output 3.12 *Partial Results from the Second PRINT Statement*

Results of the combine_tables function

combTable: Results from simple.mdSummary

Origin=Asia Type=Sedan

MDSummary

Column	Minimum	Maximum	N	Sum	Mean	Std Dev	Std Error	Variance	Coeff of Variation	Corrected SS	USS	t Value	Pr > |t|	N Miss
MSRP	10280	55750	94	2139813	22764	9613.14	991.52	92412548	42.2297	8.5944E9	5.731E10	22.96	<.0001	0
Invoice	9875.00	48583	94	1954101	20788	8363.51	862.63	69948245	40.2318	6.5052E9	4.713E10	24.10	<.0001	0
EngineSize	1.5000	4.5000	94	248.9	2.6479	0.7790	0.08035	0.6068	29.4194	56.4346	715.49	32.96	<.0001	0
Cylinders	4.0000	8.0000	94	474	5.0426	1.1632	0.1200	1.3530	23.0675	125.83	2516.00	42.03	<.0001	0
Horsepower	103.00	340.00	94	17106	181.98	57.2929	5.9093	3282.47	31.4833	305270	3418198	30.80	<.0001	0
MPG_City	16.0000	36.0000	94	2147	22.8404	4.9390	0.5094	24.3936	21.6239	2268.61	51307	44.84	<.0001	0
MPG_Highway	22.0000	44.0000	94	2817	29.9681	4.8846	0.5038	23.8592	16.2993	2218.90	86639	59.48	<.0001	0
Weight	2035.00	4802.00	94	297169	3161.37	584.29	60.2654	341400	18.4823	31750244	9.7121E8	52.46	<.0001	0
Wheelbase	93.0000	124.00	94	9931	105.65	6.4068	0.6608	41.0475	6.0643	3817.41	1053017	159.88	<.0001	0
Length	154.00	204.00	94	17297	184.01	10.4506	1.0779	109.21	5.6793	10157	3192989	170.71	<.0001	0
MSRP	18345	81795	9	407315	45257	21279	7093.15	4.5281E8	47.0189	3.6225E9	2.206E10	6.38	0.0002	0
		74451	9	370302	41145	19430	6476.54	3.7751E8	47.2227					

Output 3.13 *Partial Results from the PRINT Procedure*

Combined Table

Obs	Origin	Type	Column	Min	Max	N	NMiss	Mean	Sum	Std	StdErr	Var	USS	CSS	CV	TValue	ProbT
1	Asia	Sedan	MSRP	10280	55750	94	0	22764	2139813	9613.14	991.52	92412548	5.731E10	8.5944E9	42.2297	22.96	<.0001
2	Asia	Sedan	Invoice	9875.00	48583	94	0	20788	1954101	8363.51	862.63	69948245	4.713E10	6.5052E9	40.2318	24.10	<.0001
3	Asia	Sedan	EngineSize	1.5000	4.5000	94	0	2.6479	248.9	0.7790	0.08035	0.6068	715.49	56.4346	29.4194	32.96	<.0001
4	Asia	Sedan	Cylinders	4.0000	8.0000	94	0	5.0426	474	1.1632	0.1200	1.3530	2516.00	125.83	23.0675	42.03	<.0001
5	Asia	Sedan	Horsepower	103.00	340.00	94	0	181.98	17106	57.2929	5.9093	3282.47	3418198	305270	31.4833	30.80	<.0001
6	Asia	Sedan	MPG_City	16.0000	36.0000	94	0	22.8404	2147	4.9390	0.5094	24.3936	51307	2268.61	21.6239	44.84	<.0001
7	Asia	Sedan	MPG_Highway	22.0000	44.0000	94	0	29.9681	2817	4.8846	0.5038	23.8592	86639	2218.90	16.2993	59.48	<.0001
8	Asia	Sedan	Weight	2035.00	4802.00	94	0	3161.37	297169	584.29	60.2654	341400	9.7121E8	31750244	18.4823	52.46	<.0001
9	Asia	Sedan	Wheelbase	93.0000	124.00	94	0	105.65	9931	6.4068	0.6608	41.0475	1053017	3817.41	6.0643	159.88	<.0001
10	Asia	Sedan	Length	154.00	204.00	94	0	184.01	17297	10.4506	1.0779	109.21	3192989	10157	5.6793	170.71	<.0001
11	USA	Sports	MSRP	18345	81795	9	0	45257	407315	21279	7093.15	4.5281E8	2.206E10	3.6225E9	47.0189	6.38	0.0002
12	USA	Sports	Invoice	16943	74451	9	0	41145	370302	19430	6476.54	3.7751E8	1.826E10	3.0201E9	47.2227	6.35	0.0002
13	USA	Sports	Ep...	...	8.3000	...	...	...	...	...	...	...	...	...	...	...	0.5075

Example 3: Using the COMBINE_TABLES Function to Combine Multiple ByGroupSet Tables

Use the simple.mdSummary action to generate descriptive summary statistics for the Horsepower column in the cars data set. Use the groupBy parameter to save the results to separate tables by Origin, and the unique combinations of the variables Type and Drivetrain. Use the COMBINE_TABLES function to combine all tables that contain "ByGroupSetn" in their keyname.

```
libname casuser cas caslib="casuser";

data casuser.cars;                                                    /*
1 */
  set sashelp.cars;
run;

cas casauto;

proc cas;
```

```
   simple.mdSummary result=res status=s /                              /*
2 */
      table={name="cars"}
      inputs={"Horsepower"},
      subSet={"mean", "N"},
      sets={
            {groupBy={"origin"}}
            {groupBy={"type", "drivetrain"}}
      };
run;
describe res;                                                          /*
3 */
run;
do i=1 to 2;                                                           /*
4 */
  s=combine_tables(res, cats("ByGroupSet", i));
  table = cats("summary_set", i);
  saveresult s casout=table caslib="casuser" replace;
end;
run;
quit;
```

1 Load the cars table from SASHELP into the Casuser library.

2 Use the simple.mdSummary action to generate descriptive summary statistics (mean and count) for the Horsepower column in the cars data set. Use the groupBy parameter to save the results to separate tables by Origin, and the unique combinations of the variables Type and Drivetrain. This action creates multiple ByGroupSets because there are two groupBy parameters

3 Use the DESCRIBE statement to view the data type and values of `res`. In the log, the output of the DESCRIBE statement shows that `res` is a dictionary with 19 entries where each entry is a table. The first table in each ByGroupSet is the ByGroupInfo table.

4 The combine_tables function combines all the tables that contain "ByGroupSet1" or "ByGroupSet2" in their keyname (the ByGroupInfo is ignored).

Output 3.14 *Log Output from the DESCRIBE Statement*

```
dictionary ( 19 entries, 19 used);
 [ByGroupSet1.ByGroupInfo] Table ( [3] Rows [3] columns
Column Names:
 [1] Origin           [               ] (char)
 [2] Origin_f         [               ] (char)
 [3] _key_            [               ] (varchar)
 [ByGroupSet1.ByGroup1.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet1.ByGroup2.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet1.ByGroup3.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroupInfo] Table ( [14] Rows [5] columns
Column Names:
 [1] Type             [               ] (char)
 [2] Type_f           [               ] (char)
 [3] DriveTrain       [               ] (char)
 [4] DriveTrain_f     [               ] (char)
 [5] _key_            [               ] (varchar)
 [ByGroupSet2.ByGroup1.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup2.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup3.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup4.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup5.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup6.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup7.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup8.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
 [1] Column           [Analysis Variable] (char)
 [2] N                [N              ] (double) [BEST10.]
 [3] Mean             [Mean           ] (double) [NLD8.4]
 [ByGroupSet2.ByGroup9.MDSummary] Table[MDSummary] ( [1] Rows [3] columns
Column Names:
```

Output 3.15 *Log Output*

```
NOTE: The table summary_set1 loaded into casuser with 3 observations and 4
variables.
NOTE: The table summary_set2 loaded into casuser with 14 observations and 5
variables.
```

CREATE_PARALLEL_SESSION Function

Starts a session with the same identity as the calling action. You can call this function to create multiple sessions.

Category: Server-side

Returned data type: If the function is not successful, the Boolean value of FALSE is returned. Otherwise, the returned data type is a string with the session name.

Tip: You can also specify a session on an action invocation.

Syntax

CREATE_PARALLEL_SESSION (<*number-of-workers*>);

Required Argument

number-of-workers
specifies the number of worker nodes to run the session on. If you do not specify a number, all of the workers are used.

Default All workers

Examples

Example 1: Create Parallel Sessions with All Workers

This example uses the CREATE_PARALLEL_SESSION function to create parallel sessions with all workers. This function is only available in server-side CASL and is meant to be submitted to the CAS server through the runCasl action.

```
cas casauto;
proc cas;
    source casl_code;
    parallel_session = create_parallel_session();
    endsource;
    sccasl.runCASL / code = casl_code;
quit;
```

Example 2: Create Parallel Sessions with n-Workers

This example uses the CREATE_PARALLEL_SESSION function to create parallel sessions with 37 workers. This function is only available in server-side CASL and is meant to be submitted to the CAS server through the runCasl action.

```
cas casauto;
proc cas;
    source casl_code;
    parallel_session = create_parallel_session(37);
    endsource;
    sccasl.runCASL / code = casl_code;
quit;
```

DEFAULT_CASLSTORE Function

Sets the default caslstore.

See: User Defined Functions in CASL Programmer's Guide

Syntax

DEFAULT_CASLSTORE(*caslstore-value*);

Required Argument

caslstore-value
specifies the name of the default caslstore value.

Details

The default caslstore is the code repository that is searched to locate unqualified user-defined functions. If two functions have the same name and are stored in separate caslstores, the DEFAULT_CASLSTORE function specifies which caslstore to search for the function first. If the default caslstore is not set using the DEFAULT_CASLSTORE function, then the last specified caslstore is searched for the function first. The default caslstore persists until a new default caslstore is set.

Example: Specifying Caslstore Priority Using the DEFAULT_CASLTORE Function

In this example, two functions named "myfunction" are stored in separate caslstores. The first "myfunction" calculates the factorial of the input value. The second "myfunction" converts the input Fahrenheit temperature to Celsius

temperature. When both caslstores are called in a program, the most recent caslstore is searched for the function "myfunction" first. In this program, if a default caslstore is not set, the temperature conversion function would be used when "myfunction" is called since that was the last caslstore to be defined. Once the default caslstore is set, the "myfunction" stored in the default caslstore will be used when "myfunction is called, in this case the factorial function.

```
proc cas;
source funcs;                                                    /*1*/
   function myfunction(x);
      if (x < 1.0) then return(x);
      else do;
         return exp(lgamma (x+1));
      end;
   end func;
endsource;

upload_caslstore({caslib="casuser",

     name="store1", replace=true}, funcs);

source funcs2;                                                   /*2*/
   function myfunction(temp);
      Celsius = (5/9*(temp-32));
      return(Celsius);
   end func;
endsource;

upload_caslstore({caslib="casuser",
     name="store2", replace=true}, funcs2);
run;

proc cas;
   cs1 = caslstore({caslib="casuser", name="store1"});
   cs2 = caslstore({caslib="casuser", name="store2"});
   dcs = default_caslstore(cs1);                                 /*3*/

   myresult=myfunction(10);
   print "10! = 3628800";
   print "10'F = -12.2'C";
   print "My function result= " myresult;
run;
```

1 The first "myfunction" is defined and stored in a caslstore named "store1".

2 The second "myfunction" is defined and stored in a second calstore named "store2"

3 The default caslstore is set to "store1" the factorial function.

Example Code 3.6 *Log Output:*

```
10! = 3628800
10'F = -12.2'C
My function result= 3628800
```

See Also

- "CASLSTORE Function"
- "UPLOAD_CASLSTORE Function"
- "CASLstore" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

DICTIONARY Function

Searches for a key in the given dictionary and, if found, returns its associated value.

Category: Dictionary

Syntax

DICTIONARY (*dictionary*, *string*);

Required Arguments

dictionary
specifies the dictionary to search.

string
specifies the dictionary key.

Details

Return Values

The following are the possible returned data values for the DICTIONARY function:

FALSE	The key is not found within the dictionary, or there is a parameter error.
string	Returns the value associated with the key.

Example: Using the DICTIONARY Function

The following example demonstrates how to use the DICTIONARY function to obtain the value associated with a key in a dictionary. The example searches for the key *Keefer* in the *studentId* dictionary. Because the key Keefer is present within the dictionary, the DICTIONARY function returns the value of 68101 associated with the key.

```
proc cas;
  studentID={Woods=13445, Shafer=36772, McKenna=37854, Keefer=68101,
Johnson=34690,              Shena=20047, Kingstem=60439,
Downdry=37710};
  print 'StudentID= ' dictionary(studentID, 'Keefer');
run;
```

Example Code 3.7 *SAS Log*

```
StudentId=68101
```

See Also

"CASL Dictionaries" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

DIM Function

Retrieves the dimensions of an array or dictionary.

Categories: Array
Dictionary

Returned data type: INTEGER

Syntax

DIM (*array-name*);

Required Argument

array-name
specifies the name of an 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.

Example: Using the DIM Function

The example demonstrates using the DIM function to retrieve the dimensions of a dictionary named *studentId*.

```
proc cas;
  studentId={Woods=13445, Shafer=36772, McKenna=37854, Keefer=68101,
Johnson=34690,              Shena=20047, Kingstem=60439,
Downdry=37710};
  print 'Dimensions are=' dim(studentId);
run;
```

Example Code 3.8 *SAS Log*

```
Dimensions are=8
```

See Also

- "CASL Arrays" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- CASL Dictionaries

ENCODING Function

Returns an encoding string for the current session encoding.

Data type: CHAR

Syntax

ENCODING( );

Without Arguments

The ENCODING function has no arguments.

Details

When uploading CSV files with the UPLOAD statement, you must specify the file encoding by using the ENCODING= parameter. For information about the ENCODING= parameter, see "Parameters for fileType="CSV"" in "Upload a resource" in *SAS Viya Platform: System Programming Guide*. You can dynamically pass the session encoding to the ENCODING= parameter by using the ENCODING function . The following IMPORTOPTION specification for CSV files uses the ENCODING

function to return the session encoding string and pass it to the ENCODING= parameter: `importoptions={filetype="CSV", encoding=encoding()};`

Example

In this example, the IMPORTOPTION specification for the CSV file uses the ENCODING function to return the session encoding string and pass it to the ENCODING= parameter.

```
cas casauto;
proc cas;
   upload path="your-file-path\titanic3.csv"
      importoptions={filetype="CSV", encoding=encoding()}
      casout={name="Titanic01"};
run;
```

```
{caslib=CASUSER,tableName=TITANIC01}
```

EXECUTE Function

Compiles and executes the given code.

Category: Control

Syntax

EXECUTE(*"code "*);

Required Argument

code
specifies CASL code for compilation and execution. Any functions that are created remain active.

Details

Return Values

The following are the possible returned data values for the EXECUTE function:

Table 3.3 Return Codes

Return	Description
1	compilation is not successful
0	compilation is successful

Example: Using the EXECUTE Function

```
proc cas;
   values=execute("x=10; y=50; r=x*y; print r;");
   print values;
run;
```

Output 3.16 Log

```
500
```

EXISTS Function

Determines whether a variable exists or whether a dictionary contains a specified key.

Categories: Dictionary
Function IO

Syntax

Form 1: **EXISTS** (*dictionary*, *key*);

Form 2: **EXISTS** (*variable-name*);

Required Arguments

dictionary
specifies the dictionary to search.

key
a string that specifies the dictionary key.

variable-name
a string that specifies the name of a variable.

Note A variable can exist even if it has not been assigned a value.

Details

Return Values

The following are the possible returned data values for the EXISTS function:

0 The variable or key does not exist, or there is a parameter error.

1 The variable or key exists.

Examples

Example 1: Check for an Existing Variable

In the following example, the EXISTS() function checks for the existence of the variable Name.

```
proc cas;
   Name = 'Jane Smith';
   run;
   print "Name variable exists: " exists("Name");
   run;
quit;
```

The return value 1 indicates that the variable Name exists.

```
Name variable exists: 1
```

Example 2: Check for an Existing Key

In the following example, the EXISTS() function checks for the existence of the key Copies in the dictionary hmeqTable.

```
proc cas;
   hmeqTable["name"] = "hmeq";
   hmeqTable["copies"] = 2;
run;
   print "Key exists: " exists(hmeqTable, "copies");
run;
quit;
```

The return value 1 indicates that the key Copies exists in the hmeqTable dictionary.

```
Key exists: 1
```

EXIT Function

Exits the block of CASL code with the given status.

Category: Control

Syntax

EXIT;

EXIT ({**severity** = *severity-code*, **reason** = *reason-code*, **statuscode** = *status-number*, **formatted** = "*message*" });

Without Arguments

If no arguments are supplied, then all arguments are assumed to be zero or NULL. When you use the EXIT function with no arguments, the zero or NULL values will be based on positional values in the list. Therefore, the status is the first argument, severity is the second argument, and message is the third argument.

Note: You can specify a nonzero positive status code along with a severity of 0.

Required Arguments

message
the status code, converted to a string and used in the message file.

reason-code
provides general information about the status code. For a list of acceptable reason codes, see "Reason Codes" in *SAS Cloud Analytic Services: CASL Programmer's Guide*.

status-number
the number that represents the status from the server. This number can be looked up using the CODETOSTATUS function to get more information.

severity-code
indicates whether the code ran successfully or failed. For a list of acceptable severity codes, see "Severity Codes" in *SAS Cloud Analytic Services: CASL Programmer's Guide*.

See Also

"CODETOSTATUS Function"

FINDTABLE Function

Searches the given variable for the first table it sees. This is useful when a result from an action returns a copy of a result table.

Category: Result Tables

Returned data type: Returns the Boolean value of FALSE if table is not found.

Example: Subsetting a Computed Column on page 112

Syntax

FINDTABLE (*variable*);

Required Argument

variable
specifies the name of a variable to search for a table in. Many actions return a result that includes a result table. Use this function to access the first result table in a variable.

Example: Using the FINDTABLE function

The following example demonstrates using the FINDTABLE function to find the result *r*. *r* was defined as the result variable for the Fetch action. *Findcar* is the new variable in which the results from FINDTABLE and SORT function is placed into.

```
cas casauto;

proc casutil;
   load data=sashelp.cars
   casout="cars"
   outcaslib="casuser"
run;

proc cas;
  table.fetch result=r / table="cars" to=428;
  findcar=sort_rev(findtable(r), "Invoice")[1:25, {"Make", "Model",
"Type", "DriveTrain" "MSRP", "Invoice"}];
  print findcar;
run;
```

Output 3.17 *Findcar Results*

findcar: Results

Make	Model	Type	DriveTrain	MSRP	Invoice
Porsche	911 GT2 2dr	Sports	Rear	$192,465	$173,560
Mercedes-Benz	CL600 2dr	Sedan	Rear	$128,420	$119,600
Mercedes-Benz	SL600 convertible 2dr	Sports	Rear	$126,670	$117,854
Mercedes-Benz	SL55 AMG 2dr	Sports	Rear	$121,770	$113,388
Mercedes-Benz	CL500 2dr	Sedan	Rear	$94,820	$88,324
Mercedes-Benz	SL500 convertible 2dr	Sports	Rear	$90,520	$84,325
Mercedes-Benz	S500 4dr	Sedan	All	$86,970	$80,939
Acura	NSX coupe 2dr manual S	Sports	Rear	$89,765	$79,978
Jaguar	XKR convertible 2dr	Sports	Rear	$86,995	$79,226
Audi	RS 6 4dr	Sports	Front	$84,600	$76,417
Jaguar	XKR coupe 2dr	Sports	Rear	$81,995	$74,676
Dodge	Viper SRT-10 convertible 2dr	Sports	Rear	$81,795	$74,451
Porsche	911 Carrera 4S coupe 2dr (convert)	Sports	All	$84,165	$72,206
Mercedes-Benz	G500	SUV	All	$76,870	$71,540
Cadillac	XLR convertible 2dr	Sports	Rear	$76,200	$70,546
Porsche	911 Carrera convertible 2dr (coupe)	Sports	Rear	$79,165	$69,229
Mercedes-Benz	S430 4dr	Sedan	Rear	$74,320	$69,168
Volkswagen	Phaeton W12 4dr	Sedan	Front	$75,000	$69,130
Jaguar	XJR 4dr	Sedan	Rear	$74,995	$68,306
Jaguar	XK8 convertible 2dr	Sports	Rear	$74,995	$68,306
Porsche	911 Targa coupe 2dr	Sports	Rear	$76,765	$67,128
BMW	745Li 4dr	Sedan	Rear	$73,195	$66,830
Land Rover	Range Rover HSE	SUV	All	$72,250	$65,807
Audi	A8 L Quattro 4dr	Sedan	All	$69,190	$64,740
Jaguar	XK8 coupe 2dr	Sports	Rear	$69,995	$63,756

FMTVAR Function

Creates a CASL format variable that can be used as a substitute for a SAS format.

Category: Formatting

Syntax

FMTVAR(*format-name*, <*format-width*>, <*d*>);

Required Argument

format-name
specifies a string that contains the name of the format, without width, decimal specifications, or containing any decimal point. The format is a SAS format or a user-defined format that was previously defined. Some examples of valid format names are BEST, COMMA, DAY, $HEX, and DATETIME.

Optional Arguments

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

d
specifies an optional decimal scaling factor in the numeric format. The value that you specify with a format displays that many decimal places.

Details

Return Values

The following are the possible returned data values for the FMTVAR function:

format	When the FMTVAR function succeeds, the function returns a format value that can be assigned to a variable. The variable can then be used in place of a constant format specification.
0	The FMTVAR function failed to locate the format.

Example: Creating a Format Variable

```
proc cas;
   x=62427229.72748;
   fv=fmtvar("comma",14,2);
run;
print put(x,fv);
run;
```

Example Code 3.9 *Log*

```
62,427,229.73
```

GETCOLUMN Function

Extracts the column into an array.

Category: Result Tables

Returned data type: Array

Syntax

GETCOLUMN(*table-name*, *column-name* | *column-number*);

Required Arguments

table-name
specifies the name of the table to extract the column from.

column-name
specifies the name of the column to extract into an array.

column-number
specifies the number of the column to extract into an array.

Details

The GETCOLUMN function extracts all of the rows from a column into an array. The column can be specified by column name or column number.

Note: If you specify an invalid column name or invalid column number, the GETCOLUMN function returns an empty array.

Note: When you use the GETCOLUMN function on a STRING column, trailing blanks are added to make all values of the array the same length. When you use the GETCOLUMN function on a VARCHAR column, the values of the array are null-terminated and no trailing blanks are added.

Example: Extract a Column from a Table

This example illustrates how to extract a column from a table into an array.

```
proc casutil;
```

```
   load data=sashelp.cars;
quit;

proc cas;
   table.fetch result=r / table="cars" to=428;
   findcar=findtable(r);

describe findcar;

   colarray=getcolumn(findcar, "Make");
   print colarray;
run;
quit;
```

Example Code 3.10 *(DESCRIBE Statement*

```
[Fetch] Table ( [50] Rows [16] columns
Column Names:
[1] _Index_          [               ] (int64)
[2] Make             [               ] (char)
[3] Model            [               ] (char)
[4] Type             [               ] (char)
[5] Origin           [               ] (char)
[6] DriveTrain       [               ] (char)
[7] MSRP             [               ] (double) [DOLLAR8.]
[8] Invoice          [               ] (double) [DOLLAR8.]
[9] EngineSize       [Engine Size (L) ] (double)
[10] Cylinders        [               ] (double)
[11] Horsepower       [               ] (double)
[12] MPG_City         [MPG (City)     ] (double)
[13] MPG_Highway      [MPG (Highway)  ] (double)
[14] Weight           [Weight (LBS)   ] (double)
[15] Wheelbase        [Wheelbase (IN) ] (double)
[16] Length           [Length (IN)    ] (double)
```

Example Code 3.11 *(PRINT Statement)*

```
{Acura    ,Acura    ,Acura    ,Audi     ,Audi     ,Audi     ,Audi     ,Audi     ,Audi
     ,BMW      ,BMW      ,BMW      ,BMW      ,
BMW      ,BMW      ,BMW      ,Buick    ,Buick    ,Buick    ,Cadillac ,Cadillac ,Chevr
olet,Chevrolet,Chevrolet,Chevrolet,Chevrolet,
Chevrolet,Chevrolet,Chevrolet,Chevrolet,Chrysler ,Chrysler ,Chrysler ,Chrysler ,Chrys
ler ,Dodge    ,Dodge    ,Dodge    ,Dodge    ,
Dodge    ,Ford     ,Ford     ,Ford     ,Ford     ,Ford     ,Ford     ,Ford     ,GMC
    ,GMC      ,GMC      }
```

GETENV Function

Returns the environment variable for the given string.

Categories: Control
Status

Returned data type: If the function is successful, then the returned data type is a string with the value of the environment variable. Otherwise, the returned value is 0 for failure.

Syntax

GETENV(<*string*>);

Optional Argument

string
one or more consecutive alphanumeric characters, other keyboard characters, or both.

Example

This example uses the GETENV function to get the host name for the server running the Compute Session.

```
cas casauto;

proc cas;
   /* Get the host name for the server running our Compute Session. */
   ComputeHost=GETENV("HOSTNAME");
   print(NOTE) "The Compute Server host is named " ComputeHost;
run;
quit;
```

The program writes the following output to the log:

```
NOTE: The Compute Server host is named sas-compute-server-123abcd-efgh-ijkl-mnop-
qrstuvwxyz12-34567
```

GETKEYS Function

Extracts the keys from the dictionary into an array.

Category: Dictionary

Returned data type: Array

Syntax

GETKEYS(*dictionary*);

Required Argument

dictionary
specifies the dictionary to search.

Example: Extract Dictionary Keys into an Array

This example illustrates how to extract the keys from a dictionary into an array.

```
proc cas;
   studentID={Woods=13445, Shafer=36772, McKenna=37854};
   keyArray=getkeys(studentID);
   print keyArray;
run;
```

Example Code 3.12 *Log*

```
{Woods, Shafer, McKenna}
```

GETRESULT Function

Retrieves results associated with ID or tag.

Category: Result Tables

Syntax

GETRESULT("*session-reference*", <*action-tag*><*id*>);

Required Argument

session-reference
specifies the session name to retrieve the results.

Optional Arguments

action-tag
refers to the job name of the action. The action tag is used with asynchronous sessions.

id
refers to the number in the action queue for a job. You can obtain the connection identifier by using the LISTRESULTS action. For more information, see "view saved results" in *SAS Viya Platform: System Programming Guide*.

Example: Using the GETRESULT Function

```
cas mySession1;

proc cas;
   session mySession1;
   sessionid async='session1';
run;
session.listresults;
run;
result=getresult("mySession1","session1");
describe result;
print result;
run;
```

Output 3.18 *Session.ListResults Output*

Results from session.listresults

Id	Timestamp	Action	Responses	Size	Action Tag	Description
5	10/28/09 17	listsessions	2	1180	session1	Final
6	10/28/09 17	listsessions	2	1180	session2	Final
7	10/28/09 17	listsessions	2	1171	session3	Final
8	10/28/09 17	sessionid	2	149	session1	Final

Output 3.19 *Job: Results from session.listSessions Output*

job: Results from session.listSessions

Name	UUID	Session Properties	Authentication Type	User ID
MYSESSION1:Tue Oct 29 13:10:44 2019	010be3c4-a5af-8944-b560-28f3778a0241	Connected	OAuth/External PAM	sasdemo
MYSESSION1:Tue Oct 29 13:20:42 2019	0165286d-af96-cc4a-bdb4-63dd9ee66115	Connected	OAuth/External PAM	sasdemo

Example Code 3.13 *Log (DESCRIBE Statement)*

```
dictionary ( 8 entries, 8 used);
 [session] string;
 [job] string;
 [status] dictionary ( 4 entries, 4 used);
  [severity] int64_t;
  [reason] int64_t;
  [status] nil;
  [statusCode] int64_t;
 [result] dictionary ( 1 entries, 1 used);
  [Session] Table[Listsessions] ( [2] Rows [5] columns
Column Names:
  [1] SessionName      [Name            ] (varchar)
  [2] UUID             [UUID            ] (char)
  [3] State            [Session Properties] (varchar)
  [4] Authentication   [Authentication Type] (varchar)
  [5] Userid           [User ID         ] (varchar)
 [logs] array ( 5 entries, 5 used);
  [ 1] nil;
  [ 2] nil;
  [ 3] nil;
  [ 4] nil;
  [ 5] nil;
 [loglevels] array ( 5 entries, 5 used);
  [ 1] nil;
  [ 2] nil;
  [ 3] nil;
  [ 4] nil;
  [ 5] nil;
 [timedout] nil;
 [canceled] nil;
```

Example Code 3.14 *Log (PRINT Statement)*

```
{session=mySession1,job=session1,status={severity=0,reason=0,,statusCode=0},,logs={,,
,,},loglevels={,,,,},,}
```

GETVALUES Function

Extracts the values from the dictionary into an array.

Category: Dictionary

Returned data type: Array

Syntax

GETVALUES(*dictionary*);

Required Argument

dictionary
specifies the dictionary to search.

Example: Extract Dictionary Values into an Array

This example illustrates how to extract the values from a dictionary into an array.

```
proc cas;
   studentID={Woods=13445, Shafer=36772, McKenna=37854};
   valArray=getvalues(studentID);
   print valArray;
run;
```

Example Code 3.15 *Log*

```
{13445,36772,37854}
```

INPUT_FORMAT Function

Converts the string to the best numeric value.

Category: Formatting

Returned data type: For a list of possible returned data types, see the Details section below.

Syntax

INPUT_FORMAT (*string*);

Required Argument

string
one or more consecutive alphanumeric characters, other keyboard characters, or both.

Details

Return Values

The following is a list of possible returned data values for the INPUT_FORMAT function:

0	The wrong arguments were specified.
-1	The results are invalid.
-2	overflow
-3	No value is specified.
-4	There are errors in the conversion.
-5	The specified value is not a string.

Example: Converting a String

```
proc cas;

x=input_format("12345678901234567890123456789012345678901234567890");
   print x;
run;
```

Example Code 3.16 *Log*

```
1.2345679E49
```

isType Function

Returns TRUE or FALSE based on whether an *expression* results in a value of a specified type.

Category:	Variable Information
Returned data type:	Boolean (TRUE or FALSE)
See:	"Boolean" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

Syntax

is*Type* (*value-1*, <*value-1*><,...><, *value-n*>);

Required Arguments

Type
specifies the function type.

Supported function types:

- Array
- Blob
- Double
- Dictionary
- Integer
- List
- String
- Table
- Varbinary

value

specifies the *expression* that results in a value that does or does not match the specified supported function type. The expression can be a literal value, a function, or a sequence of operands and operators that are a combination of these to produce a resulting value.

Note: There can one or more expressions defined for *value*. If there are multiple arguments, they must all evaluate to the specified function type for the function to return TRUE.

See "Definitions for CASL Expressions" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

"Examples of Expressions" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

"Operators in Expressions" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

"CASL Data Types" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

Examples

Example 1: isDouble Function: Is a Double

The following example uses the isType function to determine whether the literal (assigned to x) is a double.

```
proc cas;
  x=21/3;
    if isDouble(x) then print x "is a double.";
      else print x "is not a double.";
run;
```

Example Code 3.17 *SAS Log*

```
7 is a double.
```

Example 2: Specifying Various Types of Expressions with the isType Function

The following examples show how value can be expressed as a literal, a function, or an expression that contains operators. For these examples, note the following:

- The CAT function returns a string.
- The DIM function returns an integer.
- The EXISTS function returns an integer.

```
proc cas;
   num=isInteger( max(10, 4) );
   print 'The result of the isInteger function is ' num;
run;

proc cas;
   x="Hello"; y="World!";
   x=isString(cat(x||y));
   print 'The result of the isString function is ' x;
run;

proc cas;
   my_array = 1:10;
   x=isInteger( DIM(my_array), exists("my_array"), 5+5 );
   print 'The result of the isInteger function is ' x;
run;

proc cas;
   x=5;
   y=( x IN (1:10) );
   string=isString( (x IN (1:10)) ) ;
   print 'The result of the isString function is ' string;
run;
quit;
```

```
82   proc cas;
83      num=isInteger( max(10, 4) );
84      print 'The result of the isInteger function is ' num;
85   run;
The result of the isInteger function is TRUE
86
NOTE: PROCEDURE CAS used (Total process time):
      real time           0.01 seconds
      cpu time            0.02 seconds

87   proc cas;
89      x="Hello"; y="World!";
90      x=isString(cat(x||y));
91      print 'The result of the isString function is ' x;
92   run;
The result of the isString function is TRUE
93
NOTE: PROCEDURE CAS used (Total process time):
      real time           0.10 seconds
      cpu time            0.19 seconds

94   proc cas;
95      my_array = 1:10;
96      x=isInteger( DIM(my_array), exists("my_array"), 5+5 );
97      print 'The result of the isInteger function is ' x;
98   run;
The result of the isInteger function is TRUE
99
NOTE: PROCEDURE CAS used (Total process time):
      real time           0.01 seconds
      cpu time            0.03 seconds

100  proc cas;
101     x=5;
102     y=( x IN (1:10) );
103     string=isString( (x IN (1:10)) ) ;
104     print 'The result of the isString function is ' string;
105  run;
The result of the isString function is FALSE
106  quit;
```

JSON2CASL Function

Converts a string containing JSON formatted text to a CASL dictionary.

Category: Dictionary

Interaction: See “Interaction” on page 176.

Syntax

JSON2CASL(*string*)

Required Argument

string
a string of JSON formatted text.

Details

Return Values

The following are the possible returned data values for the JSON2CASL function:

dictionary	a dictionary that contains converted JSON.
FALSE	the JSON string failed to be converted to a dictionary.

Interaction

Setting the `CASL_STD_JSON` environment variable and the STDJSON option affect whether the string argument passed to the JSON2CASL function is interpreted strictly with regard to the JSON standard.

By default, and for backward compatibility of existing code, strict adherence to the JSON standard for input to the JSON2CASL function is not expected nor enforced. Allowances are made specifically for non-conformant text that is produced by the CASL2JSON function for use as input to the JSON2CASL function. You can manage this non-conformant behavior in the following ways:

- by setting the STDJSON option in a CASL session. This is a local setting that affects only the environment in which the CASL program is executing, whether execution is on the SAS Compute Server (SAS client) or the CAS server.
  - To set the STDJSON option on the SAS client, specify `set stdjson` or `unset stdjson` in a PROC CAS code block:

    ```
    proc cas;
       set stdjson;
    run; quit;
    ```

    Setting the STDJSON argument in a program that is running on the SAS client affects only the behavior of the function on the SAS client.
  - To set the STDJSON option on the CAS server, ensure that `set stdjson` or `unset stdjson` appears in the code submitted to the sccasl.runCasl action.

    Setting the STDJSON argument in a program that is running on the CAS server affects only the behavior of the function on the CAS server.
- by setting the `CASL_STD_JSON` environment variable. The environment variable can be set on the SAS Compute Server (SAS client) or on the CAS server. Setting the environment variable in either of these environments has a global effect and applies to all CASL programs submitted for execution on each server.
  - To set the `CASL_STD_JSON` environment variable on the SAS Compute Server, add the environment variable to the `sas.compute.server:`

startup_commands configuration instance for the Compute service in SAS Environment Manager.

```
# to enable the JSON standard
      export CASL_STD_JSON = true
# to disable the JSON standard
      export CASL_STD_JSON = false
```

Setting the environment variable in the SAS Compute Server configuration instance (sas.compute.server: startup_commands) affects only CASL code that is running on the SAS client.

- To set the CASL_STD_JSON environment variable on the CAS Server, add the environment variable to the sas.cas.instance.config: settings configuration instance for the cas-shared-default service in SAS Environment Manager.

```
# to enable the JSON standard
      export CASL_STD_JSON = true
# to disable the JSON standard
      export CASL_STD_JSON = false
```

Setting environment variables in the CAS server configuration instance (sas.cas.instance.config: settings) affects only the CASL code that is running on the CAS server.

For more information about setting environment variables in configuration instances, see "Edit Server Configuration Instances" in *SAS Environment Manager: User's Guide*.

For more information about JSON, see RFC 8259

When executing the CASL2JSON function and the JSON2CASL function in the same program and using the output of one function as the input to the other function, the STDJSON option should be set consistently:

- If the STDJSON option is *not set* when running the CASL2JSON function, then it should not be set when running the JSON2CASL function.
- If the STDJSON option *is* set when running the JSON2CASL function, then it should be set when running the CASL2JSON function.

Example: Convert a JSON File to a CASL Dictionary

The following example converts JSON to a CASL dictionary. The JSON file is created by PROC JSON and is stored in the file myjson.json. The READPATH function stores the contents of the file in the variable **Json1** as a string. The JSON2CASL function converts the JSON string in the **Json1** variable to a CASL dictionary.

```
proc json out="/tmp/myjson.json";            /* 1 */
   export sashelp.class (where=(age=12));
run;

proc cas;
   json1=readpath("/tmp/myjson.json");       /* 2 */
   mydict=json2casl(json1);                  /* 3 */
```

```
   print mydict;                                    /* 4 */
   describe mydict;                                 /* 5 */
run;
```

1 Use PROC JSON to create the JSON file Myjson.json.

2 The READPATH function stores the contents of Myjson.json in variable json1 as a string.

3 The JSON2CASL function converts the JSON string in the Json1 variable to a CASL dictionary. The dictionary is stored in the variable MyDict.

4 The PRINT statement prints the CASL dictionary in the SAS log.

5 The DESCRIBE statement prints the structure of the variable MyDict in the SAS log.

Example Code 3.18 *JSON Converted to a CASL Dictionary*

```
{SASJSONExport=1.0,SASTableData
+CLASS={{Name=James,Sex=M,Age=12,Height=57.3,Weight=83},
{Name=Jane,Sex=F,Age=12,Height=59.8,
Weight=84.5},{Name=John,Sex=M,Age=12,Height=59,Weight=99.5},
{Name=Louise,Sex=F,Age=12,Height=56.3,Weight=77},{Name=Robert,Sex=M,
Age=12,Height=64.8,Weight=128}}}
dictionary ( 2 entries, 2 used);
 [SASJSONExport] string;
 [SASTableData+CLASS] array ( 5 entries, 5 used);
  [ 1] dictionary ( 5 entries, 5 used);
   [Name] string;
   [Sex] string;
   [Age] int64_t;
   [Height] double;
   [Weight] int64_t;
  [ 2] dictionary ( 5 entries, 5 used);
   [Name] string;
   [Sex] string;
   [Age] int64_t;
   [Height] double;
   [Weight] double;
  [ 3] dictionary ( 5 entries, 5 used);
   [Name] string;
   [Sex] string;
   [Age] int64_t;
   [Height] int64_t;
   [Weight] double;
  [ 4] dictionary ( 5 entries, 5 used);
   [Name] string;
   [Sex] string;
   [Age] int64_t;
   [Height] double;
   [Weight] int64_t;
  [ 5] dictionary ( 5 entries, 5 used);
   [Name] string;
   [Sex] string;
   [Age] int64_t;
   [Height] double;
   [Weight] int64_t;
```

See Also

- “CASL2JSON Function” on page 133

- “READPATH Function” on page 185
- “PRINT Statement” on page 79
- “JSON Procedure” in *Base SAS Procedures Guide*

LIST_PARALLEL_SESSIONS Function

Lists parallel sessions.

Category: Server-side

Syntax

LIST_PARALLEL_SESSIONS< ("*session-reference*")>;

Optional Argument

session-reference
specifies the session name to list.

Example: List Parallel Sessions

This example uses the LIST_PARALLEL_SESSIONS function to list parallel sessions. This function is only available in server-side CASL and is meant to be submitted to the CAS server through the runCasl action.

```
proc cas;
    source casl_code;
    list_parallel_sessions();
    endsource;
    sccasl.runCASL / code = casl_code;
quit;
```

LOC Function

Returns the row number in which the given value is found in the given column.

Category: Result Tables

Returned data type: Returns the value of -1 when the value is not found, otherwise returns the row number.

Syntax

LOC (*table-name*, *column*, *value*);

Required Arguments

table-name
The name of the table from which the row value is retrieved.

column
The column can be can be a number or a character value.

value
The value to search for in the table. The value can be a character or numeric value.

Example: Using the LOC Function

The following example demonstrates how to use the LOC function to retrieve a row number for a specified value of *75000*.

```
proc casutil;
   load data=sashelp.cars
   casout="cars"
   outcaslib="casuser"
   replace;
run;

proc cas;
  table.fetch result=r/ table="cars" to=428;
   findcar=sort_rev(findtable(r), "Invoice")[1:25,     {"Make",
"Model", "Type", "DriveTrain" "MSRP", "Invoice"}];
  y=loc(findcar, "MSRP", 75000);
   print "The row in which the value appears is= " y;
run;
```

Example Code 3.19 *SAS Log*

```
The row in which the value appears is= 18
```

MEMORYSTATS Function

Displays CPU and memory statistics.

Category: Status

Syntax

MEMORYSTATS ();

Without Arguments

There are no arguments for this function.

Example: Using the MEMORYSTATS Function

```
proc cas;
  memorystats();
print memorystats();
run;
```

Example Code 3.20 *SAS Log*

```
{user_cpu_time=9.68,system_cpu_time=2.64,timestamp=1840474683,current_memory=52015104
,high_water_mark_memory=52015104,current_thread_count=12}
```

NEWTABLE Function

Creates a new table.

Category: Result Tables

Syntax

NEWTABLE("*table-name*", *column-name*, *data-type*, *row-1*<, *row-2*, ... *row-N*>);

Required Arguments

"table-name"
specifies the name of the table.

column-name
specifies a list of column names for the new table. *column-name* can be any expression that results in a list of values.

Note: The list is treated as an array and values within the list are assigned to the table in increasing array index order.

data-type
specifies a list of values that defines the data type for the column values. *data-type* can be any expression that results in a list of values.

Note: The list is treated as an array and values within the list are assigned to columns in the table row in increasing array index order.

See For information about supported data types, see Data Types Supported in CASL.

Optional Argument

row
specifies one or more lists of column values to add as rows to the table. Each row argument can be any expression that results in a list of values.

Note: The lists are treated as arrays and values within the lists are assigned to columns in the table row in increasing array index order.

Example

This example creates a new table, Average Class Grades, which contains three rows and three columns. The columns contain int64 type data.

```
proc cas;
title "Average Class Grades";
  colNames={"Class A","Class B","Class C"}; /*1*/
  colTypes={"int64","int64","int64"};       /*2*/
  row1={71 74 84};                          /*3*/
  row2={65 74 65};
  row3={80 80 90};
  myTable=newtable("myTable",colNames,colTypes,row1,row2,row3); /*4*/
  print myTable;
  describe myTable;                         /*5*/
run;
quit;
```

1 Specify a list of column names. In this example, the names are specified as an array.

2 Specify a list of data types for each column. The types are specified as an array of integers.

3 Specify three rows for the table. The values for the rows are specified as arrays.

4 Specify the NEWTABLE function to create the table.

5 Specify the describe statement to list your new table with each column and its data types as you set them.

Output 3.20 *NEWTABLE Function Output: Average Class Grades Output*

Average Class Grades

myTable: Results

Class A	Class B	Class C
71	74	84
65	74	65
80	80	90

Example Code 3.21 *SAS Log: DESCRIBE Statement*

```
[myTable] Table ( [3] Rows [3] columns
Column Names:
[1] Class A          [Class A         ] (int64)
[2] Class B          [Class B         ] (int64)
[3] Class C          [Class C         ] (int64)
```

See Also

- "CASL Data Types" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "DESCRIBE Statement"

PUT Function

Formats a value with the given format and returns it as a string.

Category: Formatting

Syntax

PUT (*value*, *format*);

Required Arguments

value
specifies a numeric value to be formatted.

format
specifies a format.

Example: Using the PUT Function

You can use the PUT function with the PRINT statement to format your value.

```
proc cas;
  print put(10, hex8.);
run;
```

Example Code 3.22 *SAS Log*

```
0000000A
```

See Also

"PRINT Statement"

READFILE Function

Reads and returns the contents of the specified file reference.

Valid in:	Client-side only
Category:	File IO
Returned data type:	String | Varbinary

Syntax

READFILE("*file-reference*");

Required Argument

"file-reference"
The file reference of the file whose contents you are attempting to input.

Details

By default, the READFILE function requires text based data in your file. Input text files that are encoded with SAS session encoding are transcoded to UTF-8 encoding and then output as a string.

Example: Using the READFILE Function

The following example illustrates how to read the contents of the file *Findtable1.sas*.

```
proc cas;
  filename findtest '/u/sasdemo/public/findtable1.sas';
  test=readfile("findtest");
  print test;
run;
```

The output of this example is the contents of the *Findtable1.sas*, which contains a program with actions and CASL statements.

READPATH Function

Reads and returns the contents of the specified filename.

Valid in:	Client-side only
Category:	File IO
Returned data type:	String | Varbinary

Syntax

READPATH ("*file-name*");

Required Argument

"file-name"
The name of the file that you want to read.

Details

By default, the READPATH function requires text based data in your file. Input text files that are encoded with SAS session encoding are transcoded to UTF-8 encoding and then output as a string.

Example: Read the Contents of a File Into a Variable

```
proc cas;
   store = readpath("/tmp/sgeon1.sas");
   print store;
run;
```

Output 3.21 *Log Output*

```
ods listing sge=on;ods html sge=on;ods graphics / reset imagename="sgeon"
test;proc sgscatter data=sashelp.class;   matrix age
height weight / group=sex diagonal=(histogram);run;ods listing close;proc
sgscatter data=sashelp.class;   matrix age height
weight / group=sex diagonal=(histogram);run;ods _all_ close;
```

RESULT_BY_COL Function

Creates a new table with the given columns.

Category: Result Tables

Returned data type: CAS Table

Syntax

RESULT_BY_COL (*table-name*, "*column-name*", *column_number*);

Required Arguments

table-name
specifies the name of the new table.

"*column-name*," or *column_number*
Columns are referenced either by name or number. The order of the columns in the new table is the same as the order given in the call.

Example

This example uses the RESULT_BY_COL function to create new tables with the columns `Name` and `G1` from the result table `myTable`.

```
proc cas;
/* Create a result table to work with*/
   myTable=newtable("myTable" /* Table Name */
                  ,{"Name","G1","G2"} /* Column names in order */
```

```
                    ,{"Varchar","int64","int64"} /* Column data types in
order */
                    ,{"Able" 74 84}    /* Row 1 values */
                    ,{"Baker" 74 65}   /* Row 2 values */
                    ,{"Charlie" 80 90} /* Row 3 values */);
   Result=RESULT_BY_COL(myTable,"Name");
   print Result;
   Result=RESULT_BY_COL(myTable,2);
   print result;
run;
quit;
```

The program writes the following output to Results:

Result: Results

Name
Able
Baker
Charlie

result: Results

G1
74
74
80

RESULT_BY_TYPE Function

Creates a new table with columns that match the type specifications.

Category: Result Tables

Syntax

RESULT_BY_TYPE (*table-name*, "<*data types*>");

Required Argument

table-name
specifies the name of the new table.

Optional Argument

data types

A data type is an attribute that specifies what type of data the column stores. For information about supported data types, see Data Types Supported in CASL.

Example

This example uses the RESULT_BY_TYPE function to create new tables that include columns with VARCHAR and INT64 data types from the result table `myTable`.

```
proc cas;
/* Create a result table to work with*/
   myTable=newtable("myTable" /* Table Name */
                  ,{"Name","G1","G2"} /* Column names in order */
                  ,{"Varchar","int64","int64"} /* Column data types in
order */
                  ,{"Able" 74 84}    /* Row 1 values */
                  ,{"Baker" 74 65}   /* Row 2 values */
                  ,{"Charlie" 80 90} /* Row 3 values */);
   Result=RESULT_BY_TYPE(myTable,"VARCHAR");
   TITLE "Values for VARCHAR columns:";
   print Result;print;
   Result=RESULT_BY_TYPE(myTable,"INT64");
   TITLE "Values for INT64 columns";
   print result;
run;
quit;
```

The program writes the following output to Results:

Values for VARCHAR columns

Result: Results

Name
Able
Baker
Charlie

Values for INT64 columns

Result: Results

G1	G2
74	84
74	65
80	90

See Also

"CASL Data Types" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

SEND_RESPONSE Function

Sends the specified result back to the client.

Category: Server-side

Syntax

SEND_RESPONSE (*result*);

Required Argument

result
specifies the results to send back to the client.

Example: Send Results Back to the Client

This example uses the SEND_RESPONSE function to send the resulting objects on the server back to the client.

```
cas casauto;
proc cas;
   builtins.defineActionSet / name="myFunctionSet"
      actions={
         {
          name="TempCtoF"
          desc="Celsius to Fahrenheit Conversion"
          parms={
             {name="c" type="double" required=TRUE}
             }
          definition="
              val=(c*(9/5)+32);
              send_response({val});
              "
          }
          };
quit;
```

SLEEP Function

Sleep in the operating system (OS) for the number of seconds specified.

Categories: Control
Status

Syntax

SLEEP (<*value*>);

Optional Argument

value
specifies the number of seconds that the OS sleeps.

Example

This example uses the SLEEP function to specify 120 as the number of seconds that the OS sleeps.

```
cas casauto;

proc cas;
```

```
   sleep(120);
run;
```

SORT Function

Returns a list sorted in ascending order.

Category: Array

Default: Ascending Order

Syntax

SORT (*list*);

Required Argument

list
specifies a countable number of values.

Examples

Example 1: Sort a List In Ascending Order

The following example creates a list with numerical values, and then uses the SORT function to return the list in ascending order.

```
proc cas;
  list={98, 74, 54, 18, 101, 67, 80, 90, 62};   /* #1 */
  alist= sort(list);                            /* #2 */
print alist;                                    /* #3 */
run;
```

1 Create a list with numerical values.

2 The SORT function returns the list sorted in ascending order and saves the list in the *alist* variable.

3 Print the *alist* variable. For more information, see PRINT Statement.

The sorted list is printed in the SAS log

Example Code 3.23 *SAS Log*

```
{18,54,62,67,74,80,90,98,101}
```

Example 2: Sort a Keyed List in Ascending Order

The following example creates a keyed list with numerical values, and then uses the SORT function to return the keyed list in ascending order.

```
proc cas;
  Mary_Jones_Average={Math_Avg=88.6                /* #1 */
   Science_Avg=90.3
   English_Avg=89.5
   Economics_Avg=82.5};
  MJ_Average=sort(Mary_Jones_Average);            /* #2 */
print MJ_Average;                                 /* #3 */
run;
```

1 Create a keyed list named *Mary_Jones_Average*. In this example, we are looking for Mary Jones's average in all her classes.

2 The SORT function returns the keyed list sorted in ascending order and saves the list in the *MJ_Average* variable.

3 Print the *MJ_Average* variable. For more information, see PRINT Statement.

The sorted keyed list is printed in the SAS log.

Example Code 3.24 *SAS Log*

```
{Economics_Avg=82.5,Math_Avg=88.6,English_Avg=89.5,Science_Avg=90.3}
```

See Also

"SORT REV Function"

SORT REV Function

Returns a list sorted in descending order.

Category: Array

Syntax

SORT_REV (*list*);

Required Argument

list
specifies a countable number of values.

Examples

Example 1: Sort a List In Descending Order

The following example creates a list with numerical values, and then uses the SORT function to return the list in descending order.

```
proc cas;
  list={98, 74, 54, 18, 101, 67, 80, 90, 62};      /* #1 */
  alist= sort_rev(list);                            /* #2 */
print alist;                                        /* #3 */
run;
```

1 Create a list with numerical values.

2 The SORT_REV function returns the list sorted in descending order and saves the list in the *alist* variable.

3 Print the *alist* variable. For more information, see PRINT Statement.

The sorted list is printed in the SAS log

Example Code 3.25 *SAS Log*

```
{101,98,90,80,74,67,62,54,18}
```

Example 2: Sort a Keyed List in Descending Order

The following example creates a keyed list with numerical values, and then uses the SORT function to return the keyed list in descending order.

```
proc cas;
  Mary_Jones_Average={Math_Avg=88.6                  /* #1 */
   Science_Avg=90.3
   English_Avg=89.5
   Economics_Avg=82.5};
  MJ_Average=sort_rev(Mary_Jones_Average);          /* #2 */
print MJ_Average;                                   /* #3 */
run;
```

1 Create a keyed list named *Mary_Jones_Average*. In this example, we are looking for Mary Jones's average in all her classes.

2 The SORT_REV function returns the keyed list sorted in descending order and saves the list in the *MJ_Average* variable.

3 Print the *MJ_Average* variable. For more information, see PRINT Statement.

The sorted keyed list is printed in the SAS log.

Example Code 3.26 *SAS Log*

```
{Science_Avg=90.3,English_Avg=89.5,Math_Avg=88.6,Economics_Avg=82.5}
```

See Also

"SORT Function"

SYMDEL Function

Deletes the specified macro variable.

Category: Macros

Syntax

SYMDEL ("*macro-variable*");

Required Argument

macro-variable
The name of the macro variable that you want to delete.

Details

In SAS Viya, the Compute Server includes the SAS Macro Facility but the CAS server does not. Macro variables are created and managed by the SAS Macro Facility on the Compute Server in memory spaces known as Symbol Tables. When executed on the Compute Server, the "SYM" functions (SYMPUT, SYMPUTX, SYMGET, and SYMDEL) create, modify, retrieve, and delete macro Symbol Table variable values during program execution. Because the Macro Facility does not exist in CAS, no Symbol Tables are available. When executed in CAS, these "SYM" functions create, modify, retrieve, and delete values from a special memory area known as the Named Repository. The Named Repository only accepts values stored using SYMPUT or SYMPUTX, and those values can only be retrieved using the SYMGET function. There is no interaction between the Named Repository on CAS and the Global Symbol Table on the Compute Server.

Examples

Example 1: Delete a Macro Variable from the Named Repository in CAS Using the SYMDEL Function

The following example shows how the SYMDEL function runs on the CAS server to delete a macro variable.

Note: The CAS server does not support SAS macro processing. Because there is no macro facility in CAS, there is no macro symbol table. The macro variable values are stored in memory in a Named Repository. When the SYMDEL function runs on the CAS server, it deletes the specified named memory location from the Named Repository in CAS. For documentation purposes, both storage locations are referred to as a *macro-variable*.

```
/* Pre-processing on the Compute Server                          /* 1
*/
%let mvar1=Compute Server value
1;
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;

cas casauto;                                                     /* 2
*/

proc cas;                                                        /* 3
*/
   source myProgram;                                             /* 4
*/
      symputx("mvar1",'mvar1 in the Named Repository');          /* 5
*/
      print "After CASL SYMPUTX, mvar1=" symget('mvar1');        /* 6
*/
      SYMDEL("mvar1");                                           /* 7
*/
      print "After CASL SYMDEL, mvar1=" symget('mvar1');         /* 8
*/
   endsource;
run;
   runcasl code=myProgram;                                       /* 9
*/
run;

%put NOTE: *** Compute Server Symbol Table Variable Values ***;  /*
10 */
%put NOTE: &=mvar1;
```

1 Set the macro variable value to `mvar1` on the Compute Server.

2 If you have not already started a CAS session, specify the CAS statement to start a CAS session.

3 Specify the PROC CAS statement to define your CASL program. PROC CAS also enables you to specify CAS actions that run on the CAS server.

4 Specify the SOURCE statement to write the CASL program and store the program in the CASL variable `myProgram`.

5 Specify the SYMPUTX function on page 205 to store the specified text value in *macro-variable-name*, named memory location `mvar1`.

6 Specify the PRINT statement to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.

7 Specify the SYMDEL function to delete *macro-variable-name*, the named memory location `mvar1`.

8 Specify the PRINT statement to attempt to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.

9 Specify the runCasl action to run the program and print the results to the log. Because CAS actions can execute only in CAS, using the runCASL action ensures that this code executes in CAS.

10 Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table remain unchanged.

The results show that `mvar1` is missing from the Named Repository after being deleted by SYMDEL.

```
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=Compute Server value 1
...
After CASL SYMPUTX, mvar1=mvar1 in the Named Repository
...
After CASL SYMDEL, mvar1=.
...
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=Compute Server value 1
```

Example 2: Delete a Macro Variable from the Compute Server Symbol Table Using the SYMDEL Function

The following example shows how the SYMDEL function runs on the Compute Server to delete a macro variable from the Compute Server macro symbol table.

Note: The SAS client (the SAS Compute Server) supports full macro processing. When executed on the SAS Compute Server, SYMDEL deletes a macro variable from the Compute Server macro symbol table.

```
/* Pre-processing on the Compute Server                          /* 1
*/
%let mvar1=Compute Server value
1;
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;

proc cas;                                                        /* 2
*/
   symputx("mvar1",'mvar1 in the Named Repository');             /* 3
*/
   print "After CASL SYMPUTX, mvar1=" symget('mvar1');           /* 4
*/
   SYMDEL("mvar1");                                              /* 5
*/
   print "After CASL SYMDEL, mvar1=" symget('mvar1');            /* 6
*/
run;
```

```
%put NOTE: *** Compute Server Symbol Table Variable Values ***;  /* 7
*/
%put NOTE: &=mvar1;
```

1 Set the macro variable values to `mvar1` on the Compute Server.

2 Specify the PROC CAS statement to define your CASL program.

3 Specify the SYMPUTX function on page 205 to store the specified text value in *macro-variable-name*, named memory location `mvar1`.

4 Specify the PRINT statement to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.

5 Specify the SYMDEL function to delete `mvar1` from the Named Repository.

6 Specify the PRINT statement to attempt to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.

7 Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table have been altered by the CASL program.

The results show that `mvar1` is no longer found in the macro symbol table after being deleted by SYMDEL.

```
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=Compute Server value 1
...
After CASL SYMPUTX, mvar1=mvar1 in the Named Repository
After CASL SYMDEL, mvar1=.
...
NOTE: *** Compute Server Symbol Table Variable Values ***
...
WARNING: Apparent symbolic reference MVAR1 not resolved.
NOTE: mvar1
```

See Also

- "SYMGET Function"
- "SYMPUT Function"
- "SYMPUTX Function"

SYMGET Function

Returns the value of a macro variable.

Category:	Macros
Returned data type:	String, missing value (if the function fails to return a value)

Note: The SYMGET function can be run on the SAS Compute Server or the CAS Server. If the SYMGET function is run on the Compute Server, the resulting value is retrieved from the specified macro variable in the macro facility's Global Symbol Table. Because the macro facility does not exist in CAS, when the SYMGET function is run on the CAS Server, the values are retrieved from a special memory area known as the Named Repository. The Named Repository only accepts values stored using SYMPUT or SYMPUTX, and those values can only be retrieved using the SYMGET function. There is no interaction between the Named Repository on CAS and the Global Symbol Table on the Compute Server.

See: "Using the Macro Facility on the SAS Viya Platform" in *SAS Macro Language: Reference*

Syntax

SYMGET(*macro-variable-name*);

Required Argument

macro-variable-name
The name of the macro-variable from which the value is returned.

Details

In SAS Viya, the Compute Server includes the SAS Macro Facility but the CAS server does not. Macro variables are created and managed by the SAS Macro Facility on the Compute Server in memory spaces known as Symbol Tables. When executed on the Compute Server, the "SYM" functions (SYMPUT, SYMPUTX, SYMGET, and SYMDEL) create, modify, retrieve, and delete macro Symbol Table variable values during program execution. Because the Macro Facility does not exist in CAS, no Symbol Tables are available. When executed in CAS, these "SYM" functions create, modify, retrieve, and delete values from a special memory area known as the Named Repository. The Named Repository only accepts values stored using SYMPUT or SYMPUTX, and those values can only be retrieved using the SYMGET function. There is no interaction between the Named Repository on CAS and the Global Symbol Table on the Compute Server.

Examples

Example 1: Retrieve a Macro Variable from the Named Repository in CAS Using the SYMGET Function

The following example shows how the SYMGET function runs on the CAS server to retrieve a macro variable.

Note: The CAS server does not support SAS macro processing. Because there is no macro facility in CAS, there is no macro symbol table. The macro variable values are stored in memory in a Named Repository. When the SYMGET function runs on the CAS server, it retrieves the specified named memory location from the Named Repository in CAS. For documentation purposes, both storage locations are referred to as a *macro-variable*.

```
/* Pre-processing on the Compute Server                          /* 1
*/
%let mvar1=Compute Server value
1;
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;

cas casauto;                                                     /* 2
*/

proc cas;                                                        /* 3
*/
   source myProgram;                                             /* 4
*/
      symputx("mvar1",'mvar1 in the Named Repository');          /* 5
*/
      print "After CASL SYMPUTX, mvar1=" symget('mvar1');        /* 6
*/
   endsource;
run;
   runcasl code=myProgram;                                       /* 7
*/
run;
quit;

%put NOTE: *** Compute Server Symbol Table Variable Values ***;  /* 8
*/
%put NOTE: &=mvar1;
```

1 Set the macro variable value to `mvar1` on the Compute Server.

2 If you have not already started a CAS session, specify the CAS statement to start a CAS session.

3 Specify the PROC CAS statement to define your CASL program. PROC CAS also enables you to specify CAS actions that run on the CAS server.

4 Specify the SOURCE statement to write the CASL program and store the program in the CASL variable `myProgram`.

5 Specify the SYMPUTX function on page 205 to store the specified text value in *macro-variable-name*, named memory location `mvar1`.

6 Specify the PRINT statement to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.

7 Specify the runCasl action to run the program and print the results to the log. Because CAS actions can execute only in CAS, using the runCASL action ensures that this code executes in CAS.

8 Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table remain unchanged.

The results show that the value of `mvar1` has not been modified in the macro symbol table.

```
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=Compute Server value 1
...
After CASL SYMPUTX, mvar1=mvar1 in the Named Repository
...
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=Compute Server value 1
```

Example 2: Retrieve a Macro Variable from the Compute Server Symbol Table Using the SYMGET Function

The following example shows how the SYMGET function runs on the Compute Server to retrieve a macro variable from the Compute Server macro symbol table.

Note: The SAS client (the SAS Compute Server) supports full macro processing. When executed on the SAS Compute Server, SYMGET retrieves a macro variable from the Compute Server macro symbol table.

```
/* Pre-processing on the Compute Server                          /* 1
*/
%let mvar1=Compute Server value
1;
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;

cas casauto;                                                     /* 2
*/
.
proc cas;                                                        /* 3
*/
      symputx("mvar1",'mvar1 modified by CASL');                 /* 4
*/
      print "After CASL SYMPUTX, mvar1=" symget('mvar1');        /* 5
*/
run;
quit;

%put NOTE: *** Compute Server Symbol Table Variable Values ***;  /* 6
*/
%put NOTE: &=mvar1;
```

1 Set the macro variable value to `mvar1` on the Compute Server.

2 If you have not already started a CAS session, specify the CAS statement to start a CAS session.

3 Specify the PROC CAS statement to define your CASL program.

4 Specify the SYMPUTX function on page 205 to store the specified text value in *macro-variable-name* (`mvar1`).

5 Specify the PRINT statement to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.

6 Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table have been altered by the CASL program.

The results show that the value of `mvar1` has been modified in the macro symbol table.

```
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=Compute Server value 1
...
After CASL SYMPUTX, mvar1=mvar1 modified by CASL
...
*** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=mvar1 modified by CASL
```

See Also

- "SYMDEL Function"
- "SYMPUT Function"
- "SYMPUTX Function"
- "Using the Macro Facility on the SAS Viya Platform" in *SAS Macro Language: Reference*

SYMPUT Function

Stores a value in a SAS macro variable.

Category: Macros

Note: When run on the CAS server, the SYMPUT function does not modify the values of the macro variable in the SAS client. For more information, see Details. on page 202

See: "Using the Macro Facility on the SAS Viya Platform" in *SAS Macro Language: Reference*

Syntax

SYMPUT (*macro-variable-name*, *value*);

Required Arguments

macro-variable-name
the name of the macro variable to store a value in.

value
the numeric or character value to store in the macro variable.

Details

In SAS Viya, the Compute Server includes the SAS Macro Facility but the CAS server does not. Macro variables are created and managed by the SAS Macro Facility on the Compute Server in memory spaces known as Symbol Tables. When executed on the Compute Server, the "SYM" functions (SYMPUT, SYMPUTX, SYMGET, and SYMDEL) create, modify, retrieve, and delete macro Symbol Table variable values during program execution. Because the Macro Facility does not exist in CAS, no Symbol Tables are available. When executed in CAS, these "SYM" functions create, modify, retrieve, and delete values from a special memory area known as the Named Repository. The Named Repository only accepts values stored using SYMPUT or SYMPUTX, and those values can only be retrieved using the SYMGET function. There is no interaction between the Named Repository on CAS and the Global Symbol Table on the Compute Server.

Examples

Example 1: Assign a Value to a Macro Variable in CAS Using the SYMPUT Function

The following example shows how the SYMPUT function runs on the CAS server to create or modify a macro variable and assign it the value specified in the function. The function does not remove leading and trailing blanks from the *macro-variable-value*.

Note: The CAS server does not support SAS macro processing. When the SYMPUT function runs on the CAS server, it assigns the specified value to a named memory location in CAS. Because there is no macro facility in CAS, there is no macro symbol table. The macro variable values are stored in memory in a Named Repository and are only accessible using the SYMGET function. For documentation purposes, both storage locations are referred to as a *macro-variable*.

```
/* Pre-processing on the Compute Server                          /* 1
*/
%let mvar1=Compute Server value
1;
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;
```

```
cas casauto;                                                    /* 2
*/

proc cas;                                                       /* 3
*/
   source myProgram;                                            /* 4
*/
      symput("mvar1","    modified by CASL SYMPUT");             /* 5
*/
      print "CASL SYMPUT mvar1=" symget('mvar1');               /* 6
*/
   endsource;
run;

   runcasl code=myProgram;                                      /* 7
*/
run;

%put NOTE: *** Compute Server Symbol Table Variable Values ***;  /* 8
*/
%put NOTE: &=mvar1;
```

1 Set the macro variable value to `mvar1` on the Compute Server.

2 If you have not already started a CAS session, specify the CAS statement to start a CAS session.

3 Specify the PROC CAS statement to define your CASL program. PROC CAS also enables you to specify CAS actions that run on the CAS server.

4 Specify the SOURCE statement to write the CASL program and store the program in the CASL variable `myProgram`.

5 Specify the SYMPUT function to store the specified text value in *macro-variable-name*, named memory location `mvar1`.

6 Specify the PRINT statement to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.

7 Specify the runCasl action to run the program and print the results to the log. Because CAS actions can execute only in CAS, using the runCASL action ensures that this code executes in CAS.

8 Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table remain unchanged.

The results show that the leading and trailing blanks are not removed from the value, and the value of `mvar1` in the Compute Server macro symbol table has not been modified.

```
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=Compute Server value 1
...
CASL SYMPUT mvar1=    modified by CASL SYMPUT
...
NOTE: *** Compute Server Symbol Table Variable Values ***
...
MVAR1=Compute Server value 1
```

Example 2: Assign a Value to a Macro Variable on the Compute Server Using the SYMPUT Function

The following example shows how the SYMPUT function runs on the Compute Server to create or modify a macro variable and assign it the value specified in the function. The function does not remove leading or trailing blanks from the *macro-variable-value*.

Note: The SAS client (the SAS Compute Server) supports full macro processing. When executed on the SAS Compute Server, SYMPUT creates or modifies a macro variable in the Compute Server macro symbol table.

```
/* Set some macro variable values on the Compute Server */
%let mvar1='Compute Server value 1';                                    /*
1 */
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;

proc cas;                                                               /*
2 */
      symput("mvar1","    modified by CASL SYMPUT");                    /*
3 */
run;
   print "CASL SYMPUT mvar1=" symget('mvar1');                          /*
4 */
run;

%put NOTE: *** Compute Server Symbol Table Variable Values ***;         /*
5 */
%put NOTE: &=mvar1;
```

1. Set the macro variable value to `mvar1` on the Compute Server.
2. Specify the PROC CAS statement to define your CASL program.
3. Specify the SYMPUT function to modify the value of macro variable `mvar1`.
4. Specify the PRINT statement to retrieve the value of `mvar1` using the SYMGET function, and write the result to the log.
5. Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table have been altered by the CASL program.

The results show that the leading and trailing blanks are not removed from the value, and the value of `mvar1` has been modified in the Compute Server macro symbol table.

```
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1='Compute Server value 1'
...
CASL SYMPUT mvar1=    modified by CASL SYMPUT

NOTE: *** Compute Server Symbol Table Variable Values ***;
...
```

See Also

- "SYMGET Function"
- "SYMDEL Function"
- "SYMPUTX Function"
- "Using the Macro Facility on the SAS Viya Platform" in *SAS Macro Language: Reference*

SYMPUTX Function

Assigns a value to a macro variable, and removes both leading and trailing blanks.

Category: Macros

Note: When run on the CAS server, the SYMPUTX function does not modify the values of the macro variable in the SAS client. For more information, see Details on page 206.

See: "Using the Macro Facility on the SAS Viya Platform" in *SAS Macro Language: Reference*

Syntax

SYMPUTX (*macro-variable-name*, *macro-variable-value*, <*symbol-table*>);

Required Arguments

macro-variable-name
can be one of the following values:

- a character string that is a SAS variable name enclosed in quotation marks.
- the name of a character variable whose value is a SAS name.
- an expression that produces a valid SAS variable name. This form is useful for creating a series of macro variables.
- Leading and trailing blanks are removed from the *macro-variable-name*.

macro-variable-value
specifies a character or numeric constant, variable, or expression. If *macro-variable-value* is numeric, SAS converts the value to a character string using the BESTw. format and does not issue a note to the SAS log. Leading and trailing blanks are removed, and the resulting character string is assigned to the macro variable.

Optional Argument

symbol-table
specifies a character constant, variable, or expression. The value of *symbol-table* is not case sensitive. On the SAS Compute Server, the first non-blank character in *symbol-table* specifies the symbol table in which to store the macro variable. On the CAS server, this parameter is ignored. The following values are valid as the first non-blank character in *symbol-table*:

Default F

Restriction The *symbol-table* argument in the SYMPUTX function is not supported when running the SYMPUTX function on the CAS server. The parameter is ignored.

F specifies that if the macro variable exists in any symbol table, SYMPUTX uses the version in the most local symbol table in which it exists. If the macro variable does not exist, SYMPUTX stores the variable in the most local symbol table. This is the default behavior if the *symbol-table* argument is not specified.

G specifies that the macro variable is stored in the global symbol table, even if the local symbol table exists.

L specifies that the macro variable is stored in the most local symbol table that exists. The global symbol table is always used when SYMPUTX is called outside of a macro definition.

Details

Overview

In SAS Viya, the Compute Server includes the SAS Macro Facility but the CAS server does not. Macro variables are created and managed by the SAS Macro Facility on the Compute Server in memory spaces known as Symbol Tables. When executed on the Compute Server, the "SYM" functions (SYMPUT, SYMPUTX, SYMGET, and SYMDEL) create, modify, retrieve, and delete macro Symbol Table variable values during program execution. Because the Macro Facility does not exist in CAS, no Symbol Tables are available. When executed in CAS, these "SYM" functions create, modify, retrieve, and delete values from a special area of memory known as the Named Repository. The Named Repository only accepts values stored using SYMPUT or SYMPUTX, and those values can only be retrieved using the SYMGET function. There is no interaction between the Named Repository on CAS and the Global Symbol Table on the Compute Server.

About the SYMPUTX Function

When executed on the Compute Server, SYMPUTX removes both leading and trailing blanks from the value and assigns the value to a macro variable stored in a macro symbol table. If executed on the CAS server, SYMPUTX removes both leading and trailing blanks, then assigns a value to a named memory location in

CAS. For documentation purposes, both storage locations are referred to as a macro variable.

Comparisons

The SYMPUTX function is similar to the SYMPUT function except for the following differences:

Table 3.4 *Comparing SYMPUT with SYMPUTX*

SYMPUT Function	SYMPUTX Function
converts the macro variable to a string and writes a note to the SAS log if the *macro-variable-value* is a numeric constant	converts the macro variable to a string using the BESTw. format and does not write a note to the SAS log if the *macro-variable-value* is a numeric constant
uses a field width of up to 12 characters	uses a field width of up to 32 characters when it converts the macro variable to a character data type
does not allow leading blanks in the *macro-variable-name* must be a valid SAS name that begins with a letter of the Latin Alphabet, A...Z, a...z, or an underscore (_) does not remove leading and trailing blanks from either the *macro-variable-name* or the *macro-variable-value*	allows leading blanks in the *macro-variable-name* removes leading and trailing blanks from the *macro-variable-name*

See "Example 2: Create Macro Variables on the SAS Compute Server Using the SYMPUT Function and the SYMPUTX Function", to see some differences between the SYMPUT and SYMPUTX macro functions.

Examples

Example 1: Assign a Value to a Macro Variable and Remove Leading and Trailing Blanks

The following example shows how the SYMPUTX function runs on the CAS server to create or modify a macro variable and assign it the value specified in the function. The function removes leading and trailing blanks from both the *macro-variable-name* and the *macro-variable-value*.

Note: The CAS server does not support SAS macro processing. When the SYMPUTX function runs on the CAS server, it removes leading and trailing blanks from the specified value and then assigns the value to a named memory location in CAS. Because there is no macro facility in CAS, there is no macro symbol table. The macro variable values are stored in memory in a Named Repository and are only accessible using the SYMGET function. For documentation purposes, both storage locations are referred to as a *macro-variable*.

```
/* Pre-processing on the Compute
Server                                        /*1*/
%let mvar1=Compute Server value
1;
%let mvar2=Compute Server value 2;
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;
%put NOTE: &=mvar2;

cas
casauto;
 /*2*/

proc
cas;
/*3*/
   source
myProgram;                                                          /*4*
/
      x = symputx("    mvar1    ","    removes leading/trailing
blanks    "); /*5*/
      y = symputx("   mvar2    ",
123.456    );                              /*6*/
      A =
symget("mvar1");                                                    /*7*
/
      B = symget("   mvar2    ");
      print "SYMPUTX "
A;                                                    /*8*/
      print "Numeric constant " B " is converted to string and blanks
removed.";
   endsource;
run;

   runcasl
code=myProgram;                                                     /*9*/
run;

%put NOTE: *** Compute Server Symbol Table Variable Values
***;              /*10*/
%put NOTE: &=mvar1;
%put NOTE: &=mvar2;
```

1 Set the macro variable values to `mvar1` and `mvar2` on the Compute Server.

2 If you have not already started a CAS session, specify the CAS statement to start a CAS session.

3 Specify the PROC CAS statement to define your CASL program. PROC CAS also enables you to specify CAS actions that run on the CAS server.

4 Specify the SOURCE statement to write the CASL program and store the program in the variable *myProgram*.

5 Specify the SYMPUTX function to store the specified text value in *macro-variable-name*, named memory location `mvar1`.

6 Specify the SYMPUTX function to store the specified numeric value in *macro-variable-name*, named memory location `mvar2`. CAS converts the numeric constant a string before storing it as `mvar2`.

7 Specify the SYMGET function to retrieve the values of `mvar1` and `mvar2` and store the values in CASL variables `A` and `B`, respectively.

8 Specify the PRINT statement to write the values of the variables `A` and `B` to the log.

9 Specify the runCasl action to run the program and print the results to the log. Because CAS actions can execute only in CAS, using the runCASL action ensures that this code executes in CAS.

10 Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table remain unchanged.

The results show that the leading and trailing blanks are removed from both macro values.

```
NOTE: Active Session now CASAUTO.
SYMPUTX removes leading/trailing blanks
Numeric constant 123.456 is converted to string and blanks removed.
```

Example 2: Create Macro Variables on the SAS Compute Server Using the SYMPUT Function and the SYMPUTX Function

The following example shows how the SYMPUTX function runs on the Compute Server server to create or modify a macro variable and assign it the value specified in the function. The function removes leading and trailing blanks from both the *macro-variable-name* and the *macro-variable-value*. The program compares the behavior of SYMPUTX with the SYMPUT function, which does not remove leading and trailing blanks from either the *macro-variable-name* or the *macro-variable-value*.

Note: The SAS client (the SAS Compute Server) supports full macro processing. When executed on the SAS Compute Server, the SYMPUT and SYMPUTX functions create or modify a macro variable in the Compute Server *macro symbol table*.

```
/* Set some macro variable values on the Compute Server
*/                    /*1*/
%let mvar1='Compute Server value 1';
%let mvar2='Compute Server value 2';
%put NOTE: *** Compute Server Symbol Table Variable Values ***;
%put NOTE: &=mvar1;
%put NOTE: &=mvar2;
```

```
proc
cas;
  /*2*/
   symputx('   items1   ', '    removes leading/trailing blanks
');          /*3*/
   symput('items2',   '  does NOT remove leading/trailing blanks
');       /*4*/
run;

   print "SYMPUTX "
symget("items1");                                          /*5*/
   print "SYMPUT "  symget("items2");
run;

%put NOTE: *** Compute Server Symbol Table Variable Values
***;               /*6*/
%put NOTE: &=mvar1;
%put NOTE: &=mvar2;
```

1 Set the macro variable values to `mvar1` and `mvar2` on the Compute Server.

2 Specify the PROC CAS statement to define your CASL program. PROC CAS also enables you to specify CAS actions that run on the CAS server.

3 Specify the SYMPUTX function to create the macro variable `mvar1` and store its value.

4 Specify the SYMPUT function to create the macro variable `mvar2` and store its value.

5 Specify the PRINT statement to write the values of the variables to the log. The results show that the SYMPUTX function removes leading and trailing blanks, but the SYMPUT function does not.

6 Execute macro code on the Compute Server to verify that the variables in the Compute Server Symbol Table were altered by the CASL program.

```
SYMPUTX removes leading/trailing blanks
SYMPUT  does NOT remove leading/trailing blanks    .
NOTE: *** Compute Server Symbol Table Variable Values ***
...
NOTE: MVAR1=removes leading/trailing blanks
...
NOTE: MVAR2=    does NOT remove leading/trailing blanks
```

See Also

- “SYMGET Function”
- “SYMDEL Function”
- “SYMPUT Function”
- “Using the Macro Facility on the SAS Viya Platform” in *SAS Macro Language: Reference*

TABCOLUMNS Function

Returns the names and labels of columns from a result table into a dictionary. For each item in the dictionary, the key is the name of a column, and the value is the label of the column.

Category: Result Tables

Note: The TABCOLUMNS function operates only on result tables. The function generates an error in the SAS log if a result table name was not used.

Syntax

TABCOLUMNS (*table-name*);

Required Argument

table-name
specifies the name of the table.

Example: Using TABCOLUMNS Function

The following example demonstrates how to use the TABCOLUMNS function to return the names and labels of columns from a result table into a dictionary. For each item in the dictionary, the key is the name of a column, and the value is the label of the column. A result table is required. Then, the table.Fetch action is executed and the results are placed into the variable *r*. A new variable, *findcar*, is created to place the results from the SORT_REV and FINDTABLE functions. The TABCOLUMNS function returns the names and labels of columns from the *findcar* result table and places the names and labels of columns into a new variable named *carscol*. Use the PRINT statement to list your results from the TABCOLUMNS function.

```
proc casutil;
   load data=sashelp.cars
   casout="cars"
   outcaslib="casuser"
   replace;
run;

proc cas;
  table.fetch result=r / table="cars" to=428;
   findcar=sort_rev(findtable(r), "Invoice");
  carscol=tabcolumns(findcar);
  print carscol;
run;
```

There were 16 columns in the *findcar* result table. All 16 columns are listed in the SAS log along with their labels.

Example Code 3.27 *SAS Log*

```
{_Index_=_Index_,Make=Make,Model=Model,Type=Type,Origin=Origin,DriveTrain=DriveTrain,
MSRP=MSRP,Invoice=Invoice,EngineSize=

EngineSize,Cylinders=Cylinders,Horsepower=Horsepower,MPG_City=MPG_City,MPG_Highway=MP
G_Highway,Weight=Weight,Wheelbase=Wheelbase,Len
 gth=
 Length}
```

See Also

- "FINDTABLE Function"
- "SORT REV Function"
- "PRINT Statement"

TABTYPES Function

Extracts the data types from a result table into a dictionary.

Category: Result Tables

Note: The TABTYPES function only operates on result tables. The function generates an error in the SAS log if a result table name was not used.

Syntax

TABTYPES (*table-name*);

Required Argument

table-name
specifies the name of the table.

Example: Using TABTYPES Function

The following example demonstrates using the TABTYPES function to retrieve the data types from a result table. A result table is required, therefore the table.Fetch action is executed and the results are placed into the variable *r*. A new variable, *findcar*, is created to place the results from the SORT_REV and FINDTABLE

functions. The TABTYPES function retrieves the data types for each column from the *findcar* result table and places the column names and its data types into a new variable named *carscol*. Use the DESCRIBE statement to list your results from the TABTYPES function.

```
proc casutil;
   load data=sashelp.cars
   casout="cars"
   outcaslib="casuser"
   replace;
run;

proc cas;
  table.fetch result=r / table="cars" to=428;
   findcar=sort_rev(findtable(r), "Invoice");
  carscol=tabtypes(findcar);
  describe carscol;
run;
```

There were 16 columns in the *findcar* result table. All 16 columns are listed in the SAS log along with their data types.

Example Code 3.28 *SAS Log*

```
dictionary ( 16 entries, 16 used);
 [_Index_] string;
 [Make] string;
 [Model] string;
 [Type] string;
 [Origin] string;
 [DriveTrain] string;
 [MSRP] string;
 [Invoice] string;
 [EngineSize] string;
 [Cylinders] string;
 [Horsepower] string;
 [MPG_City] string;
 [MPG_Highway] string;
 [Weight] string;
 [Wheelbase] string;
 [Length] string;
```

See Also

- "CASL Data Types" in *SAS Cloud Analytic Services: CASL Programmer's Guide*
- "DESCRIBE Statement"
- "FINDTABLE Function"
- "SORT REV Function"

TERM_PARALLEL_SESSION Function

Terminates a parallel session.

Category: Server-side

Returned data type: Number of sessions the function failed to terminate

Syntax

TERM_PARALLEL_SESSION (<*session-reference-1*>, <*session-reference-N*>);

Optional Argument

session-reference
specifies the session name to cancel.

> **TIP** Multiple session names can be listed.

Example: Terminate a Parallel Session

This example uses the TERM_PARALLEL_SESSION function to end the parallel session named session. This function is only available in server-side CASL and is meant to be submitted to the CAS server through the runCasl action.

```
cas casauto;
proc cas;
    source casl_code;
    term_parallel_session(session);
    endsource;
    sccasl.runCASL / code = casl_code;
quit;
```

TIMESTAMP Function

Returns the current date and time in a specified format.

Category: Control

Default: "string"

Returned data type: STRING, INT64

Note: SAS defines the current date and time as the number of seconds since January 1, 1960. For more information about SAS datetime formats, see "About SAS Date, Time, and Datetime Values" in *SAS Formats and Informats: Reference*.

See: *SAS Formats and Informats: Reference*

Syntax

TIMESTAMP (<*expression*> | <*format*>)

Without Arguments

The TIMESTAMP function can be specified without arguments. When no arguments are specified, the function returns the current date and time as a formatted string.

Returned data type: STRING

Example: "Example 1: Using the TIMESTAMP Function" on page 217

Optional Arguments

expression
specifies a string, a numeric value, or an expression.

See "CASL Data Types" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

Example "Example 1: Using the TIMESTAMP Function" on page 217.

"string"
returns the current date and time as a formatted string.

Returned data type STRING

"int64"
returns the current date and time as an INT64 numeric data type.

Returned data type INT64

format
specifies a date or datetime format.

Returned data type STRING

Note Do not put formats in quotation marks.

Example "Example 2: Formatting the Current Date and Time" on page 218

DATETIME*w.d*
writes datetime values in the form *ddmmmyy*:*hh*:*mm*:*ss.ss*.

w
specifies the width of the output field.

Default 16

Range 7–40

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

See "DATETIMEw.d Format" in *SAS Formats and Informats: Reference*

d
specifies the number of digits to the right of the decimal point in the seconds value. This argument is optional.

Range 0–39

Requirement *d* must be less than *w*.

Alignment Right

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

Example "Example 1: Using the TIMESTAMP Function" on page 217.

NLDATMTM*w*.
converts the time portion of the current date and time to the time-of-day value for the specified locale, and then writes the value as the time of day in the form *hh:mm:ss*.

Note: The time of day is written in 24-hour notation.

w
specifies the width of the output field.

Default 16

Range 16–200

Example "Example 2: Formatting the Current Date and Time" on page 218

Alignment Left

NLDATM*w*.
converts a SAS datetime value to the datetime value of the specified locale, and then writes the value as a datetime.

w
specifies the width of the output field. If necessary, SAS abbreviates the datetime value to fit the format width.

Default 30

Range 10–200 (inclusive)

Example "Example 2: Formatting the Current Date and Time" on page 218

Alignment Left

MDYAMPM*w.d*

writes the current date and time in the form *mm/dd/yy<yy>hh:mm* AM|PM. The year can be either two or four digits.

w

specifies the width of the output field.

Default 19

Range 8-40

d

specifies the number of digits to the right of the decimal point in the minutes value. This argument is optional.

Default 0

Range 0–39

Example "Example 2: Formatting the Current Date and Time" on page 218

Examples

Example 1: Using the TIMESTAMP Function

You can use the TIMESTAMP function with no arguments to get the current date and time. When specified without arguments, the function displays the current date in the NLDATM30. SAS datetime format.

```
proc cas;
    print timestamp();
run;
```

The following is printed to the SAS log:

```
Wed May 23 23:58:54 2018
```

You can use the TIMESTAMP function with the **int64** argument to get the current date and time as a 64-bit integer.

```
proc cas;
   print timestamp("int64");
run;
```

The following is printed to the SAS log:

```
1842739241
```

You can use the TIMESTAMP function with the **string** argument to get the current date and time as a string.

```
proc cas;
   print timestamp("string");
run;
```

The following is printed to the SAS log:

```
Thu May 24 00:01:06 2018
```

Example 2: Formatting the Current Date and Time

You can format the results of the TIMESTAMP function. The current date and time is printed to the SAS log in the format specified.

```
proc cas;
   print timestamp(datetime14.2);
   print timestamp(MDYAMPM20.);
   print timestamp(NLDATMTM16.);
   print timestamp(NLDATM30.);
run;
```

The following is printed to the SAS log:

```
20SEP19:16:13
9/20/2019  4:13 PM
16:13:18
20Sep2019:16:13:18
```

See Also

References

- "Summary of CASL Data Types" in *SAS Cloud Analytic Services: CASL Programmer's Guide*

Statements

- "PRINT Statement"

UNIQUE Function

Searches an array to determine if a value exists in the array.

Category: Array

Returned data type: If the value is not found in the array, then the Boolean value of TRUE is returned. The value TRUE means that the value is a unique value. Otherwise, FALSE is returned.

Syntax

UNIQUE (*array*, <*value*>);

Required Argument

array
specifies a countable number of values.

Optional Argument

value
specifies a character or numeric constant, variable, or expression. If value is numeric, SAS converts the value to a character string using the BEST. format.

Example

This example uses the UNIQUE function to search the array to determine if the value 0 is not found.

```
cas casauto;
proc cas;
   a=unique({1, 2, 3, 4, 5}, 0);
   print a;
run;
```

```
TRUE
```

unpackData Function

Unpacks the string, packed by base64Encode, according to the given format.

Syntax

unpackData("*name*")

Required Argument

name
specifies the name of the variable or expression where the string needs to be unpacked.

Example

```
proc cas;
   text="Hello World!";
   packed=base64Encode(text);
   unPacked=unpackData(packed);
   print unPacked;
run;
```

```
0
```

See Also

- "base64Decode Function"
- "base64Encode Function"

UPLOAD_CASLSTORE Function

Uploads CASL functions from a specified CASL source code string into the specified code storage repository.

Syntax

Form 1: **UPLOAD_CASLSTORE**({NAME=*store-name* <, CASLIB=*caslib-name*> <, REPLACE=TRUE | FALSE> <, ENCRYPT=TRUE | FALSE>}, *casl-code–1*, <, ...*casl-code-n*>);

Form 2: **UPLOAD_CASLSTORE**({PATH=*path-name* <, REPLACE=TRUE | FALSE> <, ENCRYPT=TRUE | FALSE>}, *casl-code–1*, <, ...*casl-code-n*>);

Required Arguments

location
specifies the location where the CASL code is stored.

casl-code
specifies the CASL function code to be uploaded. Multiple functions can be specified at once by separating the code with commas.

Optional Argument

ENCRYPT=TRUE | FALSE
specifies if the uploaded functions will be encrypted.

Examples

When defining CASL functions in a SOURCE block, you can terminate a FUNCTION definition with either the END statement or the ENDSUB statement. Both forms are supported when uploading functions to a CASL store.

Note: For functions that are defined within `SOURCE...ENDSOURCE` blocks, SAS recommends using `ENDSUB` because it improves compatibility with FCMP-style function definitions and CASL source code that is stored by using functions such as `UPLOAD_CASLSTORE`.

Example 1: Using the UPLOAD_CASLSTORE Function

This example defines two user-defined CASL functions in a SOURCE block and uploads them to a CASL store by using the UPLOAD_CASLSTORE function. The stored functions convert temperature values between Celsius and Fahrenheit. The example then loads the CASL store and calls the functions to perform the conversions.

```
proc cas;
   source funcs;
      function c2f(c);
          /*(°Cx9/5)+32*/
          return ((c*9/5)+32);
      end;
      function f2c(f);
          /* (°F-32)x5/9 */
          return ((f-32)*5/9);
      end;
   endsource;
   upload_caslstore({caslib="casuser",
name="store1", replace=true}, funcs);
run;

proc cas;
  cs = caslstore({caslib="casuser", name="store1"});
  DegF=37.0;
  DegC=37.0;
  f=round(c2f(DegC),.1);
  c=round(f2c(DegF),.1);
  print "NOTE: " degF "°F is " c "°C.";
  print "NOTE: " degC "°C is " f "°F.";
```

```
run;
```

Example Code 3.29 *The Log Returns These Results:*

```
NOTE: 37 °F is 2.8 °C.
NOTE: 37 °C is 98.6 °F.
```

Example 2: Using the UPLOAD_CASLSTORE Function with ENDSUB Statement

This example defines a CASL function in a SOURCE block and terminates the function definition with the ENDSUB statement. The function source code is then uploaded to a CASL store by using the UPLOAD_CASLSTORE function. The example demonstrates that ENDSUB can be used as an alternative to END when defining functions that are stored in a CASL store.

```
proc cas;
   source funcs;
      function c2f(c);
         return ((c*9/5)+32);
      endsub;
   endsource;

   upload_caslstore(
      {caslib="casuser",
       name="store1",
       replace=true},
      funcs);
run;
```

Example Code 3.30 *Log Output*

```
NOTE: The table store1 loaded into casuser with 1 observation and 6 variables.
```

See Also

- "CASLSTORE Function"
- "DEFAULT_CASLSTORE Function"
- "SOURCE Statement"

UUID Function

Returns the universally unique identifier (UUID) for the given session.

Categories:	Control Status
Returned data type:	36 character UUID

Syntax

UUID ("*session-reference*");

Required Argument

session-reference
specifies the session name to retrieve the UUID from.

Note: If a valid session name is not used, then a NULL string is returned.

Example: List the Actions in a Session's Action Queue

This example executes the listactionq action to list the actions that are currently in the queue for session Mysess, which is identified by its UUID. This example assumes that session Mysess is running in the SAS client and has one or more actions that were queued for execution using the ASYNC= parameter in the action statement.

```
*options cashost="cloud.example.com" casport=5570;
cas mysess;                                          /* 1 */
proc cas;
   session mysess;
   uuid = uuid("mysess");                            /* 2 */
   if uuid != "" then
      do;
         session.listactionq result=res / uuid=uuid; /* 3 */
         print res;
      end;
run;
quit;
```

1 If necessary, create session Casauto. Set system options CASHOST= and CASPORT= to a host and a port that are valid for your site.

2 Use function UUID to get the UUID of session Mysess, and assign the result to variable Uuid.

3 If the UUID of session Mysess was successfully obtained, use the UUID in the listactionq UUID= parameter to get the current action queue for session Mysess. Print the results.

Results

The following result is an example.

Connection ID	Sequence Number	Timestamp	Action	Action Parameter Count	Action Tag	Action is Active
4	12	08/15/19 16	myActionSet.prepareData	1	data_prep	Running
4	13	08/15/19 16	sessionProp.addFmtLib	3	load_formats	Active
4	14	08/15/19 16	simple.summary	3	summarize	Active

WAIT_FOR_NEXT_ACTION Function

Wait for a completed action.

Category: Server-side

Interaction: This function works on the client side with async actions.

Syntax

WAIT_FOR_NEXT_ACTION (*job-name*);

Required Argument

job-name
specifies the name of the job that the function waits for to be completed.

Examples

Example 1: wait_for_next_action No Job Name

```
proc cas;
       job = wait_for_next_action(0);
      do while(job); do;
                  print ;
                  print "Job [" i "]" ;
                  do  k,j over job;
                     print k " "  j;
                  end;
                  job = wait_for_next_action(0);
                  i = i + 1;
                  end;
      end;
run;
```

Example 2: wait_for_next_action With Job Name

```
proc cas;
      job = wait_for_next_action({'a','b'});
      do while(job); do;
```

```
                print ;
                print "Job [" i "]" ;
                do  k,j over job;
                   print k " "  j;
                end;
                job = wait_for_next_action(0);
                i = i + 1;
                end;
      end;
run;
```

4

Common Functions

Overview

A function is a component of the CASL programming language that can accept arguments, perform a computation or other operation, and return a value. The value that is returned can be used in an assignment statement or elsewhere in expressions. The CASL language provides built-in functions that provide functionality that is unique to CASL, and functions that provide functionality found throughout SAS software. You can also create your own user-defined functions. Here are some examples of CASL functions:

Function type	Example	See
built-in	`readpath ("/u/sasdemo/ds/samplecode.sas");`	"CASL Built-In Functions" (p. 115)
common	`datetime();`	"Common Functions" (p. 227)
user-defined	`SharedBday(365,n);`	"FUNCTION Statement" (p. 63)

Many CASL statements and functions can be executed client-side. Because these functions do not require interaction with the CAS server, they run as CASL scripts without the need to establish a CAS session. Functions that interact with CAS tables or perform distributed computations require an active CAS session to ensure they are executed in the correct environment. Built-in CASL functions categorized as server-side functions on page 121 always require a CAS session to execute.

Date, Time, and Datetime Constants

A date constant, time constant, or datetime constant is a date or time or datetime in single or double quotation marks, followed by a D (date), T (time), or DT (datetime) to indicate the type of value. For an example of how to use D, T, and DT to create date, time, and datetime constants, see "Example: Define Date, Time, and Datetime Values in Date Constants" in *SAS Programmer's Guide: Essentials*.

Constant	Format	Examples
Date	*'ddmmm<yy>yy'*D "*ddmmm<yy>yy*"D	`'1jan2018'D` `"01jan18"D`
Time	*'hh:mm<:ss.s>'*T "*hh:mm<:ss.s>*"T	`'9:25'T` `"9:25:19pm"T`
Datetime	*'ddmmm<yy>yy:hh:mm<:ss.s>'*DT "*ddmmm<yy>yy:hh:mm<:ss.s>*"DT	`'01may18:9:30:00'DT` `"18jan2018:9:27:05am"DT`
Datetime with time zone designators (ISO 8601 standard)	*'yyyy-mm-dd*T*hh:mm:ss*Z'DT *'yyyy-mm-dd*T*hh:mm:ss*+|-*hh:ss'*DT	`'2018-07-20T12:00:00Z'DT` `'2018-05-17T09:15:30-05:00'DT`

IMPORTANT UTC or ISO 8601 Datetime constants, which have the Zulu timezone indication or a numeric offset from the Universal Coordinate Time, are converted to local time by adjusting the internal value according to the system timezone offset. This adjustment occurs regardless of whether the system TIMEZONE option has been explicitly set. If you have specified the TIMEZONE= system option, then SAS converts UTC and ISO 8601 DATETIME constants based on the value that you specify in the TIMEZONE= system option. Otherwise, if you do not specify the TIMEZONE= system option, then SAS converts UTC and ISO 8601 DATETIME constants based on the system UTC time zone offset.

Trailing blanks or leading blanks that are included within the quotation marks do not affect the processing of the date constant, time constant, or datetime constant.

See Also

"Example: Define Date, Time, and Datetime Values in Date Constants" in *SAS Programmer's Guide: Essentials*

Function Categories

Table 4.1 *Category and Definition Table*

Category	Definition
CAS	CALL routines and functions that run on the CAS server. See CAS for a list of functions.
Character	Returns information based on character data. See Character 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.
Financial	Calculates financial values such as interest, periodic payments, depreciation, and prices for European options on stocks. See Financial for a list of functions.
Random Number	Returns random variates from specific distributions. See Random Number for a list of functions.
Truncation	Truncates numeric values and returns numeric values, often using fuzzing or zero fuzzing. See Truncation for a list of functions.

Category	Language Elements	Description
CAS	BLACKPTPRC Function (p. 276)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 279)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	COMPOUND Function (p. 316)	Returns compound interest parameters.
	FUZZ Function (p. 390)	Returns the nearest whole number if the argument is within 1E-12 of that number.
	GARKHCLPRC Function (p. 394)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
	MEDIAN Function (p. 499)	Returns the median value.
	SMALLEST Function (p. 618)	Returns the kth smallest non-null or nonmissing value.
	STDERR Function (p. 621)	Returns the standard error of the mean.
	UNIFORM Function (p. 650)	Returns a random variate from a uniform distribution.
Character	CAT Function (p. 288)	Does not remove leading or trailing blanks, and returns a concatenated character string.
Descriptive Statistics	SMALLEST Function (p. 618)	Returns the kth smallest non-null or nonmissing value.
	STDERR Function (p. 621)	Returns the standard error of the mean.
Financial	BLACKPTPRC Function (p. 276)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 279)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	COMPOUND Function (p. 316)	Returns compound interest parameters.
	GARKHCLPRC Function (p. 394)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
Random Number	UNIFORM Function (p. 650)	Returns a random variate from a uniform distribution.
Truncation	FUZZ Function (p. 390)	Returns the nearest whole number if the argument is within 1E-12 of that number.

Dictionary

ABS Function

Returns the absolute value.

Returned data type: BIGINT, DECIMAL, DOUBLE, NUMERIC

Syntax

ABS(*expression*)

Required Argument

expression
specifies a numeric constant, variable, or expression.

Data type: BIGINT, DECIMAL, DOUBLE, NUMERIC

Details

If the result is a number that does not fit into the range of the argument's data type, the ABS function fails.

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Example

```
proc cas;
   x=abs(2.4);
   y=abs(-3);
   print "x=" x;
   print "y=" y;
```

```
run;
```

The following SAS statements produce these results:

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

AIRY Function

Returns the value of the Airy function.

Syntax

AIRY(*x*)

Required Argument

x
specifies a numeric constant, variable, or expression.

Example

The following SAS statements produce these results:

```
proc cas;
   x=airy(2.0);
   y=airy(-2.0);
   print "x=" x;
   print "y=" y;
run;
```

The following output is printed to the SAS Log:

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

ANYALNUM Function

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

Returned data type:	DOUBLE
Note:	This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYALNUM function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYALNUM function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase letter. If such a character is found, ANYALNUM returns the position in the string of that character. If no such character is found, ANYALNUM returns a value of 0.

If you use only one argument, ANYALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALNUM function searches a character string for an alphanumeric character. The NOTALNUM function searches a character string for a non-alphanumeric character.

Example

```
proc cas;
   strings={'~1st line---'
           ,'~~2nd line--'
           ,'~~~3rd line-'};
   do thisString over strings;
      start=anyalnum(thisString);                              /* 1 */
      length=anyalnum(thisString,-99999)-start+1;              /* 2 */
      phrase=substr(thisString,start,length);                  /* 3 */
      print "String: '" thisString "', Phrase: '" phrase"'";
   end;
run;
```

1 The ANYALNUM function finds the position of the first alphanumeric character in `thisString` and assigns the value to the `start` variable.

2 The large negative start position (-99999) causes the ANYALNUM function to search backwards from the end of `thisString` to find the position of the last alphanumeric character in the string. Subtracting this value from `start` and adding 1 identifies the length of the alphanumeric phrase.

3 Use the calculated `start` and `length` values to extract `phrase` from `thisString`.

The program writes the following output to the log:

```
String: '~1st line---', Phrase: '1st line'
String: '~~2nd line--', Phrase: '2nd line'
String: '~~~3rd line-', Phrase: '3rd line'
```

ANYALPHA Function

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

Returned data type:	DOUBLE
Note:	This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYALPHA function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYALPHA function searches a string for the first occurrence of any character that is an uppercase or lowercase letter. If such a character is found, ANYALPHA returns the position in the string of that character. If no such character is found, ANYALPHA returns a value of 0.

If you use only one argument, ANYALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALPHA function searches a character string for an alphabetic character. The NOTALPHA function searches a character string for a non-alphabetic character.

Example: Searching a String for Alphabetic Characters

```
proc cas;
   strings={' 1 January  2024'
           ,'23 February 2024'
           ,'   16March2024  '};
   do thisString over strings;
      start=anyalpha(thisString);                          /* 1 */
      length=anyalpha(thisString,-99999)-start+1;          /* 2 */
      phrase=substr(thisString,start,length);              /* 3 */
      print "String: '" thisString "', Phrase: '" phrase"'";
   end;
run;
```

1 The ANYALPHA function finds the position of the first alphabetic character in `thisString` and assigns the value to the `start` variable.

2 The large negative start position (-99999) causes the ANYALPHA function to search backwards from the end of `thisString` to find the position of the last alphabetic character in the string. Subtracting this value from `start` and adding 1 identifies the length of the alphabetic phrase.

3 Use the calculated `start` and `length` values to extract the `phrase` from `thisString`.

The program writes the following output to the log:

```
String: ' 1 January  2024', Phrase: 'January'
String: '23 February 2024', Phrase: 'February'
String: '   16March2024  ', Phrase: 'March'
```

See Also

"NOTALPHA Function"

ANYCNTRL Function

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

Data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYCNTRL function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYCNTRL function searches a string for the first occurrence of a control character. If such a character is found, ANYCNTRL returns the position in the string of that character. If no such character is found, ANYCNTRL returns a value of 0.

If you use only one argument, ANYCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYCNTRL returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.

- The value of *start* = 0.

Comparisons

The ANYCNTRL function searches a character string for a control character. The NOTCNTRL function searches a character string for a character that is not a control character.

Example

The following example uses the ANYCNTRL function to search for the first control character in a string.

```
proc cas;
   strings={'123'||'01'x||'567'
           ,'B 1AB '
           ,' '||'09'x||' -~-'};
   do thisString over strings;
      if anycntrl(thisString)>0 then
         print "Control character found in '" thisString "'";
         else print "No control characters found in '" thisString"'";
   end;
run;
```

The program writes the following output to the log:

```
Control character found in '123567'
No control characters found in 'B 1AB '
Control character found in '  -~-'
```

See Also

"NOTCNTRL Function"

ANYDIGIT Function

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

Returned data type:	DOUBLE
Note:	This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

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

The ANYDIGIT function searches a string for the first occurrence of any character that is a digit. If such a character is found, ANYDIGIT returns the position in the string of that character. If no such character is found, ANYDIGIT returns a value of 0.

If you use only one argument, ANYDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYDIGIT function searches a character string for a digit. The NOTDIGIT function searches a character string for any character that is not a digit.

Example

The following example uses the ANYDIGIT function to search for a character that is a digit.

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anydigit(string, j+1);
      if j=0 then print +3 "The end";
      else do;
         c=substr(string, j, 1);
         print "j=" j "c=" c;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=14 c=1
j=15 c=2
j=17 c=3
The end
```

See Also

“NOTDIGIT Function”

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.

Data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

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

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_) and uppercase or lowercase English letters. If such a character is found, ANYFIRST returns the position in the string of that character. If no such character is found, ANYFIRST returns a value of 0.

If you use only one argument, ANYFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7. The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Example

The following 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.

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anyfirst(string, j+1);
      if j=0 then print 'The end';
      else do;
         c=substr(string, j, 1);
         print "j=" j;
         print "c=" 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=8 c=_
j=9 c=n
j=10 c=_
j=16 c=E
The end
```

See Also

“NOTFIRST Function”

ANYGRAPH Function

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

Returned data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYGRAPH function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYGRAPH function searches a string for the first occurrence of a graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, ANYGRAPH returns the position in the string of that character. If no such character is found, ANYGRAPH returns a value of 0.

If you use only one argument, ANYGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYGRAPH function searches a character string for a graphical character. The NOTGRAPH function searches a character string for a non-graphical character.

Example: Searching a String for Graphical Characters

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

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anygraph(string, j+1);
      if j=0 then print 'The end';
      else do;
         c=substr(string, j, 1);
         print "j=" j;
         print "c=" 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=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c=+
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

See Also

“NOTGRAPH Function”

ANYLOWER Function

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

Returned data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYLOWER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYLOWER function searches a string for the first occurrence of a lowercase letter. If such a character is found, ANYLOWER returns the position in the string of that character. If no such character is found, ANYLOWER returns a value of 0.

If you use only one argument, ANYLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYLOWER function searches a character string for a lowercase letter. The NOTLOWER function searches a character string for a character that is not a lowercase letter.

Example

The following example uses the ANYLOWER function to search a string for any character that is a lowercase letter.

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anylower(string, j+1);
      if j=0 then print "The end";
      else do;
         c=substr(string, j, 1);
         print "j=" j "c=" 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
The end
```

See Also

"NOTLOWER Function"

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.

Returned data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

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

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. These characters are the underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, ANYNAME returns the position in the string of that character. If no such character is found, ANYNAME returns a value of 0.

If you use only one argument, ANYNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7. The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

Example

The following example uses the ANYNAME function to search a string for any character that is valid in a SAS variable name under VALIDVARNAME=V7.

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anyname(string, j+1);
      if j=0 then print 'The end';
      else do;
         c=substr(string, j, 1);
         print "j=" j "c=" 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=8 c=_
j=9 c=n
j=10 c=_
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
The end
```

See Also

"NOTNAME Function"

ANYPRINT Function

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

Returned data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYPRINT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYPRINT function searches a string for the first occurrence of a printable character. If such a character is found, ANYPRINT returns the position in the string of that character. If no such character is found, ANYPRINT returns a value of 0.

If you use only one argument, ANYPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYPRINT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYPRINT function searches a character string for a printable character. The NOTPRINT function searches a character string for a non-printable character.

Example: Searching a String for a Printable Character

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

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anyprint(string, j+1);
      if j=0 then print 'The end';
      else do;
         c=substr(string, j, 1);
         print "j=" j;
         print "c=" 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=16 c=E
   j=17 c=3
   j=18 c=;
   The end
```

See Also

"NOTPRINT Function"

ANYPUNCT Function

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

Returned data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Returned data type DOUBLE

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYPUNCT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYPUNCT function searches a string for the first occurrence of a punctuation character. If such a character is found, ANYPUNCT returns the position in the string of that character. If no such character is found, ANYPUNCT returns a value of 0.

If you use only one argument, ANYPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYPUNCT function searches a character string for a punctuation character. The NOTPUNCT function searches a character string for a character that is not a punctuation character.

Example: Searching a String for Punctuation Characters

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

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anypunct(string, j+1);
      if j=0 then print "The end";
      else do;
         c=substr(string, j, 1);
         print "j=" j;
         print "c=" c;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=8 c=_
j=10 c=_
j=18 c=;
The end
```

See Also

“NOTPUNCT Function”

ANYSPACE Function

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

Returned data type: DOUBLE

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Returned data type CHAR, NCHAR

Optional Argument

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYSPACE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYSPACE function searches a string for the first occurrence of any character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, ANYSPACE returns the position in the string of that character. If no such character is found, ANYSPACE returns a value of 0.

If you use only one argument, ANYSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or

form feed. The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Example: Searching a String for a White Space Character

The following example uses the ANYSPACE function to search a string for a character that is a white space character.

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anyspace(string, j+1);
      if j=0 then print "The end";
      else do;
         c=substr(string, j, 1);
         print "j=" j;
         print "c=" c;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=7 c=
j=11 c=
j=13 c=
The end
```

See Also

“NOTSPACE Function”

ANYUPPER Function

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

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
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the ANYUPPER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and the LOCALE system options.

The ANYUPPER function searches a string for the first occurrence of an uppercase letter. If such a character is found, ANYUPPER returns the position in the string of that character. If no such character is found, ANYUPPER returns a value of 0.

If you use only one argument, ANYUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYUPPER function searches a character string for an uppercase letter. The NOTUPPER function searches a character string for a character that is not an uppercase letter.

Example

The following example uses the ANYUPPER function to search a string for an uppercase letter.

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anyupper(string, j+1);
      if j=0 then print "The end";
      else do;
         c=substr(string, j, 1);
         print "j=" j  "c=" c;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=16 c=E
The end
```

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.

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
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

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

The ANYXDIGIT function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, ANYXDIGIT returns the position in the string of that character. If no such character is found, ANYXDIGIT returns a value of 0.

If you use only one argument, ANYXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYXDIGIT function searches a character string for a character that is a hexadecimal character. The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character.

Example

The following example uses the ANYXDIGIT function to search a string for a hexadecimal character that represents a digit.

```
proc cas;
```

```
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=anyxdigit(string, j+1);
         if j=0 then print "The end";
         else do;
            c=substr(string, j, 1);
            print "j=" j   "c=" 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
The end
```

ARCOS Function

Returns the arccosine.

Returned data type: DOUBLE

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

```
proc cas;
   x=arcos(1);
   y=arcos(0);
   z=arcos(-0.5);
   print "x=" x;
   print "y=" y;
   print "z=" z;
run;
```

SAS writes the following output to the log:

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

ARCOSH Function

Returns the inverse hyperbolic cosine.

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 defined 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.

```
proc cas;
   x=arcosh(5);
   x1=arcosh(13);
```

```
   print "x=" x;
   print "y=" y;
run;
```

SAS writes the following output to the log:

```
x=2.2924316696
y=3.2566139548
```

ARSIN Function

Returns the arcsine.

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

```
proc cas;
   x=arsin(0);
   y=arsin(1);
   z=arsin(-0.5);
   print "x=" x;
   print "y=" y;
   print "z=" z;
run;
```

SAS writes the following output to the log:

```
x=0 y=1.5707963268 z=-0.523598776
```

ARSINH Function

Returns the inverse hyperbolic sine.

Returned data type: DOUBLE

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 defined by the following equation, where $-\infty < x < \infty$

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

Replace the infinity symbol with the largest double precision number that is available on your machine.

Example

The following example computes the inverse hyperbolic sine.

```
proc cas;
   x=arsinh(5);
   y=arsinh(-5);
   print "x=" x;
   print "y=" y;
run;
```

SAS writes the following output to the log:

```
x=2.3124383413
y=-2.312438341
```

ARTANH Function

Returns the inverse hyperbolic tangent.

Returned data type: DOUBLE

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 defined 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.

```
proc cas;
   x=artanh(0.5);
   print "x=" x;
run;
```

SAS writes the following output to the log:

```
x=0.5493061443
```

ATAN Function

Returns the arc tangent.

Returned data type: DOUBLE

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 - π /2 to π /2. If the argument is missing, then 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

The following SAS statements produce these results:

```
proc cas;
   x=atan(0);
   print "x=" x;
run;
```

```
x=0
```

```
proc cas;
   x=atan(1);
```

```
   print "x=" x;
run;
```

```
x=0.7853981634
```

```
proc cas;
   x=atan(-9.0);
   print "x=" x;
run;
```

```
x=-1.460139106
```

ATAN2 Function

Returns the arc tangent of the ratio of two numeric variables.

Returned data type: DOUBLE

Syntax

ATAN2(*argument-1*, *argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

argument-2
specifies a numeric constant, variable, or expression.

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 - π and π. If either of the arguments in ATAN2 is missing, then 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

The following SAS statements produce these results:

```
proc cas;
   a=atan2(-1, 0.5);
run;
```

```
-1.107148718
```

```
proc cas;
   b=atan2(6, 8);
run;
```

```
0.6435011088
```

```
proc cas;
   c=atan2(5, -3);
run;
```

```
2.1112158271
```

BAND Function

Returns the bitwise logical AND of two arguments.

Returned data type: DOUBLE

Syntax

BAND(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

Example

The following statements illustrate the BAND function:

```
proc cas;
   x=band(9,11);
run;
```

SAS writes the following output to the log:

```
9
```

```
proc cas;
   x=band(15,5);
run;
```

SAS writes the following output to the log:

```
5
```

BETA Function

Returns the value of the beta function.

Returned data type: DOUBLE

Syntax

BETA(*a*, *b*)

Arguments

a

is the first shape parameter.

Range *a* >0

Data type DOUBLE

b

is the second shape parameter.

Range *b* >0

Data type DOUBLE

Details

The BETA function is mathematically given by this equation:

$$\beta(a,b) = \int_0^1 x^{a-1}(1-x)^{b-1}dx$$

Note the following:

$$\beta(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

In the previous equation, $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, BETA returns a missing value.

Example

The following statements illustrate the BETA function:

```
proc cas;
   x=beta(5,3);
   y=logbeta(5,3);
   print "x=" x "y=" y;
run;
```

The following line is written to the SAS log.

```
x=0.0095238095 y=-4.65396035
```

BETAINV Function

Returns a quantile from the beta distribution.

Returned data type: DOUBLE

Syntax

BETAINV(*p*, *a*, *b*)

Arguments

p

is a numeric probability.

Range $0 \le p \le 1$

Data type DOUBLE

a

is a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

b
is a numeric shape parameter.

Range $b > 0$

Data type DOUBLE

Details

The BETAINV function returns the *p*th quantile from the beta distribution with shape parameters *a* and *b*. The probability that an observation from a beta distribution is less than or equal to the returned quantile is *p*.

Note: BETAINV is the inverse of the PROBBETA function.

Example

The following example illustrates the BETAINV function.

```
proc cas;
   y=betainv(0.001, 2, 4);
   print "y=" y;
run;
```

The following line is written to the SAS log.

```
y= 0.0101017879
```

BLACKCLPRC Function

Calculates call prices for European options on futures, based on the Black model.

Returned data type: DOUBLE

Syntax

BLACKCLPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E
is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

Data type DOUBLE

t
is a nonmissing value that specifies time to maturity, in years.

Data type DOUBLE

F
is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

Data type DOUBLE

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Data type DOUBLE

Details

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$\text{CALL} = \varepsilon^{-rt}(FN(d_1) - EN(d_2))$$

Arguments

F
specifies future price.

N
specifies the cumulative normal density function.

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate, 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:

$$\text{CALL} = \max((F - E), 0)$$

Comparisons

The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following statements illustrate the BLACKCLPRC function:

```
proc cas;
   a=blackclprc(50, .25, 48, .05, .25);
   b=blackclprc(9, 1/12, 10, .05, .2);
   print "a=" a;
   print "b=" b;
run;
```

The following line is written to the SAS log.

```
a=1.5513014272
b=1.0031514194
```

BLACKPTPRC Function

Calculates put prices for European options on futures, based on the Black model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

BLACKPTPRC(*E*, *t*, *F*, *r*, *sigma*)

Arguments

E
is a nonmissing, positive value that specifies exercise price.

Requirement: Specify *E* and *F* in the same units.

Data type: DOUBLE

t
is a nonmissing value that specifies time to maturity, in years.

Data type: DOUBLE

F
is a nonmissing, positive value that specifies future price.

Requirement: Specify *F* and *E* in the same units.

Data type: DOUBLE

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type: DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Data type: DOUBLE

Details

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} + \varepsilon^{-rt}(E - F)$$

Arguments

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate, which is an annual rate that is expressed in terms of continuous compounding.

t
specifies the time to expiration, in years.

F
specifies future price.

$$d_1 = \frac{\left(\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of t=0, the following equation is true:

$$\text{PUT} = \max((E - F), 0)$$

Comparisons

The BLACKPTPRC function calculates put prices for European options on futures, based on the Black model. The BLACKCLPRC function calculates call prices for European options on futures, based on the Black model. These functions return a scalar value.

Example

The following statements illustrate the BLACKPTPRC function:

```
proc cas;
   a=blackptprc(298, .25, 350, .06, .25);
   b=blackptprc(145, .5, 170, .05, .2);
   print "a=" a
   print "b=" b;
run;
```

The following line is written to the SAS log.

```
a=1.8598056393
b=1.4123497991
```

BLKSHCLPRC Function

Calculates call prices for European options on stocks, based on the Black-Scholes model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

BLKSHCLPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t
is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S
is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

r
is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$\text{CALL} = SN(d_1) - EN(d_2)\varepsilon^{-rt}$$

Arguments

S
is a nonmissing, positive value that specifies the share price.

N
specifies the cumulative normal density function.

E
is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration, in years.

r
specifies the risk-free interest rate, 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:

$$\text{CALL} = \max((S - E), 0)$$

Comparisons

The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following statements illustrate the BLKSHCLPRC function:

```
proc cas;
   a=blkshclprc(50, .25, 48, .05, .25);
   b=blkshclprc(9, 1/12, 10, .05, .2);
   print "a=" a;
   print "b=" b;
run;
```

The following line is written to the SAS log.

```
a=1.7989420195
b=1.0435083341
```

BLKSHPTPRC Function

Calculates put prices for European options on stocks, based on the Black-Scholes model.

Returned data type: DOUBLE

Syntax

BLKSHPTPRC(*E*, *t*, *S*, *r*, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement: Specify *E* and *S* in the same units.

Data type: DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type: DOUBLE

S

is a nonmissing, positive value that specifies the share price.

Requirement: Specify *S* and *E* in the same units.

Data type: DOUBLE

r

is a nonmissing, positive value that specifies the annualized risk-free interest rate, continuously compounded.

Data type: DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Data type DOUBLE

Details

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} - S + E\varepsilon^{-rt}$$

Arguments

S
is a nonmissing, positive value that specifies the share price.

E
is a nonmissing, positive value that specifies the exercise price of the option.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(r + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration, in years.

r
specifies the risk-free interest rate, 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:

$$\text{PUT} = \max((E - S), 0)$$

Comparisons

The BLKSHPTPRC function calculates the put prices for European options on stocks, based on the Black-Scholes model. The BLKSHCLPRC function calculates the call prices for European options on stocks, based on the Black-Scholes model. These functions return a scalar value.

Example

The following statements illustrate the BLKSHPTPRC function:

```
proc cas;
   a=blkshptprc(230, .5, 290, .04, .25);
   b=blkshptprc(350, .3, 400, .05, .2);
   print "a=" a;
   print "b=" b;
run;
```

The following line is written to the SAS log.

```
a=1.5659744295
b=1.6409194307
```

BLSHIFT Function

Returns the bitwise logical left shift of two arguments.

Returned data type: DOUBLE

Syntax

BLSHIFT(*expression-1*, *expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

expression-2
specifies any valid expression that evaluates to a numeric value.

Range 0 to 31, inclusive

Data type DOUBLE

Example

The following statement illustrates the BLSHIFT function:

```
proc cas;
   x=blshift(7,2);
   print "x=" x;
run;
```

The following line is written to the SAS log.

```
x=28
```

BNOT Function

Returns the bitwise logical NOT of an argument.

Returned data type: DOUBLE

Syntax

BNOT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

Example

The following statement illustrates the BNOT function:

```
proc cas;
   a=bnot(16);
   print a;
run;
```

The following line is written to the SAS log.

```
4294967279
```

BOR Function

Returns the bitwise logical OR of two arguments.

Returned data type: DOUBLE

Syntax

BOR(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

Example

The following statement illustrates the BOR function:

```
proc cas;
   a=bor(4,8);
   print "a=" a;
run;
```

The following line is written to the SAS log.

```
a=12
```

BRSHIFT Function

Returns the bitwise logical right shift of two arguments.

Returned data type: DOUBLE

Syntax

BRSHIFT(*expression-1*, *expression-2*)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

expression-2
specifies any valid expression that evaluates to a numeric value.

Range 0 to 31, inclusive

Data type DOUBLE

Example

The following statement illustrates the BRSHIFT function:

```
proc cas;
   x=brshift(64,2);
   print "x=" x;
run;
```

The following line is written to the SAS log.

```
x=16
```

BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two arguments.

Returned data type: DOUBLE

Syntax

BXOR(*expression-1, expression-2*)

Arguments

expression-1, expression-2
specifies any valid expression that evaluates to a numeric value.

Range between 0 and $(2^{32})-1$ inclusive

Data type DOUBLE

Example

The following statement illustrates the BXOR function:

```
proc cas;
   x=bxor(128,64);
   print "x=" x;
run;
```

The following line is written to the SAS log.

```
x=192
```

BYTE Function

Returns one character in the ASCII or the EBCDIC collating sequence.

Returned data type: STRING, VARCHAR

Syntax

BYTE(*n*)

Arguments

n

specifies a whole number that represents a specific ASCII or EBCDIC character.

Range 0–255

Data type DOUBLE

Details

For EBCDIC collating sequences, *n* is between 0 and 255. For ASCII collating sequences, the characters that correspond to values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to values between 128 and 255 are available on certain ASCII operating environments, but the information those characters represent varies with the operating environment.

Example

The following statement illustrates the BYTE function:

```
proc cas;
   x=byte(80);
   print "x=" x;
run;
```

The following line is written to the SAS log.

```
x=P
```

CAT Function

Does not remove leading or trailing blanks, and returns a concatenated character string.

Category: Character

Returned data type: STRING, VARCHAR

Syntax

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

Arguments

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CAT function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CAT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CAT function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses

- up to 65534 characters when CAT is called from the macro processor

If CAT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CAT finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank line in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CAT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Example

The following example shows how the CAT function concatenates strings.

```
proc cas;
      x='  The 2012 Olym';
      y='pic Arts Festi';
      z='  val included works by D  ';
      a='ale Chihuly.';
      result=cat(x,y,z,a);
      print "result=" result;
   end;
run;
```

SAS writes the following output to the log:

```
result=  The 2012 Olympic Arts Festi  val included works by D  ale Chihuly.
```

CATQ Function

Concatenates character or numeric values by using a delimiter to separate items and by adding quotation marks to strings that contain the delimiter.

Returned data type: STRING, VARCHAR

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, then CATQ does not insert delimiters for item arguments that have a length of zero after processing that is specified by other modifiers. The M modifier can cause delimiters to appear at the beginning or end of the result and can cause multiple consecutive delimiters to appear in the result.

n or N
converts item arguments to name literals when the value does not conform to the usual syntactic conventions for a SAS name. A name literal is a string in quotation marks that is followed by the letter "n" without any intervening blanks. To use name literals in SAS statements, you must specify the SAS option, VALIDVARNAME=ANY.

q or Q
adds quotation marks to item arguments that already contain quotation marks.

s or S
strips leading and trailing blanks from subsequently processed arguments:

- To strip leading and trailing blanks from the delimiter argument, specify the S modifier *before* the D modifier.
- To strip leading and trailing blanks from the item arguments but *not* from the delimiter argument, specify the S modifier *after* the D modifier.

t or T
trims trailing blanks from subsequently processed arguments:

- To trim trailing blanks from the delimiter argument, specify the T modifier before the D modifier.
- To trim trailing blanks from the item arguments but not from the delimiter argument, specify the T modifier after the D modifier.

x or X
converts item arguments to hexadecimal literals when the value contains nonprintable characters.

Tips If *modifier* is a constant, enclose it in quotation marks. You can also express *modifier* as a variable name or an expression.

The A, B, N, Q, S, T, and X modifiers operate internally to the CATQ function. If an item argument is a variable, then the value of that variable is not changed by CATQ unless the result is assigned to that variable.

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Optional Argument

delimiter
specifies a character constant, variable, or expression that is used as a delimiter between concatenated strings. If you specify this argument, then you must also specify the D modifier.

Details

Length of Returned Variable

The CATQ function returns a value to a variable or if CATQ is called inside an expression, CATQ returns a value to a temporary buffer. The value that is returned has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in the DATA step except in WHERE clauses
- up to 65534 characters when CATQ is called from the macro processor

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, then SAS does the following steps:

- changes the result to a blank value in the DATA step and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1 in the DATA step

If CATQ returns a value in a temporary buffer, then the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATQ finishes processing. In this case, SAS does not write a message about the truncation to the log.

The Basics

If you do not use the C, D, or H modifiers, then CATQ uses a blank as a delimiter.

If you specify neither a quotation mark in *modifier* nor the 1 or 2 modifiers, then CATQ decides independently for each item argument which type of quotation mark to use, if quotation marks are required. The following rules apply:

- CATQ uses single quotation marks for strings that contain an ampersand (&) or percent (%) sign, or that contain more double quotation marks than single quotation marks.
- CATQ uses double quotation marks for all other strings.

The CATQ function initializes the result to a length of zero and then performs the following actions for each item argument:

1. If *item* is not a character string, then CATQ converts *item* to a character string by using the BEST*w*. format and removes leading blanks.
2. If you used the S modifier, then CATQ removes leading blanks from the string.
3. If you used the S or T modifiers, then CATQ removes trailing blanks from the string.
4. CATQ determines whether to add quotation marks based on the following conditions:

- If you use the X modifier and the string contains control characters, then the string is converted to a hexadecimal literal.
- If you use the N modifier, then the string is converted to a name literal if either of the following conditions is true:
  - The first character in the string is not an underscore or an English letter.
  - The string contains any character that is not a digit, underscore, or English letter.
- If you did not use the X or the N modifiers, then CATQ adds quotation marks to the string if any of the following conditions is true:
  - You used the A modifier.
  - You used the B modifier and the string contains leading or trailing blanks that were not removed by the S or T modifiers.
  - You used the Q modifier and the string contains quotation marks.
  - The string contains a substring that equals the delimiter with leading and trailing blanks omitted.

5 For the second and subsequent item arguments, CATQ appends the delimiter to the result if either of the following conditions is true:

- You used the M modifier.
- The string has a length greater than zero after it has been processed by the preceding steps.

6 CATQ appends the string to the result.

Comparisons

The CATX function is similar to the CATQ function except that CATX does not add quotation marks.

Example: Concatenating Strings with the CATQ Function

The following example shows how the CATQ function concatenates strings.

```
proc cas;
         result1=CATQ(' ',
                   'noblanks',
                   'one blank',
                   12345,
                   '  lots of blanks    ');
         print "result1=" result1;
         result2=CATQ('CS',
                   'Period (.)                ',
                   'Ampersand (&)             ',
```

```
                    'Comma (,)                  ',
                    'Double quotation marks (") ',
                    '   Leading Blanks');
          print "result2=" result2;
          result3=CATQ('BCQT',
                    'Period (.)                 ',
                    'Ampersand (&)              ',
                    'Comma (,)                  ',
                    'Double quotation marks (") ',
                    '   Leading Blanks');
          print "result3=" result3;
          result4=CATQ('ADT',
                    '#=#',
                    'Period (.)                 ',
                    'Ampersand (&)              ',
                    'Comma (,)                  ',
                    'Double quotation marks (") ',
                    '   Leading Blanks');
          print "result4=" result4;
          result5=CATQ('N',
                    'ABC_123   ',
                    '123    ',
                    'ABC 123');
          print "result5=" result5;
       end;
run;
```

SAS writes the following output to the log:

```
result1=noblanks "one blank" 12345 " lots of blanks "

result2=Period (.),Ampersand (&),"Comma (,)",Double quotation marks (),Leading
Blanks

result3=Period (.),Ampersand (&),"Comma (,)",'Double quotation marks (')'," Leading
Blanks"

result4="Period (.)"#=#'Ampersand (&)'#=#"Comma (,)"#=#'Double quotation marks
(")'#=#" Leading Blanks"

result5=ABC_123 "123"n "ABC 123"n
```

CATS Function

Removes leading and trailing blanks, and returns a concatenated character string.

Returned data type: STRING, VARCHAR

Syntax

CATS(*item*<, ...*item*>)

Arguments

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CATS function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CATS function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATS function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CATS is called from the macro processor

If CATS returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATS finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CATS function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Example

The following example shows how the CATS function concatenates strings.

```
proc cas;
      x='  The   Olym';
      y='pic Arts Festi';
      z='  val includes works by D  ';
      a='ale Chihuly.';
    result=cats(x,y,z,a);
    print "result=" result;
   end;
run;
```

SAS writes the following output to the log:

```
result=The   Olympic Arts Festival includes works by Dale Chihuly.
```

CATT Function

Removes trailing blanks, and returns a concatenated character string.

Returned data type: STRING, VARCHAR

Syntax

CATT(*item*<, ...*item*>)

Arguments

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*.

format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CATT function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CATT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATT function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CATT is called from the macro processor

If CATT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATT finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CATT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Example

The following example shows how the CATT function concatenates strings.

```
proc cas;
      x='  The 2012 Olym';
      y='pic Arts Festi';
      z='  val included works by D  ';
      a='ale Chihuly.';
      result=catt(x,y,z,a);
      print "result=" result;
   end;
run;
```

SAS writes the following output to the log:

```
result=  The 2012 Olympic Arts Festi  val included works by Dale Chihuly.
```

CATX Function

Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.

Returned data type: STRING, VARCHAR

Syntax

CATX(*delimiter*, *item-1*<, ...*item-n*>)

Arguments

delimiter
specifies a character string that is used as a delimiter between concatenated items.

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, SAS does not write a note to the log.

Details

The Basics

The CATX function first copies *item-1* to the result, omitting leading and trailing blanks. Then for each subsequent argument *item-i*, i=2, ..., n, if *item-i* contains at least one non-blank character, then CATX appends *delimiter* and *item-i* to the result, omitting leading and trailing blanks from *item-i*. CATX does not insert the delimiter at the beginning or end of the result. Blank items do not produce delimiters at the beginning or end of the result, nor do blank items produce multiple consecutive delimiters.

Length of Returned Variable

The CATX function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATX function can be up to 32767 characters, except in WHERE clauses.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS truncates the result.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operator.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT.

Note: In the case of variables that have missing values, the concatenation produces different results.

Example

The following example shows how the CATX function concatenates strings. The first data program creates the Values table. The second and third data programs use the Values table as input.

```
proc cas;
   separator='%%$%%';
   x='The Olympic  ';
   y='   Arts Festival ';
   z='   includes works by ';
   a='Dale Chihuly.';
   result=catx(separator, x, y, z, a);
```

```
    print "result=" result;
run;
```

SAS writes the following output to the log:

```
result= The Olympic%%$%%Arts Festival%%$%%includes works by%%$%%Dale Chihuly.
```

CEIL Function

Returns the smallest integer greater than or equal to a numeric value expression.

Returned data type: DOUBLE

Syntax

CEIL(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DECIMAL, DOUBLE, NUMERIC

Details

If *expression* is null, then the CEILING function returns null. If the result is a number that does not fit into the range of the argument's data type, the CEIL function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

Unlike the CEILZ function, the CEIL function fuzzes the result. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following statements illustrate the CEIL function:

```
proc cas;
   a=ceil(-2.4);
   b=ceil(1+1.e-11);
   c=ceil(-1+1.e-11);
   d=ceil(1+1.e-13);
   print "a=" a "b=" b "c=" c "d=" d;
run;
```

The following line is written to the SAS log.

```
   a=-2 b=2 c=0 d=1
```

CEILZ Function

Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.

Returned data type: DOUBLE

Syntax

CEILZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Comparisons

Unlike the CEIL function, the CEILZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function, you might get unexpected results.

Example

The following statements illustrate the CEILZ function:

```
proc cas;
   a=ceilz(2.1);
   b=ceilz(3);
   c=ceilz(1+1.e-11);
   d=ceilz(223.456);
   e=ceilz(-223.456);
   print "a=" a "b=" b "c=" c "d=" d "e=" e;
run;
```

The following line is written to the SAS log.

```
   a=3 b=3 c=2 d=224 e=-223
```

CHOOSEC Function

Returns a character value that represents the results of choosing from a list of arguments.

Returned data type: VARCHAR

Syntax

CHOOSEC(*index-expression*, *selection-1*<, ...*selection-n*>)

Arguments

index-expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

selection
specifies a character constant, variable, or expression. The value of this argument is returned by the CHOOSEC function.

Data type DOUBLE

Details

The CHOOSEC function uses the value of *index-expression* to select from the arguments that follow. For example, if *index-expression* is 3, CHOOSEC returns the value of *selection-3*. If the first argument is negative, the function counts backward from the list of arguments, and returns that value.

Comparisons

The CHOOSEC function is similar to the CHOOSEN function except that CHOOSEC returns a character value while CHOOSEN returns a numeric value.

Example

The following example shows how CHOOSEC chooses from a series of values:

```
proc cas;
   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');
   items={Fruit, Color, Planet, Sport};
   print "Choices Are=" items;
run;
```

SAS writes the following output to the log:

```
Choices Are={apple,green,Mercury,baseball}
```

CHOOSEN Function

Returns a numeric value that represents the results of choosing from a list of arguments.

Returned data type: DOUBLE

Syntax

CHOOSEN(*index-expression*, *selection-1*<, ...*selection-n*>)

Arguments

index-expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

selection
specifies a numeric constant, variable, or expression. The value of this argument is returned by the CHOOSEN function.

Data type DOUBLE

Details

The CHOOSEN function uses the value of *index-expression* to select from the arguments that follow. For example, if *index-expression* is 3, CHOOSEN returns the value of *selection-3*. If the first argument is negative, the function counts backward from the list of arguments, and returns that value.

Comparisons

The CHOOSEN function is similar to the CHOOSEC function except that CHOOSEC returns a character value while CHOOSEN returns a numeric value.

Example

The following example shows how CHOOSEN chooses from a series of values:

```
proc cas;
   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);
   print "Item Number=" ItemNumber;
   print "Rank=" Rank;
   print "Score=" Score;
   print "Value=" Value;
run;
```

SAS writes the following output to the log:

```
Item Number=679
Rank=4
Score=33
Value=1014
```

CMISS Function

Counts the number of missing arguments.

Returned data type: DOUBLE

Syntax

CMISS(*argument* <, ...*argument,*>)

Arguments

argument
specifies a constant, variable, or expression. *argument* can be either a character value or a numeric value.

Details

A character expression is counted as missing if it evaluates to a string that contains all blanks or has a length of zero, except when you use the CMISS function in macro processing. A numeric expression is counted as missing if it evaluates to a numeric missing value: ., ._, .A, ... , .Z.

When you use the CMISS function in macro processing, use a period (.) to represent both a character and a numeric missing value. If you use a blank or null value for a character argument, SAS returns an error.

Example

The following statements illustrate the CMISS function:

```
proc cas;
   x = CMISS(1,0, ' ' , 2,5,' ');
run;
```

The following line is written to the SAS log.

```
1
```

```
proc cas;
   x = CMISS(1, ' ');
run;
```

The following line is written to the SAS log.

```
0
```

CNONCT Function

Returns the noncentrality parameter from a chi-square distribution.

Returned data type: DOUBLE

Syntax

CNONCT(*x*, *df*, *probability*)

Required Arguments

x

is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

df

is a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

probability

is a probability.

Range $0 < probability < 1$

Data type DOUBLE

Details

The CNONCT function returns the nonnegative noncentrality parameter from a noncentral chi-square distribution whose parameters are *x*, *df*, and *nc*. If *probability* is greater than the probability from the central chi-square distribution with the parameters *x* and *df*, a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root *nc* of the equation

$$P_c(x|\mathrm{df}, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_c(x|\mathrm{df}, \mathrm{nc}) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} P_g\left(\frac{x}{2}\middle|\frac{df}{2} + j\right)$$

The following relationship applies to the preceding equation:

$$P_g(x|a)$$

is the probability from the gamma distribution given by

$$P_g(x|a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} \varepsilon^{-t} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

```
proc cas;
   x=2;
   df=4;
   do nc=1 to 3 by .5;
      probability=probchi(x, df, nc);
      ncc=cnonct(x, df, probability);
   print "probability=" probability;
   print "ncc=" ncc;
   end;
run;
```

SAS writes the following output to the log:

```
probability=0.1861097793
ncc=1
probability=0.1559154065
ncc=1.5
probability=0.1304765495
ncc=2
probability=0.1090741908
ncc=2.5
probability=0.0910916212
ncc=3
```

COALESCE Function

Returns the first non-null or nonmissing value from a list of numeric arguments.

Returned data type: DOUBLE

Syntax

COALESCE(*expression*<, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

COALESCE accepts one or more numeric expressions. The COALESCE function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all expressions are null or missing, then the COALESCE function returns a null or a missing value depending on whether you are in ANSI mode or SAS mode.

Comparisons

The COALESCE function searches numeric expressions, whereas the COALESCEC function searches character expressions.

Example

The following statement illustrates the COALESCE function:

```
proc cas;
   x=csc(0.5);
   print "x=" x;
   y=csc(1);
   print "y=" y;
   z=csc(3.14159/3);
   print "z=" z;
run;
```

The following line is written to the SAS log.

```
x=42
y=.
z=7
```

COALESCEC Function

Returns the first non-null or nonmissing value from a list of character arguments.

Returned data type: STRING, VARCHAR

Syntax

COALESCEC(*expression*<, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

COALESCEC accepts one or more character expressions. The COALESCEC function checks the value of each expression in the order in which they are listed and returns the first non-null or nonmissing value. If only one value is listed, then the COALESCEC function returns the value of that expression. If all the values of all expressions are null or missing, then the COALESCEC function returns a null or missing value depending on whether you are in ANSI mode or SAS mode.

Comparisons

The COALESCEC function searches character expressions, whereas the COALESCE function searches numeric expressions.

Example

The following statements illustrate the COALESCEC function:

```
proc cas;
   x = COALESCEC('', 'Hello');
   y = COALESCEC ('', 'Goodbye', 'Hello');
   print "x=" x;
   print "y=" y;
run;
```

The following line is written to the SAS log.

```
   x=Hello
   y=Goodbye
```

COMB Function

Computes the number of combinations of *n* elements taken *r* at a time.

Returned data type: DOUBLE

Syntax

COMB(*n*, *r*)

Arguments

n

is a nonnegative whole number that represents the total number of elements from which the sample is chosen.

Data type DOUBLE

r

is a nonnegative whole number that represents the number of chosen elements.

Restriction $r \leq n$

Data type DOUBLE

Details

The mathematical representation of the COMB function is given by the following equation:

$$COMB\,(n, r) = \binom{n}{r} = \frac{n!}{r!\,(n-r)!}$$

In the preceding equation, $n \geq 0$, $r \geq 0$, and $n \geq r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the COMB function.

Example

The following statement illustrates the COMB function:

```
proc cas;
   x=comb(5,1);
   print "x=" x;
run;
```

The following line is written to the SAS log.

```
x=5
```

COMPARE Function

Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.

Returned data type: STRING, VARCHAR

Syntax

COMPARE(*string-1*, *string-2*<, *modifiers*>)

Arguments

string-1
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

string-2
specifies a character constant, variable, or expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

modifiers
specifies a character string that can modify the action of the COMPARE function. You can use one or more of the following characters as a valid modifier: I (ignores case), L (removes leading blanks), N (removes quotation marks and ignores case), and : (colon, truncates the longer of the strings to the length of the shorter one, or to one).

specifies a character string that can modify the action of the COMPARE function. You can use one or more of the following characters as a valid modifier:

i or I	ignores the case in *string-1* and *string-2*.
l or L	removes leading blanks in *string-1* and *string-2* before comparing the values.
n or N	removes quotation marks from any argument that is a name literal and ignores the case of *string-1* and *string-2*. A name literal is a name token that is expressed as a string within quotation marks, followed by the uppercase or lowercase letter *n*. Name literals enable you to use special characters (including blanks) that are not otherwise allowed in table or variable names. For COMPARE to

recognize a string as a name literal, the first character must be a quotation mark.

: (colon) truncates the longer of *string-1* or *string-2* to the length of the shorter string, or to one, whichever is greater. If you do not specify this modifier, the shorter string is padded with blanks to the same length as the longer string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip COMPARE ignores blanks that are used as modifiers.

Details

The Basics

The order in which the modifiers appear in the COMPARE function is relevant.

- "LN" first removes leading blanks from each string, and then removes quotation marks from name literals.
- "NL" first removes quotation marks from name literals, and then removes leading blanks from each string.

In the COMPARE function, if *string-1* and *string-2* do not differ, COMPARE returns a value of zero. If the arguments differ, then the following apply:

- The sign of the result is negative if *string-1* precedes *string-2* in a sort sequence, and positive if *string-1* follows *string-2* in a sort sequence.
- The magnitude of the result is equal to the position of the leftmost character at which the strings differ.

DBCS Compatibility

The DBCS equivalent function is KCOMPARE. There are minor differences between the COMPARE and KCOMPARE functions. Both functions accept varying numbers of arguments, but usage of the third argument is not compatible. The following example shows the differences in the syntax:

COMPARE(*string-1*, *string-2*<, *modifiers*>)

KCOMPARE(*string-1*<, *position*<, *count*>>, *string-2*)

Example: Truncating Strings Using the COMPARE Function

The following example uses the : (colon) modifier to truncate strings.

```
proc cas;
   pad1=compare('abc', 'abc           ');
   pad2=compare('abc', 'abcdef        ');
```

```
    truncate1=compare('abc', 'abcdef',':');
    truncate2=compare('abcdef', 'abc',':');
    blank=compare('', 'abc',          ':');
    columns={'pad1', 'pad2', 'truncate1', 'truncate2', 'blank'};
    coltypes={'int64', 'int64', 'int64', 'int64', 'int64'};
    row1={pad1, pad2, truncate1, truncate2, blank};
    mytable=newtable('mytable', columns, coltypes, row1);
    print mytable;
    describe mytable;
run;
```

SAS writes the following output to the log:

```
[mytable] Table ( [1] Rows [5] columns
 Column Names:
 [1] pad1             [pad1            ] (int64)
 [2] pad2             [pad2            ] (int64)
 [3] truncate1        [truncate1       ] (int64)
 [4] truncate2        [truncate2       ] (int64)
 [5] blank            [blank           ] (int64)
```

Output 4.1 *Using the COMPARE Function*

mytable: Results

pad1	pad2	truncate1	truncate2	blank
0	-4	0	0	-1

COMPBL Function

Removes multiple blanks from a character string.

Returned data type: STRING, VARCHAR

Syntax

COMPBL(*character-expression*)

Arguments

character-expression
specifies any valid expression that evaluates or can be coerced to a character string and that specifies the character string to compress.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The COMPBL function removes multiple blanks in a character string by translating each occurrence of two or more consecutive blanks into a single blank.

Comparisons

The COMPRESS function removes every occurrence of the specific character from a string. If you specify a blank as the character to remove from the source string, the COMPRESS function is similar to the COMPBL function. However, the COMPRESS function removes all blanks from the source string. The COMPBL function compresses multiple blanks to a single blank and has no effect on a single blank.

Example

The following statements illustrate the COMPBL function.

```
proc cas;
   a=compbl('January   Status');
   print a;
run;
```

The following line is written to the SAS log.

```
January Status
```

COMPFUZZ Function

Performs a fuzzy comparison of two numeric values.

Returned data type: DOUBLE

Syntax

COMPFUZZ(*expression-1*, *expression-2*<, *fuzz*<, *scale*>>)

Arguments

expression-1
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

expression-2
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

fuzz
is a nonnegative numeric value that specifies the relative threshold for comparisons. Values that are greater than or equal to one are treated as multiples of the machine precision.

Default 1024

Data type DOUBLE

scale
specifies the scale factor.

Default MAX (ABS (*expression-1*), ABS (*expression-2*))

Data type DOUBLE

Details

The COMPFUZZ function returns the following values if you specify all four arguments:

- -1 if *expression-1* < *expression-2* – threshold
- 0 if ABS(*expression-1* - *expression-2*) ≤ threshold
- 1 if *expression-1* > *expression-2* + threshold

The following relationships exist:

- threshold = *fuzz* * ABS(*scale*) if 0 ≤ *fuzz* < 1
- threshold = *fuzz* * ABS(*scale*) * CONSTANT('MACEPS') if 1 ≤ *fuzz* < 1 / CONSTANT('MACEPS')

COMPFUZZ avoids floating-point underflow or overflow.

Comparisons

The COMPFUZZ function compares two floating-point numbers and returns a value based on the comparison. The ROUND function rounds an argument to a value that is very close to a multiple of a second argument. The result might not be an exact multiple of the second argument.

Example

In floating-point arithmetic, the value of a sum sometimes depends on the order in which the numbers are added. One approximate bound for the floating-point error in the computation of a sum of *n* numbers, x1 through x*n*, is expressed by the following formula:

```
n * machine_precision * sum (abs(x1) + ... + abs(xn))
```

To compare sums of *n* floating-point numbers with the COMPFUZZ function, you can therefore use *n* as the fuzz value and the sum of the absolute values as the scale factor, as shown in the following DATA step:

```
proc cas;
   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);
   print "sum1=" sum1;
   print "sum2=" sum2;
   print "diff=" 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);
   print "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);
   print "fuzz and scale (correct):  " compfuzz;
run;
```

The following lines are written to the SAS log:

```
sum1=1.8095238209
 sum2=1.8095238095
 diff=1.1353266E-8
 fuzz only (wrong):         1
 fuzz and scale (correct):  0
```

COMPOUND Function

Returns compound interest parameters.

Categories: CAS

Financial

Returned data type: DOUBLE

Syntax

COMPOUND(*a*, *f*, *r*, *n*)

Arguments

a

specifies the initial amount.

Range *a* ≥ 0

Data type DOUBLE

f

specifies the future amount (at the end of *n* periods).

Range *f* ≥ 0

Data type DOUBLE

r

specifies the periodic interest rate expressed as a fraction.

Range *r* ≥ 0

Data type DOUBLE

n

specifies the number of compounding periods.

Range *n* ≥ 0

Data type DOUBLE

Details

The COMPOUND function returns the missing argument in the list of four arguments from a compound interest calculation. The arguments are related by the following equation:

$$f = a(1 + r)^n$$

One missing argument must be provided. A compound interest parameter is then calculated from the remaining three values. No adjustment is made to convert the results to round numbers.

If *n*=0, then

$$f = a$$

and

$$(1 + r)^n$$

is equal to 1.

Note: If you choose *r* as your missing value, then COMPOUND returns an error.

Example

The accumulated value of an investment of $2000 at a nominal annual interest rate of 9%, compounded monthly after 30 months, can be expressed as follows:

```
proc cas;
   future=compound(2000, ., 0.09/12, 30);
run;
```

The value returned is 2502.544. The second argument has been set to missing, indicating that the future amount is to be calculated. The 9% nominal annual rate has been converted to a monthly rate of 0.09/12. The rate argument is the fractional (not the percentage) interest rate per compounding period.

COMPRESS Function

Returns a character string with specified characters removed from the original string.

Returned data type: STRING, VARCHAR

Syntax

COMPRESS(*character-expression*<, *character-list-expression*>)

Arguments

character-expression
specifies any valid expression that evaluates to a character expression and from which specified characters will be removed.

Requirement: Enclose a literal string of characters in single quotation marks.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

character-list-expression
specifies a variable or any valid expression that initializes a list of characters.

By default, the characters in this list are removed from *character-expression*.

Requirement Enclose a literal string of characters in single quotation marks.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The COMPRESS function allows null arguments. A null argument is treated as a string that has a length of zero.

Based on the number of arguments, the COMPRESS functions works as follows:

Number of Arguments	Results
Only the first argument, *source*	All blanks have been removed. If the argument is completely blank, then the result is a string with a length of zero. If you assign the result to a character variable with a fixed length, then the value of that variable will be padded with blanks to fill its defined length.
Two arguments, *source* and *chars*	All characters that appear in the second argument are removed from the result.

To remove digits and plus or minus signs, you could use the following function call:

```
COMPRESS(source, "1234567890+-");
```

Examples

Example 1: Compressing Blanks

This example shows how to remove blanks from a character string.

```
proc cas;
   a='AB C D ';
   b=compress(a);
   print b;
run;
```

The following line is written to the SAS log.

```
   ABCD
```

Example 2: Compressing Vowels

This example shows how to remove characters from a string.

```
proc cas;
   x='123-4567-8901 e 234-5678-9012 i';
   y=compress(x,'aeiou');
   print y;
run;
```

The following line is written to the SAS log.

```
   123-4567-8901  234-5678-9012
```

CONSTANT Function

Computes machine and mathematical constants.

Returned data type: DOUBLE

Syntax

CONSTANT(*constant*<, *parameter*>)

Arguments

constant

is a character constant, variable, or expression that identifies the constant to be returned. Valid constants are as follows: 'E', 'EULER', 'PI', 'EXACTINT', 'BIG', 'LOGBIG', 'SQRTBIG', 'SMALL', 'LOGSMALL', 'SQRTSMALL', 'MACEPS', 'LOGMACEPS', and 'SQRTMACEPS'.

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
`'PI'`	Pi
`'EXACTINT' [,nbytes]`	Exact integer
`'BIG'`	The largest double-precision number

Constant	Description
'LOGBIG' [,base]	The log with respect to *base* of BIG
'SQRTBIG'	The square root of BIG
'SMALL'	The smallest double-precision number
'LOGSMALL' [,base]	The log with respect to *base* of SMALL
'SQRTSMALL'	The square root of SMALL
'MACEPS'	Machine precision constant
'LOGMACEPS' [,base]	The log with respect to *base* of MACEPS
'SQRTMACEPS'	The square root of MACEPS

parameter
is a numeric parameter that can be used as an optional argument with some of the constants specified in *constant*. When used, *parameter* alters the functionality of the CONSTANT function.

Details

Overview

CAUTION
In some operating environments, the run-time library might have limitations that prevent the use of the full range of floating-point numbers that the hardware provides. In such cases, the CONSTANT function attempts to return values that are compatible with the limitations of the run-time library. For example, if the run-time library cannot compute `EXP(LOG(CONSTANT('BIG')))`, then `CONSTANT('LOGBIG')` will not return the same value as `LOG(CONSTANT('BIG'))`, but will return a value such that `EXP(CONSTANT('LOGBIG'))` can be computed.

The Natural Base

CONSTANT('E')

The natural base is described by the following equation:

$$\lim_{x \to 0} (1+x)^{\frac{1}{x}} \approx 2.718281828459045$$

Euler Constant

CONSTANT('EULER')

Euler's constant is described by the following equation:

$$\lim_{n \to \infty} \left\{ \sum_{j=1}^{j=n} \frac{1}{j} - \log(n) \right\} \approx 0.577215664901532860$$

Pi

CONSTANT('PI')

Pi is the ratio between the circumference and the diameter of a circle. Many expressions exist for computing this constant. One such expression for the series is described by the following equation:

$$4 \sum_{j=0}^{j=\infty} \frac{(-1)^j}{2j+1} \approx 3.14159265358979323846$$

Exact Integer

CONSTANT('EXACTINT'<, *nbytes*>)

Arguments

nbytes
is a numeric value that is the number of bytes.

Default 8

Range 2 ≤nbytes ≤ 8

The exact integer is the largest integer *k* such that all integers less than or equal to *k* in absolute value have an exact representation in a SAS numeric variable of length *nbytes*. This information can be useful to know before you trim a SAS numeric variable from the default 8 bytes of storage to a lower number of bytes to save storage.

The Largest Double-Precision Number

CONSTANT('BIG')

This case returns the largest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of BIG

CONSTANT('LOGBIG'<, *base*>)

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given *base* raised to a power less than or equal to CONSTANT('LOGBIG', *base*) by using the power operation (**) without causing any overflows.

It is safe to exponentiate any floating-point number less than or equal to CONSTANT('LOGBIG') by using the exponential function, EXP, without causing any overflows.

The Square Root of BIG

CONSTANT('SQRTBIG')

This case returns the square root of the largest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to square any floating-point number less than or equal to CONSTANT('SQRTBIG') without causing any overflows.

The Smallest Double-Precision Number

CONSTANT('SMALL')

This case returns the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

The Logarithm of SMALL

CONSTANT('LOGSMALL'<, *base*>)

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of the smallest double-precision floating-point number (8-bytes) that is representable on your computer.

It is safe to exponentiate the given *base* raised to a power greater than or equal to CONSTANT('LOGSMALL', *base*) by using the power operation (**) without causing any underflows or 0.

It is safe to exponentiate any floating-point number greater than or equal to CONSTANT('LOGSMALL') by using the exponential function, EXP, without causing any underflows or 0.

The Square Root of SMALL

CONSTANT('SQRTSMALL')

This case returns the square root of the smallest double-precision floating-point number (8-bytes) that is representable on the computer.

It is safe to square any floating-point number greater than or equal to CONSTANT('SQRTBIG') without causing any underflows or 0.

Machine Precision

CONSTANT('MACEPS')

This case returns the smallest double-precision floating-point number (8-bytes) $\varepsilon = 2^{-j}$ for some integer *j*, such that $1 + \varepsilon > 1$.

This constant is important in finite precision computations.

The Logarithm of MACEPS

CONSTANT('LOGMACEPS'<, *base*>)

Arguments

base
is a numeric value that is the base of the logarithm.

Default the natural base, E

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

This case returns the logarithm with respect to *base* of CONSTANT('MACEPS').

The Square Root of MACEPS

CONSTANT('SQRTMACEPS')

This case returns the square root of CONSTANT('MACEPS').

Example

The following example uses the CONSTANT function to return values for various constants.

```
proc cas;
      a=constant('E');
      b=constant('EULER');
      c=constant('PI');
      d=constant('EXACTINT');
      e=constant('BIG');
      f=constant('LOGBIG');
      g=constant('SQRTBIG');
      h=constant('SMALL');
```

```
        i=constant('LOGSMALL');
        j=constant('SQRTSMALL');
        k=constant('MACEPS');
        l=constant('LOGMACEPS');
        m=constant('SQRTMACEPS');
        print 'a= ' a;
        print 'b= ' b;
        print 'c= ' c;
        print 'd= ' d;
        print 'e= ' e;
        print 'f= ' f;
        print 'g= ' g;
        print 'h= ' h;
        print 'i= ' i;
        print 'j= ' j;
        print 'k= ' k;
        print 'l= ' l;
        print 'm= ' m;
    end;
run;
```

SAS writes the following output to the log:

```
a= 2.7182818285
b= 0.5772156649
c= 3.1415926536
d= 9.0071993E15
e= 1.797693E308
f= 709.78271289
g= 1.340781E154
h= 2.22507E-308
i= -708.4018337
j= 1.49167E-154
k= 2.220446E-16
l= -36.04365339
m= 1.4901161E-8
```

CONVX Function

Returns the convexity for an enumerated cash flow.

Returned data type: DOUBLE

Syntax

CONVX(*y*, *f*, *c(1)*, ..., *c(k)*)

Arguments

y

specifies the effective per-period yield-to-maturity.

Range $0 < y < 1$

Data type DOUBLE

Tip If you express *y* as a fraction, the dividend must be written as a decimal value. In DS2, integer division results in a value of zero. Zero is converted to a DOUBLE and is passed as the first argument to the CONVX function. The CONVX function returns missing when a zero is passed as the first parameter.

f

specifies the frequency of cash flows per period.

Range $f > 0$

Data type DOUBLE

c(1), ... , c(k)

specifies a list of cash flows.

Data type DOUBLE

Details

The CONVX function returns the value from the following equation.

$$C = \sum_{k=1}^{K} \frac{k(k+f)\dfrac{c(k)}{(1+y)^{\frac{k}{f}}}}{P\left((1+y)^2\right)f^2}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example: Using the CONVX Function

```
proc cas;
  c=convx(.1,.33,.44,.55,.49,.50,.22,.4,.8,.01);
  print "c=" c;
run;
```

SAS writes the following output to the log:

```
c=100.04943781
```

CONVXP Function

Returns the convexity for a periodic cash flow stream, such as a bond.

Returned data type: DOUBLE

Syntax

CONVXP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Ranges $A > 0$

$A > 0$

Data type DOUBLE

c

specifies the nominal per-period coupon rate, expressed as a decimal.

Range $0 \le c < 1$

Data type DOUBLE

n

specifies the number of coupons per period.

Range $n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons.

Range $K > 0$

Data type DOUBLE

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Ranges $0 < k_0 \le \frac{1}{n}$

$0 < k_0 \leq 1/n$

Data type DOUBLE

y

specifies the nominal per-period yield-to-maturity.

Range $y > 0$

Data type DOUBLE

Details

The CONVXP function returns the value from the following equation.

$$C = \frac{1}{n^2}\left(\frac{\sum_{k=1}^{K} t_k(t_k+1)\frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)^2}\right)$$

The following relationships apply to the preceding equation:

$$t_k = nk_0 + k - 1$$

$$c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$$

$$c(K) = \left(1+\frac{c}{n}\right)A$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}$$

Example: Computing the Convexity of a Bond

In the following 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.

```
proc cas;
  c=convxp(1000,.01,4,14,.33/2,.08);
  print "c=" c;
run;
```

SAS writes the following output to the log:

```
c=11.729001987
```

COS Function

Returns the cosine in radians.

Returned data type: DOUBLE

Syntax

COS(*expression*)

Arguments

expression
is any valid expression that evaluates to a numeric value.

Data type DOUBLE

Example

The following statements illustrate the COS function:

```
proc cas;
   x=cos(0.5);
   print "x=" x;
   y=cos(0);
   print "y=" y;
   z=cos(3.14159/3);
   print "z=" z;
run;
```

The following line is written to the SAS log.

```
x=0.8775825619
y=1
z=0.500000766
```

COSH Function

Returns the hyperbolic cosine in radians.

Returned data type: DOUBLE

Syntax

COSH(*expression*)

Arguments

expression
is any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The COSH function returns the hyperbolic cosine of the argument, given by the following equation.

$$(e^{argument} + e^{-argument})/2$$

Example

The following statements illustrate the COSH function:

```
proc cas;
   x=cosh(0);
   print "x=" x;
   y=cosh(-5.0);
   print "y=" y;
   z=cosh(0.5);
   print "z=" z;
run;
```

The following line is written to the SAS log.

```
x=1
y=74.209948525
z=1.1276259652
```

COT Function

Returns the cotangent.

Returned data type: DOUBLE

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

The following statements illustrate the COT function:

```
proc cas;
   x=cot(0.5);
   print "x=" x;
   y=cot(1);
   print "y=" y;
   z=cot(3.14159/3);
   print "z=" z;
run;
```

The following line is written to the SAS log.

```
x=1.8304877217
y=0.6420926159
z=0.5773514486
```

Note: If you use `x=cot(0);`, then the COT function returns a missing value, and a note is written to the log that indicates you entered an invalid argument to the function. This is the correct behavior.

COUNT Function

Counts the number of times that a specified substring appears within a character string.

Returned data type: DOUBLE

Syntax

COUNT(*string*, *substring*<, *modifiers*>)

Arguments

string
specifies a character constant, variable, or expression in which substrings are to be counted.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

substring
specifies the character constant, variable, or expression to be counted in *string*.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

modifiers
is a character constant, variable, or expression that specifies one or more modifiers. The following characters, in uppercase or lowercase, can be used as modifiers:

i or I: ignores character case during the count. If this modifier is not specified, COUNT counts only character substrings with the same case as the characters in *substring*.

t or T: trims trailing blanks from *string* and *substring*.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: If *modifiers* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifiers* can also be expressed as a variable or an expression.

Details

The Basics

The COUNT function searches *string*, from left to right, for the number of occurrences of the specified *substring*, and returns that number of occurrences. If the substring is not found in *string*, COUNT returns a value of 0.

CAUTION
If two occurrences of the specified substring overlap in the string, the result is undefined. For example, COUNT('boobooboo', 'booboo') might return either a 1 or a 2.

Comparisons

The COUNT function counts substrings of characters in a character string, whereas the COUNTC function counts individual characters in a character string.

Example

The following statements illustrate the COUNT function:

```
proc cas;
   xyz='This is a thistle? Yes, this is a thistle.';
   howmanythis=count(xyz,'this');
   put howmanythis;
run;
```

The following line is written to the SAS log.

```
3
```

```
proc cas;
   xyz='This is a thistle? Yes, this is a thistle.';
   howmanyis=count(xyz,'is');
   put howmanyis;
run;
```

The following line is written to the SAS log.

```
6
```

```
proc cas;
   howmanythis_i=count('This is a thistle? Yes, this is a thistle.'
      ,'this','i');
   put howmanythis_i;
run;;
```

The following line is written to the SAS log.

```
4
```

```
proc cas;
   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 following line is written to the SAS log.

```
4
```

```
proc cas;
   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 following line is written to the SAS log.

```
6
```

COUNTC Function

Counts the number of characters in a string that appear or do not appear in a list of characters.

Returned data type: DOUBLE

Syntax

COUNTC(*string*, *charlist*<, *modifiers*>)

Arguments

string
specifies a character constant, variable, or expression in which characters are counted.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

charlist
specifies a character constant, variable, or expression that initializes a list of characters. COUNTC counts characters in this list, provided that you do not specify the V modifier in the *modifiers* argument. If you specify the V modifier, then all characters that are not in this list are counted. You can add more characters to the list by using other modifiers.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tips

Enclose a literal string of characters in quotation marks.

If there are no characters in the list after processing the modifiers, COUNTC returns 0.

modifiers

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTC function. Blanks are ignored. The following characters, in uppercase or lowercase, can be used as modifiers: A (adds alphabetic characters to the list), C (adds control characters), and D (adds digits). Additional modifiers are available.

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	scans *string* from right to left, instead of from left to right.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores case.
l or L	adds lowercase letters to the list of characters.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *charlist* and *modifier* arguments only once, at the first call to this instance of COUNTC. If you change the value of *charlist* or *modifier* in subsequent calls, the change might be ignored by COUNTC.
p or P	adds punctuation marks to the list of characters.
s or S	adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).
t or T	trims trailing blanks from *string* and *chars*. If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the COUNTC function with the T modifier.
u or U	adds uppercase letters to the list of characters.
v or V	counts characters that do not appear in the list of characters. If you do not specify this modifier, then COUNTC counts characters that do appear in the list of characters.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

Data type	CHAR, VARCHAR
Tip	If *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks.

Details

The COUNTC function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. If there are no characters in the list of characters to be counted, COUNTC returns zero.

Note: Remember that strings with a CHAR data type are always padded out with blanks to the declared length. Strings with a VARCHAR data type return the length of the actual string instead of the declared length.

Comparisons

The COUNTC function counts individual characters in a character string, whereas the COUNT function counts substrings of characters in a character string.

Example

The following example uses the COUNTC function with and without modifiers to count the number of characters in a string.

```
proc cas;
  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');
  columns={'string', 'a', 'b', 'b_i', 'abc_i', 'abc_iv', 'abc_ivt'};
  coltypes={'string','int64','int64','int64','int64','int64','int64'};
  row1={string, a, b, b_i, abc_i, abc_iv, abc_ivt};
  countctab=newtable('countctab', columns, coltypes, row1);
 print countctab;
run;
```

Output 4.2 *Results from Using the COUNTC Function with and without Modifiers*

countctab: Results

string	a	b	b_i	abc_i	abc_iv	abc_ivt
Baboons Eat Bananas	5	1	3	8	16	11

COUNTW Function

Counts the number of words in a character string.

Returned data type: DOUBLE

Syntax

COUNTW(*string*<, *chars*<, *modifiers*>>)

Arguments

string
specifies a character constant, variable, or expression in which words are counted.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

chars
specifies an optional character constant, variable, or expression that initializes a list of characters. The characters in this list are the delimiters that separate words, provided that you do not use the K modifier in the *modifier* argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to the list by using other modifiers.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

modifiers
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTW function. The following characters, in uppercase or lowercase, can be used as modifiers: A (adds alphabetic characters to the list), C (adds control characters), and D (adds digits). Additional modifiers are available.

blank is ignored.

a or A adds alphabetic characters to the list of characters.

b or B counts from right to left instead of from left to right. Right-to-left counting makes a difference only when you use the Q modifier and the string contains unbalanced quotation marks.

c or C adds control characters to the list of characters.

d or D adds digits to the list of characters.

f or F adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.

g or G adds graphic characters to the list of characters.

h or H adds a horizontal tab to the list of characters.

i or I ignores the case of the characters.

k or K causes all characters that are not in the list of characters to be treated as delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.

l or L adds lowercase letters to the list of characters.

m or M specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.

n or N adds digits, an underscore, and English letters (that is, the characters that can appear after the first character in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O processes the *chars* and *modifier* arguments only once, rather than every time the COUNTW function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make COUNTW run faster when you call it in a loop where *chars* and *modifier* arguments do not change.

p or P adds punctuation marks to the list of characters.

q or Q ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of *string* contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.

s or S adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.

t or T trims trailing blanks from the *string* and *chars* arguments.

u or U adds uppercase letters to the list of characters.

w or W adds printable characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Data type CHAR, VARCHAR

Details

Definition of “Word”

In the COUNTW function, “word” refers to a substring that has one of the following characteristics:

- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters, except if you use the Q modifier and the delimiters are within substrings that have quotation marks

Note: The definition of “word” is the same in both the SCAN function and the COUNTW function.

Delimiter refers to any of several characters that you can specify to separate words.

Using the COUNTW Function in ASCII and EBCDIC Environments

If you use the COUNTW function with only two arguments, the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.

- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

Using Null Arguments

The COUNTW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Using the M Modifier

If you do not use the M modifier, then a word must contain at least one character. If you use the M modifier, then a word can have a length of zero. In this case, the number of words is one plus the number of delimiters in the string, not counting delimiters inside strings that are enclosed in quotation marks when you use the Q modifier.

Example

The following example shows how to use the COUNTW function with the M and P modifiers.

The explanation for the value of **mp** for each string is as follows:

- The period is the delimiter and the m modifier causes the period at the end to refer to a subsequent word with zero length, but never the less, a word. So there is one word before the period and one word after the period for a total of two words.
- No delimiters, so there is only one word.
- The p modifier adds punctuation as a delimiter therefore 3 words.
- The p modifier adds punctuation, so / is a delimiter. The m modifier causes the leading / to refer to a word at beginning with zero length for a total of six words.
- The first \ is an escape character. The second \ is a delimiter, so there are six words.

```
proc cas;
      string1='The quick brown fox jumps over the lazy dog.';
      string2='        Leading blanks';
      string3='2+2=4';
      string4='/unix/path/names/use/slashes';
      string5='\Windows\Path\Names\Use\Backslashes';
   a       = countw(string1, 'a');
   b       = countw(string2, 'b');
   b_i     = countw(string3, 'b', 'i');
   abc_i   = countw(string4, 'abc', 'i');
   abc_j   = countw(string5, 'abc', 'j');
  columns={'a', 'b', 'b_i', 'abc_i', 'abc_j'};
  coltypes={'string','int64','int64','int64','int64','int64','int64'};
  row1={string1, a, b, b_i, abc_i, abc_j};
  row2={string2, a, b, b_i, abc_i, abc_j};
  row3={string3, a, b, b_i, abc_i, abc_j};
  row4={string4, a, b, b_i, abc_i, abc_j};
  row5={string5, a, b, b_i, abc_i, abc_j};
  countwtab=newtable('countwtab', columns, coltypes, row1, row2, row3,
row4, row5);
 print countwtab;
run;
```

Output 4.3 *Results from Using the COUNTW Function*

countwtab: Results

a	b	b_i	abc_i	abc_j
The quick brown fox jumps over the lazy dog.	2	2	1	4
Leading blanks	2	2	1	4
2+2=4	2	2	1	4
/unix/path/names/use/slashes	2	2	1	4
\Windows\Path\Names\Use\Backslashes	2	2	1	4

CSC Function

Returns the cosecant.

Returned data type: DOUBLE

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.

Data type DOUBLE

Comparisons

The CSC function is related to the SIN function in this way:

```
csc(x)=1/sin(x)
```

Example

The following statements illustrate the CSC function:

```
proc cas;
   data one;
   x=csc(0.5);
   put x=;
   y=csc(1);
   put y=;
   z=csc(3.14159/3);
   put z=;
run;
```

The following line is written to the SAS log.

```
x=2.0858296429
y=1.1883951058
z=1.1547011281
```

Note: If you use `x=csc(0);`, then the CSC function returns a missing value, and a note is written to the log that indicates you entered an invalid argument to the function. This is the correct behavior.

CSS Function

Returns the corrected sum of squares.

Returned data type: DOUBLE

Syntax

CSS(*expression*<, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one non-null or nonmissing expression is required.

Data type: DOUBLE

Example

The following statements illustrate the CSS function:

```
proc cas;
   x1=css(5,9,3,6);
```

```
   print "x1=" x1;
   x2=css(5,8,9,6,.);
   print "x2=" x2;
   x3=css(8,9,6,.);
   print "x3=" x3;
run;
```

The following line is written to the SAS log.

```
x1=18.75
x2=10
x3=4.6666666667
```

CUMIPMT Function

Returns the cumulative interest paid on a loan between the start and end period.

Returned data type: DOUBLE

Syntax

CUMIPMT(*rate*, *number-of-periods*, *principal-amount*<, *start-period*><, *end-period*><, *type*>)

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods. *Number-of-periods* must be a positive, whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Data type DOUBLE

start-period
specifies the start period for the calculation.

Data type DOUBLE

end-period
specifies the end period for the calculation.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Data type DOUBLE

Example

- The cumulative interest that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

```
proc cas;
      TotalInterest= CUMIPMT(0.09/12, 360, 125000, 13, 24, 0);
      print 'Total Interest=' TotalInterest;
run;
```

  This computation returns a value of $11,135.23.

- The interest that is paid on the first period of the same loan is computed in the following way:

```
proc cas;
     first_period_interest= CUMIPMT(0.09/12, 360, 125000, 1, 1, 0);
     print 'Total Interest=' first_period_interest;
run;
```

  This computation returns a value of $937.50.

CUMPRINC Function

Returns the cumulative principal paid on a loan between the start and end period.

Returned data type: DOUBLE

Syntax

CUMPRINC(*rate*, *number-of-periods*, *principal-amount*<, *start-period*><, *end-period*><, *type*>)

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing or null value is specified.

start-period
specifies the start period for the calculation.

Data type DOUBLE

end-period
specifies the end period for the calculation.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Data type DOUBLE

Example

- The cumulative principal that is paid during the second year of a $125,000, 30-year loan with end-of-period monthly payments and a nominal annual interest rate of 9%, is computed as follows:

```
proc cas;
      PrincipalYear2=CUMPRINC(0.09/12, 360, 125000, 12, 24, 0);
      print 'Principal Year 2 EOP=' PrincipalYear2;
run;
```

  This computation returns a value of $1008.23.

- The principal that is paid on the second year of the same loan with beginning-of-period payments is computed as follows:

```
proc cas;
      PrincipalYear2b = CUMPRINC(0.09/12, 360, 125000, 12, 24, 1);
      print 'Principal Year 2 BOP=' PrincipalYear2b;
run;
```

This computation returns a value of $1000.73.

CV Function

Returns the coefficient of variation.

Returned data type: DOUBLE

Syntax

CV(*expression-1*, *expression-2* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: DOUBLE

Example

The following statements illustrate the CV function:

```
proc cas;
   x1=cv(5, 9, 3, 6);
   print "x1=" x1;
   x2=cv(5, 8, 9, 6, .);
   print "x2=" x2;
   x3=cv(8, 9, 6, .);
   print "x3=" x3;
run;
```

The following line is written to the SAS log.

```
x1=43.47826087
x2=26.082026548
x3=19.924242152
```

DAIRY Function

Returns the derivative of the AIRY function.

Returned data type: DOUBLE

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.

Example

The following statements illustrate the DAIRY function:

```
proc
cas;


x=dairy(2.0);


y=dairy(-2.0);

   print "x=" x "y="
y;

run;
```

The following line is written to the SAS log.

```
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.

Returned data type: DOUBLE

Syntax

DATDIF(*sdate*, *edate*, *basis*)

Arguments

sdate

specifies a SAS date value that identifies the starting date.

Data type: DATE

Tip: If *sdate* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

edate

specifies a SAS date value that identifies the ending date.

Data type: DATE

Tip: If *edate* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

basis

specifies a character string that represents the day count basis. The following values for *basis* are valid: '30/360', 'ACT/ACT', 'ACT/360', and 'ACT/365'. For example, '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.

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.

Alias '360'

'ACT/ACT'
uses the actual number of days between dates. Each month is considered to have the actual number of calendar days in that month, and each year is considered to have the actual number of calendar days in that year.

Alias *'Actual'*

'ACT/360'
uses the actual number of calendar days in a particular month, and 360 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/360* is used for short-term securities.

'ACT/365'
uses the actual number of calendar days in a particular month, and 365 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/365* is used for short-term securities.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

The DATDIF function has a specific meaning in the securities industry, and the method of calculation is not the same as the actual day count method. Calculations can use months and years that contain the actual number of days. Calculations can also be based on a 30-day month or a 360-day year. For more information about standard securities calculation methods, see the References section at the bottom of this function.

Note: When counting the number of days in a month, DATDIF *always* includes the starting date and excludes the ending date.

Method of Calculation for Day Count Basis (30/360)

To calculate the number of days between two dates, use the following formula:

$$Number of days = [(Y2 - Y1) * 360] + [(M2 - M1) * 30] + (D2 - D1)$$

Arguments

Y2
specifies the year of the later date.

Y1
specifies the year of the earlier date.

M2
specifies the month of the later date.

M1
specifies the month of the earlier date.

D2
specifies the day of the later date.

D1
specifies the day of the earlier date.

Because all months can contain only 30 days, you must adjust for the months that do not contain 30 days. Do this before you calculate the number of days between the two dates.

The following rules apply:

- If the security follows the End-of-Month rule, and D2 is the last day of February (28 days in a non-leap year, 29 days in a leap year), and D1 is the last day of February, then change D2 to 30.
- If the security follows the End-of-Month rule, and D1 is the last day of February, then change D1 to 30.
- If the value of D2 is 31 and the value of D1 is 30 or 31, then change D2 to 30.
- If the value of D1 is 31, then change D1 to 30.

Example

In the following 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.

```
proc cas;
      sdate=19185;
      edate=21185;
      actual=datdif(sdate, edate, 'act/act');
      days360=datdif(sdate, edate, '30/360');
      print  'Actual= ' actual;
      print 'Days 360 = ' days360;
run;
```

The following lines are written to the SAS log.

```
Actual= 2000
Days 360= 1970
```

References

Securities Industry Association 1994. *Standard Securities Calculation Methods - Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, NY: Securities Industry Association.

DATE Function

Returns the current date as a SAS date value.

Alias:	TODAY
Returned data type:	DOUBLE

Syntax

DATE( )

Without Arguments

The DATE function has no arguments.

Comparisons

The DATE function does not take any arguments. The SAS date value returned is the number of days from January 1, 1960 to the current date.

Example

The following statement illustrates the DATE function:

```
proc cas;
   x=date();
   print x;
run;
```

SAS writes the following results to the log:

```
23666
```

DATEJUL Function

Converts a Julian date to a SAS date value.

Returned data type: DOUBLE

Syntax

DATEJUL(*julian-date*)

Arguments

julian-date

specifies any valid expression that evaluates to a numeric value and that represents a Julian date. A Julian date is a date in the form *yyddd* or *yyyyddd*, where *yy* or *yyyy* is a two-digit or four-digit whole number that represents the year and *ddd* is the number of the day of the year. The value of *ddd* must be between 1 and 365 (or 366 for a leap year).

Data type DOUBLE

Details

A SAS date value is the number of days from January 1, 1960 to a specified date. The DATEJUL function returns the number of days from January 1, 1960 to the Julian date specified in *julian-date*.

Example

The following statements illustrate the DATEJUL function:

```
proc cas;
   x=datejul(11365);
   print x;
run;
```

SAS writes these results to the log:

```
18992
```

DATEPART Function

Extracts the date from a SAS datetime value.

Returned data type: DOUBLE

Syntax

DATEPART(*datetime*)

Arguments

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

Details

A SAS datetime value is the number of seconds between January 1, 1960 and the hour, minute, and seconds within a specific date. The DATEPART function determines the date portion of the SAS datetime value and returns the date as a SAS date value, which is the number of days from January 1, 1960.

Example

```
proc cas;
   theTimestamp=inputn('16JUL1969:09:32:00','datetime.');
   theDate=datepart(theTimestamp);
   print(NOTE) "Timestamp=" putn(theTimestamp,'datetime.') " Date="
putn(theDate,'yymmddd10.');
run;
```

The program writes the following output to the log:

```
NOTE: Timestamp=16JUL69:09:32:00 Date=1969-07-16
```

DATETIME Function

Returns the current date and time of day as a SAS datetime value.

Returned data type: DOUBLE

Syntax

DATETIME( )

Comparisons

The DATETIME function does not take any arguments. The SAS datetime value returned is the number of seconds from January 1, 1960 to the current date and time.

Example: Returning the Number of Seconds with the DATETIME Function

The following statement illustrates the DATETIME function:

```
proc cas;
   dt=datetime();
   print dt;
run;
```

SAS writes these results to the log:

```
2044795895.6
```

DAY Function

Returns the day of the month from a SAS date value.

Returned data type: DOUBLE

Syntax

DAY(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

Details

The DAY function produces a whole number from 1 to 31 that represents the day of the month.

A SAS date value is the number of days from January 1, 1960 to a specific date.

Example

```
proc cas;
   theDate=inputn('16JUL1969','DATE9.');
   theDay=day(theDate);
   theMonth=Month(theDate);
   theYear=year(theDate);
   print(NOTE) "Date=" putn(theDate,'date9.');
   print(NOTE) "Day=" theDay ' Month=' theMonth 'Year=' theYear;
run;
```

The program writes the following output to the log:

```
NOTE: Date=16JUL1969
NOTE: Day=16  Month=7 Year=1969
```

DEQUOTE Function

Removes matching single quotation marks from a character string that begins with a single quotation mark, and deletes all characters to the right of the closing quotation mark.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

DEQUOTE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type STRING, VARCHAR

Details

The value that is returned by the DEQUOTE function depends on the first character or the first two characters in *expression*:

- If the first character of *expression* is not a quotation mark, DEQUOTE returns a syntax error.
- If the first character of *expression* is a single quotation mark, the DEQUOTE function removes that single quotation mark from the result. DEQUOTE then scans *expression* from left to right, looking for more single quotation marks or double quotation marks.

  All paired single quotation marks are replaced with a single quotation mark.

  All paired double quotation marks are retained.

  If a double quotation mark is the second character, DEQUOTE removes the double quotation mark from the result. DEQUOTE then scans *expression* from left to right. If a matching double quotation mark is found, the text between the double quotation marks is returned. Any text to the right of the closing double quotation mark, to the end of *expression* is removed from the result.

  The first non-paired single quotation mark in *expression* is the closing single quotation mark and is removed.

  If a close parentheses follows the close single quotation mark, the function returns the dequoted string. If characters exist to the right of the close single quotation mark, the function results in a syntax error and the error is printed in the SAS log.
- If *expression* is enclosed in double quotation marks, the DEQUOTE function returns a null or missing value.

Note: If *expression* is a constant enclosed in quotation marks, those quotation marks are not part of the value of *expression*. Therefore, you do not need to use DEQUOTE to remove the quotation marks that denote a constant.

Example

The following statements illustrate the DEQUOTE function:

Statements	Results
`string=dequote(No quotation marks);`	`ERROR: [HY000]Parse error. Expecting ')' in statement x: char(x) string; string=dequote(no ==>quotation. (0x817ff05c)`
`string=dequote(No 'leading' quotation marks);`	`ERROR: [HY000]Parse error. Expecting ')' in statement x: char(yyy) string; string=dequote(No ==>'leading'. (0x817ff05c)`
`string=dequote('Single matched quotation marks are removed');`	`Single matched quotation marks are removed`
`string=dequote("Matched double quotation marks result in a null or missing value");`	`.`
`string=dequote('Paired ''single'' quotation marks are reduced to a single quotation mark');`	`Paired 'single' quotation marks are reduced to a single quotation mark`
`string=dequote(' "Double quotation marks" within ‟single quotation marks‟, with space before open quotation mark');`	`"Double quotation marks" within ‟single quotation marks‟, with space before open quotation mark'`
`string=dequote('"Double quotation marks within single quotation marks, without space before open quotation mark');`	`Double quotation marks within single quotation marks, without space before open quotation mark'`
`string=dequote('"Text after closing double quotation mark" is removed')`	`Text after closing double quotation mark`
`string=dequote('No matching quotation mark);`	Statement execution does not complete. Submit the following characters to complete the execution: `');`
`string=dequote('Identifiers after close quotation mark' results in a syntax error);`	`ERROR: [HY000]Parse error. Expecting ')' in statement x: string=dequote('Identifiers after close quotation mark' ==>results. (0x817ff05c)`

DEVIANCE Function

Returns the deviance based on a probability distribution.

Returned data type: DOUBLE

Syntax

DEVIANCE(*distribution*, *variable*, *shape-parameter(s)*<, *ε*>)

Arguments

distribution

is a character constant, variable, or expression that identifies the distribution. Valid distributions are 'BERNOULLI', 'BINOMIAL', 'GAMMA', 'IGAUSS', 'NORMAL', and 'POISSON'.

is a character constant, variable, or expression that identifies the distribution. Valid distributions are listed in the following table:

Distribution	Argument
Bernoulli (p. 359)	`'BERNOULLI' | 'BERN'`
Binomial (p. 359)	`'BINOMIAL' | 'BINO'`
Gamma (p. 360)	`'GAMMA'`
Inverse Gauss (Wald) (p. 360)	`'IGAUSS' | 'WALD'`
Normal (p. 361)	`'NORMAL' | 'GAUSSIAN'`
Poisson (p. 361)	`'POISSON' | 'POIS'`

variable

is a numeric constant, variable, or expression.

shape-parameter(s)

are one or more distribution-specific numeric parameters that characterize the shape of the distribution.

ε

is an optional numeric small value used for all of the distributions, except for the normal distribution.

Details

The Bernoulli Distribution

DEVIANCE('BERNOULLI', *variable*, *p*<, *ε*>)

Arguments

variable
is a binary numeric random variable that has the value of 1 for success and 0 for failure.

p
is a numeric probability of success with $\varepsilon \le p \le 1-\varepsilon$.

ε
is an optional positive numeric value that is used to bound *p*. Any value of *p* in the interval $0 \le p \le \varepsilon$ is replaced by *ε*. Any value of *p* in the interval $1 - \varepsilon \le p \le 1$ is replaced by $1 - \varepsilon$.

The DEVIANCE function returns the deviance from a Bernoulli distribution with a probability of success *p*, where success is defined as a random variable value of 1. The equation follows:

$$\text{DEVIANCE}('BERN', variable, p, \varepsilon) = \begin{cases} -2\log(1-p) & x = 0 \\ -2\log(p) & x = 1 \\ . & otherwise \end{cases}$$

The Binomial Distribution

DEVIANCE('BINO', *variable*, *μ*, *n*<, *ε*>)

Arguments

variable
is a numeric random variable that contains the number of successes.

Range $0 \le variable \le 1$

μ
is a numeric mean parameter.

Range $n\varepsilon \le \mu \le n(1-\varepsilon)$

n
is a whole number of Bernoulli trials parameter

Range $n \ge 0$

ε
is an optional positive numeric value that is used to bound *μ*. Any value of *μ* in the interval $0 \le \mu \le n\varepsilon$ is replaced by $n\varepsilon$. Any value of *μ* in the interval $n(1 - \varepsilon) \le \mu \le n$ is replaced by $n(1 - \varepsilon)$.

The DEVIANCE function returns the deviance from a binomial distribution, with a probability of success *p*, and a number of independent Bernoulli trials *n*. The

following equation describes the DEVIANCE function for the Binomial distribution, where *x* is the random variable:

$$\text{DEVIANCE}('BINO', x, \mu, n) = \begin{cases} . & x < 0 \\ 2\left(x\log\left(\frac{x}{\mu}\right) + (n-x)\log\left(\frac{n-x}{n-\mu}\right)\right) & 0 \le x \le n \\ . & x > n \end{cases}$$

The Gamma Distribution

DEVIANCE('GAMMA', *variable*, *μ*<, *ε*>)

Arguments

variable
is a numeric random variable.

Range *variable* ≥ *ε*

μ
is a numeric mean parameter.

Range *μ* ≥*ε*

ε
is an optional positive numeric value that is used to bound *variable* and *μ*. Any value of *variable* in the interval 0 ≤ *variable* ≤ *ε* is replaced by *ε*. Any value of *μ* in the interval 0 ≤ *μ* ≤ *ε* is replaced by *ε*.

The DEVIANCE function returns the deviance from 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:

$$\text{DEVIANCE}('GAMMA', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(-\log\left(\frac{x}{\mu}\right) + \frac{x-\mu}{\mu}\right) & x \ge \varepsilon, \mu \ge \varepsilon \end{cases}$$

The Inverse Gauss (Wald) Distribution

DEVIANCE('IGAUSS' | 'WALD', *variable*, *μ*<, *ε*>)

Arguments

variable
is a numeric random variable.

Range *variable* ≥ *ε*

μ
is a numeric mean parameter.

Range *μ* ≥*ε*

ε
is an optional positive numeric value that is used to bound *variable* and *μ*. Any value of *variable* in the interval 0 ≤ *variable* ≤ *ε* is replaced by *ε*. Any value of *μ* in the interval 0 ≤ *μ* ≤ *ε* is replaced by *ε*.

The DEVIANCE function returns the deviance from an inverse Gaussian distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the inverse Gaussian distribution, where x is the random variable:

$$\text{DEVIANCE}('IGAUSS', x, \mu) = \begin{cases} . & x < 0 \\ \frac{(x-\mu)^2}{\mu^2 x} & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

The Normal Distribution

DEVIANCE('NORMAL' | 'GAUSSIAN', *variable*, μ)

Arguments

variable
is a numeric random variable.

μ
is a numeric mean parameter.

The DEVIANCE function returns the deviance from a normal distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the normal distribution, where x is the random variable:

$$\text{DEVIANCE}('NORMAL', x, \mu) = (x-\mu)^2$$

The Poisson Distribution

DEVIANCE('POISSON', *variable*, μ<, ε>)

Arguments

variable
is a numeric random variable.

Range *variable* ≥ 0

μ
is a numeric mean parameter.

Range $\mu \geq \varepsilon$

ε
is an optional positive numeric value that is used to bound μ. Any value of μ in the interval $0 \leq \mu \leq \varepsilon$ is replaced by ε.

The DEVIANCE function returns the deviance from a Poisson distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the Poisson distribution, where x is the random variable:

$$\text{DEVIANCE}('POISSON', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(x\log\left(\frac{x}{\mu}\right) - (x-\mu)\right) & x \geq 0, \mu \geq \varepsilon \end{cases}$$

Example

```
proc cas;
   print(note) "DEVIANCE('BERNOULLI',1,2)=" DEVIANCE('BERNOULLI',1,2);
   print(note) "DEVIANCE('BINOMIAL',.2,2,2)="
DEVIANCE('BINOMIAL',.2,2,2);
   print(note) "DEVIANCE('IGAUSS',1.0,.2)=" DEVIANCE('IGAUSS',1,.2);
   print(note) "DEVIANCE('NORMAL',1.0,.2)=" DEVIANCE('NORMAL',1,.2);
   print(note) "DEVIANCE('POISSON',1.0,.2)=" DEVIANCE('POISSON',1,.2);
run;
```

The program writes the following output to the log:

```
NOTE: DEVIANCE('BERNOULLI',1,2)=1.999956E-12
NOTE: DEVIANCE('BINOMIAL',.2,2,2)=98.171423763
NOTE: DEVIANCE('IGAUSS',1.0,.2)=16
NOTE: DEVIANCE('NORMAL',1.0,.2)=0.64
NOTE: DEVIANCE('POISSON',1.0,.2)=1.6188758249
```

DHMS Function

Returns a SAS datetime value from date, hour, minute, and second values.

Returned data type: DOUBLE

Syntax

DHMS(*date*, *hour*, *minute*, *second*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

hour
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

minute
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

second
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

Details

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

Example: Using the DHMS Function

```
proc cas;
  day=date();
  time=time();
  sasdt=dhms(day, 0, 0, time);
  format sasdt datetime.;
 print sasdt;
run;
```

The following is printed to the SAS log.

```
07MAY18:11:23:20
```

DIGAMMA Function

Returns the value of the digamma function.

Returned data type: DOUBLE

Syntax

DIGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction Zero and negative integers are not valid.

Data type DOUBLE

Details

The DIGAMMA function returns the ratio that is given by the following equation.

$$\Psi(x) = \Gamma'(x)/\Gamma(x)$$

$\Gamma(.)$ and $\Gamma'(.)$ denote the gamma function and its derivative, respectively. For *expression*>0, the DIGAMMA function is the derivative of the lgamma function.

Example

The following statement illustrates the DIGAMMA function:

```
proc cas;
   x=digamma(1.0);
   print x;
run;
```

SAS writes the following results to the log:

```
-0.577215665
```

DIVIDE Function

Returns the result of a division that handles special missing values for ODS output.

Returned data type: DOUBLE

Syntax

DIVIDE(*x*, *y*)

Arguments

x

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

y

specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

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 4.1 *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.

```
proc cas;
  a=divide(1,0);
  print "a=" a+3 "(infinity)";
  b=divide(2, .I);
  print "b=" b+3;
  c=divide(.I, -1);
  print "c=" c "(minus inifinity)";
  d=divide(constant('big'), constant('small'));
  print "d=" d+3 "(infinity because of overflow)";
run;
```

The following lines are written to the SAS log:

```
a=. (infinity)
b=3
c=M (minus inifinity)
d=. (infinity because of overflow)
```

DURP Function

Returns the modified duration for a periodic cash flow stream, such as a bond.

Returned data type: DOUBLE

Syntax

DURP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Range $A > 0$

Data type DOUBLE

c

specifies the nominal per-period coupon rate, expressed as a fraction.

Range $0 \le c < 1$

Data type DOUBLE

n

specifies the number of coupons per period.

Range $n > 0$ and is a whole number

Data type DOUBLE

K

specifies the number of remaining coupons.

Range $K > 0$ and is a whole number

Data type DOUBLE

k_o

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_o \leq 1/n$

Data type DOUBLE

y

specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Data type DOUBLE

Details

The DURP function returns the value from the following equation.

$$D = \frac{1}{n} \frac{\sum_{k=1}^{K} t_k \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$
- $c(K) = \left(1+\frac{c}{n}\right)A$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}$$

Example: Using the DURP Function

```
proc cas;
  d=durp(1000, 1/100, 4, 14, .33/2, .10);
  print d;
run;
```

SAS writes the following output to the log:

```
d=3.2649588109
```

EFFRATE Function

Returns the effective annual interest rate.

Returned data type: DOUBLE

Syntax

EFFRATE(*compounding-interval*, *rate*)

Arguments

compounding-interval
is a SAS interval. This value represents how often *rate* compounds.

Data type CHAR

rate
is numeric. *Rate* is a nominal annual interest rate (expressed as a percentage) that is compounded at each compounding interval.

Data type DOUBLE

Details

The EFFRATE function returns the effective annual interest rate. The function computes the effective annual interest rate that corresponds to a nominal annual interest rate.

The following details apply to the EFFRATE function:

- The values for rates must be at least –99.
- In considering a nominal interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by EFFRATE equals $e^{rate/100}-1$.

  If *compounding-interval* is not 'CONTINUOUS', and *m* compounding intervals occur in a year, the value that is returned by EFFRATE equals $(1+[rate/100\ m])^{m-1}$.
- The following values are valid for *compounding-interval*:
  - 'CONTINUOUS'
  - 'DAY'
  - 'SEMIMONTH'

- 'MONTH'
- 'QUARTER'
- 'SEMIYEAR'
- 'YEAR'

- If the interval is 'DAY', then *m*=365.

Example

The following 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

```
proc cas;
   effective_rate1=effrate('MONTH',10);
   print "effective_rate1=" effective_rate1;
run;
```

SAS writes the following results to the log:

```
effective_rate1=10.471306744
```

- If a nominal rate is 10%, then the corresponding effective rate when interest is compounded quarterly can be expressed as

```
proc cas;
   effective_rate2 = EFFRATE('QUARTER', 10);
   print "effective_rate2=" effective_rate2;
run;
```

SAS writes the following results to the log:

```
effective_rate2=10.381289062
```

ERF Function

Returns the value of the (normal) error function.

Returned data type: DOUBLE

Syntax

ERF(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The ERF function returns the integral, given by the following:

$$\mathrm{ERF}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \varepsilon^{-z^2} dz$$

You can use the ERF function to find the probability (p) that a normally distributed random variable with mean 0 and standard deviation 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));
```

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));
```

```
proc cas;
   x=erf(1.0);
   y=erf(-1.0);
   print "x=" x;
   print "y=" y;
run;
```

These statements produce these results:

```
x=0.8427007929
y=-0.842700793
```

ERFC Function

Returns the value of the complementary (normal) error function.

Returned data type: DOUBLE

Syntax

ERFC(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The ERFC function returns the complement to the ERF function (that is, 1 – ERF(*argument*)).

Example

The following statements illustrate the ERFC function:

```
proc cas;
   x=erfc(1.0);
   y=erfc(-1.0);
   print "x=" x;
   print "y=" y;
run;
```

These statements produce these results:

```
x=0.1572992071
y=1.8427007929
```

EXP Function

Returns the value of the e constant raised to a specified power.

Returned data type: DOUBLE

Syntax

EXP(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The EXP function raises the constant **e**, which is approximately given by 2.71828, to the power that is supplied by the argument. The result is limited by the maximum value of a double decimal value on the computer.

Example

The following statements illustrate the EXP function:

```
proc cas;
   x=exp(1.0);
   y=exp(0);
   print "x=" x;
   print "y=" y;
run;
```

These statements produce these results:

```
x=2.7182818285
y=1
```

FACT Function

Computes a factorial.

Returned data type: DOUBLE

Syntax

FACT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The mathematical representation of the FACT function is given by the following equation:

$$FACT(n) = n!$$

In this equation, $n \geq 0$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the FACT function.

Example

The following statement illustrates the FACT function:

```
proc
cas;


x=fact(5);

   print "x="
x;

run;
```

SAS writes the following results to the log:

```
x=120
```

FIND Function

Searches for a specific substring of characters within a character string.

Returned data type: STRING

Syntax

FIND(*string*, *substring*<, *modifier(s)*><, *startpos*>)

FIND(*string*, *substring*<, *startpos*><, *modifier(s)*>)

Arguments

string

specifies a character constant, variable, or expression that evaluates or can be coerced to a character string and that will be searched for substrings.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

substring

is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the substring of characters to search for in *string*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

modifier(s)

is a character constant, variable, or expression that specifies one or more modifiers. The following characters, in uppercase or lowercase, can be used as modifiers: I (ignores character case) or T (trims trailing blanks). If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the FIND function with the T modifier.

is a character constant, variable, or expression that specifies one or more modifiers. The following characters, in uppercase or lowercase, can be used as modifiers:

i or I

ignores character case during the search. If this modifier is not specified, FIND searches only for character substrings with the same case as the characters in *substring*.

t or T

trims trailing blanks from *string* and *substring*.

Note: If you want to remove trailing blanks from only one character argument instead of both (or all) character arguments, use the TRIM function instead of the FIND function with the T modifier.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip If *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

startpos

is a numeric constant, variable, or expression which is a whole number that specifies the position at which the search should start and the direction of the search.

Data type DOUBLE

Details

The FIND function searches *string* for the first occurrence of the specified *substring*, and returns the position of that substring. If the substring is not found in *string*, FIND returns a value of 0.

If *startpos* is not specified, FIND starts the search at the beginning of the *string* and searches the *string* from left to right. If *startpos* is specified, the absolute value of *startpos* determines the position at which to start the search. The sign of *startpos* determines the direction of the search.

Value of *startpos*	**Action**
greater than 0	starts the search at position *startpos* and the direction of the search is to the right. If *startpos* is greater than the length of *string*, FIND returns a value of 0.
less than 0	starts the search at position –*startpos* and the direction of the search is to the left. If –*startpos* is greater than the length of *string*, the search starts at the end of *string*.
equal to 0	returns a value of 0.

Comparisons

- The FIND function searches for substrings of characters in a character string, whereas the FINDC function searches for individual characters in a character string.
- The FIND function and the INDEX function both search for substrings of characters in a character string. However, the INDEX function does not have the *modifiers* nor the *startpos* arguments.

Example

The following statements illustrate the FIND function:

```
proc cas;
   whereisshe=find('She sells seashells? Yes, she does.','she ');
   print whereisshe;
run;
```

SAS writes the following results to the log:

```
   27
```

```
proc cas;
   variable1='She sells seashells? Yes, she does.';
   variable2='she ';
   variable3='i';
   whereisshe_i=find(variable1,variable2,variable3);
   print whereisshe_i;
run;
```

SAS writes the following results to the log:

```
   1
```

```
proc cas;
   expression1='She sells seashells? '||'Yes, she does.';
   expression2=kscan('he or she',3)||'  ';
   expression3=trim('t   ');
   whereisshe_t=find(expression1,expression2,expression3);
   print whereisshe_t;
run;
```

SAS writes the following results to the log:

```
   14
```

```
proc cas;
   xyz='She sells seashells? Yes, she does.';
   startposvar=22;
   whereisshe_22=find(xyz,'she',startposvar);
   print whereisshe_22;
run;
```

SAS writes the following results to the log:

```
   27
```

```
proc cas;
   xyz='She sells seashells? Yes, she does.';
   startposexp=1-23;
   whereisShe_ineg22=find(xyz,'She','i',startposexp);
   print whereisShe_ineg22;
run;
```

SAS writes the following results to the log:

```
   14
```

FINDC Function

Searches a string for any character in a list of characters.

Returned data type: DOUBLE

Syntax

FINDC(*string*<, *charlist*>)

FINDC(*string*, *charlist*<, *modifier*>)

FINDC(*string*, *charlist*, *modifier*<, *startpos*>)

FINDC(*string*, *charlist*<, *startpos*><, *modifier*>)

Arguments

string

is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the character string to be searched.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

charlist

is a constant, variable, or character expression that initializes a list of characters. FINDC searches for the characters in this list provided that you do not specify the K modifier in the *modifier* argument. If you specify the K modifier, FINDC searches for all characters that are not in this list of characters. You can add more characters to the list by using other modifiers.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

modifier

is a character constant, variable, or expression in which each character modifies the action of the FINDC function. The following characters, in uppercase or lowercase, can be used as modifiers: A (adds alphabetic characters to the list), C (adds control characters), and D (adds digits). Additional modifiers are available.

is a character constant, variable, or expression in which each character modifies the action of the FINDC function. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	searches from right to left, instead of from left to right, regardless of the sign of the *startpos* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.

h or H adds a horizontal tab to the list of characters.

i or I ignores character case during the search.

k or K searches for any character that does not appear in the list of characters. If you do not specify this modifier, then FINDC searches for any character that appears in the list of characters. The V and K modifiers perform the same function.

l or L adds lowercase letters to the list of characters.

n or N adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O processes the *charlist* and the *modifier* arguments only once, rather than every time the FINDC function is called. Using the O modifier in DS2 (excluding WHERE clauses) can make FINDC run faster when you call it in a loop where the *charlist* and the *modifier* arguments do not change.

p or P adds punctuation marks to the list of characters.

s or S adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T trims trailing blanks from the *string* and *charlist* arguments. Note that if you want to remove trailing blanks from just one character argument instead of both (or all) character arguments, use the TRIM function instead of the FINDC function with the T modifier.

u or U adds uppercase letters to the list of characters.

v or V causes all character that are not in the list of characters to be treated as delimiters. If V is not specified, then all characters that are in the list of characters are treated as delimiters. The V and K modifiers perform the same function.

w or W adds printable characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip If *modifier* is a constant, then enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

startpos

is an optional numeric constant, variable, or expression which is a whole number that specifies the position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The FINDC function searches *string* for the first occurrence of the specified characters, and returns the position of the first character found. If no characters are found in *string*, then FINDC returns a value of 0.

The FINDC function allows character arguments to be null. Null arguments are treated as character strings that have a length of zero. Numeric arguments cannot be null.

If *startpos* is not specified, FINDC begins the search at the end of the string if you use the B modifier, or at the beginning of the string if you do not use the B modifier.

If *startpos* is specified, the absolute value of *startpos* specifies the position at which to begin the search. If you use the B modifier, the search always proceeds from right to left. If you do not use the B modifier, the sign of *startpos* specifies the direction in which to search. The following table summarizes the search directions:

Value of *startpos*	**Action**
greater than 0	search begins at position *startpos* and proceeds to the right. If *startpos* is greater than the length of the string, FINDC returns a value of 0.
less than 0	search begins at position –*startpos* and proceeds to the left. If *startpos* is less than the negative of the length of the string, the search begins at the end of the string.
equal to 0	returns a value of 0.

Comparisons

- The FINDC function searches for individual characters in a character string, whereas the FIND function searches for substrings of characters in a character string.
- The FINDC function and the INDEXC function both search for individual characters in a character string. However, the INDEXC function does not have the *modifier* nor the *startpos* arguments.
- The FINDC function searches for individual characters in a character string, whereas the VERIFY function searches for the first character that is unique to an expression. The VERIFY function does not have the *modifier* nor the *startpos* arguments.

Example: Searching for Characters in a String

This example searches a character string and returns the characters that are found.

```
proc cas;
      j=1;
      do until(j=0);
         j = findc('Hi, ho!','hi',j+1);
         if j= 0 then print 'The End';
         else do;
            c = substr('Hi, ho!', j, 1);
            print "j=" j;
            print "c=" c;
         end;
      end;
run;
```

SAS writes the following output to the log:

```
j=2 c=i
j=5 c=h
The End
```

FINDW Function

Returns the character position of a word in a string, or returns the number of the word in a string.

Returned data type: DOUBLE

Syntax

FINDW(*string*, *word*<, *chars*>)

FINDW(*string*, *word*, *chars*, *modifier(s)*<, *startpos*>)

FINDW(*string*, *word*, *chars*, *startpos*<, *modifier(s)*>)

FINDW(*string*, *word*, *startpos*<, *chars*<, *modifier(s)*>>)

Arguments

string
is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the character string to be searched.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: Enclose a literal string of characters in quotation marks.

word
is a character constant, variable, or expression that evaluates or can be coerced to a character string and that specifies the word to be searched.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

chars

is an optional character constant, variable, or expression that initializes a list of characters. The characters in this list are the delimiters that separate words, provided that you do not specify the K modifier in the *modifier* argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to this list by using other modifiers.

is an optional character constant, variable, or expression that initializes a list of characters.

The characters in this list are the delimiters that separate words, provided that you do not specify the K modifier in the *modifier* argument. If you specify the K modifier, then all characters that are not in this list are delimiters. You can add more characters to this list by using other modifiers.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in quotation marks.

startpos

is an optional numeric constant, variable, or expression which is a whole number that specifies the position at which the search should begin and the direction in which to search.

Data type DOUBLE

'modifier(s)'

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the FINDW function. The following characters, in uppercase or lowercase, can be used as modifiers: A (adds alphabetic characters to the list), C (adds control characters), and D (adds digits). Additional modifiers are available.

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the FINDW function.

You can use the following characters as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	searches from right to left, instead of from left to right, regardless of the sign of the *startpos* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
e or E	counts the words that are scanned until the specified word is found, instead of determining the character position of the specified word in the string. Fragments of words are not counted.

f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters.
k or K	causes all character that are not in the list of characters to be treated as delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters. The K and V modifiers perform the same function.
l or L	adds lowercase letters to the list of characters.
m or M	specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero.
n or N	adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.
o or O	processes the *chars* and the *modifier* arguments only once, rather than every time the FINDW function is called. Using the O modifier in DS2 (excluding WHERE clauses) can make FINDW run faster when you call it in a loop where the *chars* and the *modifier* arguments do not change.
p or P	adds punctuation marks to the list of characters.
q or Q	ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.
r or R	removes leading and trailing delimiters from the *word* argument.
s or S	adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.
t or T	trims trailing blanks from the *string*, *word*, and *chars* arguments.
u or U	adds uppercase letters to the list of characters.
v or V	causes all character that are not in the list of characters to be treated as delimiters. If V is not specified, then all characters that are in the list of characters are treated as delimiters. The V and K modifiers perform the same function.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.
Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
Tip	If you use the *modifier* argument, then it must be positioned after the *chars* argument.

Details

Definition of "Delimiter"

"Delimiter" refers to any of several characters that are used to separate words. You can specify the delimiters by using the *chars* argument, the *modifier* argument, or both. If you specify the Q modifier, then the characters inside substrings that are enclosed in quotation marks are not treated as delimiters.

Definition of "Word"

"Word" refers to a substring that has both of the following characteristics:

- bounded on the left by a delimiter or the beginning of the string
- bounded on the right by a delimiter or the end of the string

Note: A word can contain delimiters. In this case, the FINDW function differs from the SCAN function, in which words are defined as not containing delimiters.

Searching for a String

If the FINDW function fails to find a substring that both matches the specified word and satisfies the definition of a word, then FINDW returns a value of 0.

If the FINDW function finds a substring that both matches the specified word and satisfies the definition of a word, the value that is returned by FINDW depends on whether the E modifier is specified:

- If you specify the E modifier, then FINDW returns the number of complete words that were scanned while searching for the specified word. If *startpos* specifies a position in the middle of a word, then that word is not counted.
- If you do not specify the E modifier, then FINDW returns the character position of the substring that is found.

If you specify the *startpos* argument, then the absolute value of *startpos* specifies the position at which to begin the search. The sign of *startpos* specifies the direction in which to search:

Value of *startpos*	**Action**
greater than 0	search begins at position *startpos* and proceeds to the right. If *startpos* is greater than the length of the string, then FINDW returns a value of 0.
less than 0	search begins at position –*startpos* and proceeds to the left. If *startpos* is less than the negative of the length of the string, then the search begins at the end of the string.
equal to 0	FINDW returns a value of 0.

If you do not specify the *startpos* argument or the B modifier, then FINDW searches from left to right starting at the beginning of the string. If you specify the B modifier, but do not use the *startpos* argument, then FINDW searches from right to left starting at the end of the string.

Using the FINDW Function in ASCII and EBCDIC Environments

If you use the FINDW function with only two arguments, the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the FINDW function uses the ~ character instead.

- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

Using Null Arguments

The FINDW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Example: Searching a Character String and Using the Chars and Startpos 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 *chars* argument specifies a space as the delimiter.

```
proc cas;
      result = findw('At least 2.5 meters of rain falls in a rain
forest.',
         'rain', ' ', 25);
      print "result=" result;
   end;
run;
```

SAS writes the following output to the log:

```
result=40
```

FLOOR Function

Returns the largest integer less than or equal to a numeric value expression.

Returned data type: DOUBLE

Syntax

FLOOR(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DECIMAL, DOUBLE, NUMERIC

Details

If *expression* is within 1E-12 of an integer, the function returns that integer. If the result is a number that does not fit into the range of a DOUBLE, the FLOOR function fails.

If the argument is DECIMAL, the result is DECIMAL. Otherwise, the argument is converted to DOUBLE (if not so already), and the result is DOUBLE.

Comparisons

The FLOOR function fuzzes the results so that if the results are within 1E-12 of an integer, the FLOOR function returns that integer. The FLOORZ function uses zero fuzzing. Therefore, with the FLOORZ function, you might get unexpected results.

Example

The following statement illustrates the FLOOR function:

```
proc cas;
   a=floor(1.95);
   print a;
run;
```

SAS writes the following output to the log:

```
1
```

FMTINFO Function

Returns information about a SAS format or informat.

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.

Returned data type: STRING

Syntax

FMTINFO('*format-name*', '*information-type*');

Arguments

'format-name'
specifies the name of a SAS format or informat.

Requirement *format-name* must be enclosed in single quotation marks.

information-type
specifies the type of information that is returned. The *format-information* can be one of the following values: 'DESC' (short description), 'MINW' (minimum width), and 'MAXW' (maximum width). Additional information types are available.

specifies the type of information that is returned. *format-information* can be one of the following values:

'CAT'
returns the function category.

'TYPE'
returns whether the *format-name* is a format, an informat, or both.

'DESC'
returns a short description of the format or informat.

'MIND'
returns the minimum number of digits to the right of the decimal place in the format or informat.

'MAXD'
returns the maximum number of digits to the right of the decimal place in the format or informat.

'DEFD'
returns the default number of digits to the right of the decimal place in the format or informat.

'MINW'
returns the minimum width value of the format or informat.

'MAXW'
returns the maximum width value of the format or informat.

'DEFW'
returns the default width value of the format or informat.

Restriction You can specify only one *information-type* argument.

Requirement *information-type* must be enclosed in single quotation marks.

Details

The FMTINFO function returns information about a format or informat. You can return information about a format or informat's category, the type of language element, a description of the language element, and the minimum, maximum, and default decimal and width values.

You cannot specify multiple arguments with the FMTINFO function.

The FMTINFO function returns a character string for all data values, including the numeric value arguments MIND, MAXD, DEFD, MINW, MAXW, and DEFW.

Example

The following example returns a character string for the variable *a*.

```
proc cas;
  a=fmtinfo('best', 'cat');
  print a;
run;
```

The following lines are written to the SAS log.

```
num
```

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Returned data type: DOUBLE

Syntax

FLOORZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Comparisons

Unlike the FLOOR function, the FLOORZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function, you might get unexpected results.

Example

The following statements illustrate the FLOORZ function:

```
proc cas;
   var1=2.1;
   a=floorz(var1);
   print a;
   b=floorz(-2.4);
   print b;
   c=floorz(-1.6);
   print c;
run;
```

These statements produce these results:

```
2
-3
-2
```

FNONCT Function

Returns the value of the noncentrality parameter of an F distribution.

Returned data type: DOUBLE

Syntax

FNONCT(*x*, *ndf*, *ddf*, *probability*)

Required Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

ndf
is a numeric numerator degree of freedom parameter.

Range $ndf > 0$

Data type DOUBLE

ddf
is a numeric denominator degree of freedom parameter.

Range $ddf > 0$

Data type DOUBLE

probability
is a probability.

Range $0 < probability < 1$

Data type DOUBLE

Details

The FNONCT function returns the nonnegative noncentrality parameter from a noncentral *F* distribution whose parameters are *x*, *ndf*, *ddf*, and *nc*. If *probability* is greater than the probability from the central *F* distribution whose parameters are *x*, *ndf*, and *ddf*, a root to this problem does not exist. In this case a missing value is returned. A Newton-type algorithm is used to find a nonnegative root *nc* of the equation

$$P_f(x|ndf, ddf, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_f(x|ndf, ddf, nc) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} I_{\frac{(ndf)x}{ddf + (ndf)x}}\left(\frac{ddf}{2} + j, \frac{ddf}{2}\right)$$

In the equation, *I* (. . .) is the probability from the beta distribution that is given by the following equation:

$$I_x(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

```
proc cas;
      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);
      end;
   end;
     columns={'x', 'df', 'ddf', 'nc', 'prob', 'ncc'};
     coltypes={'int64' 'int64','int64', 'int64'};
     row1={x, df, ddf, nc, prob, ncc};
     row2={x, df, ddf, nc, prob, ncc};
     row3={x, df, ddf, nc, prob, ncc};
     row4={x, df, ddf, nc, prob, ncc};
     row5={x, df, ddf, nc, prob, ncc};
     mytable=newtable('mytable',columns, coltypes, row1, row2, row3,
row4, row5);
     print mytable;
run;
```

Output 4.4 *Output from the FNONCT Function*

mytable: Results

x	df	ddf	nc	prob	ncc
2	4	5	3	0	3
2	4	5	3	0	3
2	4	5	3	0	3
2	4	5	3	0	3
2	4	5	3	0	3

FUZZ Function

Returns the nearest whole number if the argument is within 1E-12 of that number.

Categories: CAS
Truncation

Returned data type: DOUBLE

Syntax

FUZZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The FUZZ function returns the nearest whole number if the expression is within 1E-12 of the number (that is, if the absolute difference between the whole number and argument is less than 1E-12). Otherwise, the expression is returned.

Example

The following statements illustrates the FUZZ function:

```
proc cas;
   x1=fuzz(4.99999999999);
   x2=fuzz(5.9999999999);
   x3=fuzz(5.99999999995);
   print "X1=" X1 "X2=" X2 "X3=" X3;
run;
```

The program writes the following output to the log:

```
X1=5 X2=5.9999999999 X3=6
```

GAMINV Function

Returns a quantile from the gamma distribution.

Returned data type: DOUBLE

Syntax

GAMINV(*p*, *a*)

Arguments

p

specifies any valid expression that evaluates to a numeric probability.

Range $0 \leq p < 1$

Data type DOUBLE

a

specifies any valid expression that evaluates to a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

Details

The GAMINV function returns the p^{th} quantile from the gamma distribution, with shape parameter *a*. The probability that a row from a gamma distribution is less than or equal to the returned quantile is *p*.

Note: GAMINV is the inverse of the PROBGAM function.

Example

The following statements illustrate the GAMINV function:

```
proc
cas;

   q1=gaminv(0.5,
9);

   q2=gaminv(0.1,
2.1);

   print "q1=" q1 "q2="
q2;

run;
```

These statements produce these results:

```
q1=8.6689511844 q2=0.5841932369
```

GAMMA Function

Returns the value of the gamma function.

Returned data type: DOUBLE

Syntax

GAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction: Nonpositive integers are invalid.

Data type: DOUBLE

Details

The GAMMA function returns the integral, which is given by the following equation.

$$\mathrm{GAMMA}(x) = \int_0^\infty t^{x-1} e^{-t}\, dt.$$

For positive integers, GAMMA(x) is (x − 1)!. This function is commonly denoted by $\Gamma(x)$.

Example

The following statement illustrates the GAMMA function:

```
proc
cas;

   x=gamma(6);
   print
x;

run;
```

These statements produce this result:

```
120
```

GARKHCLPRC Function

Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.

Categories: CAS
Financial

Returned data type: DOUBLE

Syntax

GARKHCLPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t
is a nonmissing value that specifies the time to maturity.

Data type DOUBLE

S
is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

R_d
is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period *t*.

Requirement Specify a value for R_d for the same time period as the unit of *t*.

Data type DOUBLE

R_f
is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period *t*.

Requirement Specify a value for R_f for the same time period as the unit of *t*.

Data type DOUBLE

sigma
is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

Data type DOUBLE

Details

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

$$\text{CALL} = SN(d_1)\left(\varepsilon^{-R_f t}\right) - EN(d_2)\left(\varepsilon^{-R_d t}\right)$$

Arguments

S
specifies the spot currency price.

N
specifies the cumulative normal density function.

E
specifies the exercise price of the option.

t
specifies the time to expiration.

R_d
specifies the risk-free domestic interest rate for period *t*.

R_f
specifies the risk-free foreign interest rate for period *t*.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(R_d - R_f + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of t=0, the following equation is true:

$$\text{CALL} = \max((S - E), 0)$$

Comparisons

The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

Example

The following statements illustrate the GARKHCLPRC function:

```
proc cas;
   a=garkhclprc(40, .5,
38, .06, .04, .2);



   b=garkhclprc(19, .25,
20, .05, .03, .09);

   print "a=" a;
   print "b="
b;

run;
```

These statements produce these results:

```
a=1.449425106
b=1.1304209448
```

GARKHPTPRC Function

Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

Returned data type: DOUBLE

Syntax

GARKHPTPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

Data type DOUBLE

t

is a nonmissing value that specifies the time to maturity, in years.

Data type DOUBLE

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

Data type DOUBLE

R_d

is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period *t*.

Requirement Specify a value for R_d for the same time period as the unit of *t*.

Data type DOUBLE

R_f

is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period *t*.

Requirement Specify a value for R_t for the same time period as the unit of *t*.

Data type DOUBLE

sigma

is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Data type DOUBLE

Details

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The function is based on the following relationship:

$$\text{PUT} = \text{CALL} - S\left(\varepsilon^{-R_f t}\right) + E\left(\varepsilon^{-R_d t}\right)$$

Arguments

S
specifies the spot currency price.

E
specifies the exercise price of the option.

t
specifies the time to expiration, in years.

R_d
specifies the risk-free domestic interest rate for period *t*.

R_f
specifies the risk-free foreign interest rate for period *t*.

$$d_1 = \frac{\left(\ln\left(\frac{S}{E}\right) + \left(R_d - R_f + \frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

The following arguments apply to the preceding equation:

σ
specifies the volatility of the underlying asset.

σ^2
specifies the variance of the rate of return.

For the special case of *t*=0, the following equation is true:

$$\text{PUT} = \max((E - S), 0)$$

Comparisons

The GARKHPTPRC function calculates the put prices for European options on stocks, based on the Garman-Kohlhagen model. The GARKHCLPRC function calculates the call prices for European options on stocks, based on the Garman-Kohlhagen model. These functions return a scalar value.

Example

The following statements illustrate the GARKHPTPRC function:

```
proc
cas;

   a=garkhptprc(50, .7,
55, .05, .04, .2);

   b=garkhptprc(32, .3,
33, .05, .03, .3);

   print "a=" a "b="
b;

run;
```

These statements produce these results:

```
a=1.4050880945 b=1.5647320514
```

GCD Function

Returns the greatest common divisor for a set of integers.

Returned data type: DOUBLE

Syntax

GCD(*expression-1*, *expression-2* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: DOUBLE

Details

The GCD (greatest common divisor) function returns the greatest common divisor of one or more integers. For example, the greatest common divisor for 30 and 42 is 6. The greatest common divisor is also called the highest common factor.

Example

The following statements illustrate the GCD function:

```
proc cas;
   x=gcd(10, 15);
   print x;
run;
```

SAS writes the following output to the log:

```
5
```

GEODIST Function

Returns the geodetic distance between two latitude and longitude coordinates.

Returned data type: DOUBLE

Syntax

GEODIST(*latitude-1*, *longitude-1*, *latitude-2*, *longitude-2* <,*options*>)

Arguments

latitude
is a numeric constant, variable, or expression that specifies the coordinate of a given position north or south of the equator. Coordinates that are located north of the equator have positive values; coordinates that are located south of the equator have negative values.

Restriction: If the value is expressed in degrees, it must be between 90 and –90. If the value is expressed in radians, it must be between pi/2 and –pi/2.

Data type: DOUBLE

longitude
is a numeric constant, variable, or expression that specifies the coordinate of a given position east or west of the prime meridian, which runs through Greenwich, England. Coordinates that are located east of the prime meridian have positive values; coordinates that are located west of the prime meridian have negative values.

Restriction: If the value is expressed in degrees, it must be between 180 and –180. If the value is expressed in radians, it must be between pi and –pi.

Data type DOUBLE

options
specifies a character constant, variable, or expression that contains any of the following characters:

M specifies distance in miles.

K specifies distance in kilometers. K is the default value for distance.

D specifies that input values are expressed in degrees. D is the default for input values.

R specifies that input values are expressed in radians.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The GEODIST function computes the geodetic distance between any two arbitrary latitude and longitude coordinates. Input values can be expressed in degrees or in radians.

Examples

Example 1: Calculating the Geodetic Distance in Kilometers

The following 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.

```
proc cas;
      distance=geodist(30.68, -88.25, 35.43, -82.55);
      print '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).

```
proc cas;
      distance=geodist(30.68, -88.25, 35.43, -82.55, 'M');
      print 'Distance= ' distance 'miles';
run;
```

SAS writes the following output to the log:

```
Distance = 465.29081088 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.

Returned data type: DOUBLE

Syntax

GEOMEAN(*expression* <, ...*expression*>)

Arguments

expression
is any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the n^{th} root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1)+\log(x_2)+\ldots+\log(x_n))}{n}\right)$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, GEOMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the GEOMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEAN function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEAN function:

```
proc
cas;


x1=geomean(1,2,2,4);


x2=geomean(.,2,4,8);

   print "x1=" x1 "x2="
x2;

run;
```

These statements produce these results:

```
x1=2 x2=4
```

GEOMEANZ Function

Returns the geometric mean, using zero fuzzing.

Returned data type:	DOUBLE

Syntax

GEOMEANZ(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If any argument is zero, then the geometric mean is zero. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the geometric mean of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the n^{th} root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEANZ function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEANZ function:

```
proc
cas;
```

```
x1=geomeanz(1,2,2,4);


x2=geomeanz(.,2,4,8);

   print "x1=" x1 "x2="
x2;

run;
```

These statements produce these results:

```
x1=2 x2=4
```

HARMEAN Function

Returns the harmonic mean.

Returned data type: DOUBLE

Syntax

HARMEAN(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

Details

If any argument is negative, then the result is a null or missing value. A message appears in the log that the negative argument is invalid. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1}+\frac{1}{x_2}+\ldots+\frac{1}{x_n}}$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, HARMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the HARMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEAN function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEANZ, HARMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the HARMEAN function:

```
proc
cas;


x1=harmean(1,2,4,4);


x2=harmean(.,4,12,24);

   print "x1=" x1 "x2="
x2;

run;
```

These statements produce these results:

```
x1=2 x2=8
```

HARMEANZ Function

Returns the harmonic mean, using zero fuzzing.

Returned data type: DOUBLE

Syntax

HARMEANZ(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

Details

If any argument is negative, then the result is a null or value. A message appears in the log that the negative argument is invalid. If all the arguments are null or values, then the result is a null or value. Otherwise, the result is the harmonic mean of the non-null or nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is shown in this equation.

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the GEOMEAN function returns the geometric mean, whereas the HARMEANZ function returns the harmonic mean of the non-null or nonmissing values. Unlike HARMEAN, HARMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the HARMEANZ function:

```
proc
cas;


x1=harmeanz(1,2,4,4);
```

```
x2=harmeanz(.,4,12,24);

   print "x1=" x1 "x2="
x2;

run;
```

These statements produce these results:

```
x1=2 x2=8
```

HMS Function

Returns a SAS time value from hour, minute, and second values.

Returned data type: DOUBLE

Syntax

HMS(*hour*, *minute*, *second*)

Arguments

hour
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

minute
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

second
specifies a numeric expression that represents a whole number from 1 through 59.

Data type DOUBLE

Details

The HMS function returns a numeric value that represents a SAS time value. A SAS time value is a number that represents the number of seconds since midnight of the current day.

Example

The following statements illustrate the HMS function:

```
proc
cas;

   a=hms(12,45,10);
   b=put(a,
time.);

   print "a=" a "b="
b;

run;
```

These statements produce these results:

```
a=45910 b=12:45:10
```

HOLIDAY Function

Returns a SAS date value of a specified holiday for a specified year.

Returned data type: DOUBLE

Syntax

HOLIDAY('*holiday*', *year*)

Arguments

'holiday'
is a character constant, variable, or expression that specifies one of the following values: BOXING, CANADA, CANADAOBSERVED, CHRISTMAS, COLUMBUS, EASTER, FATHERS, HALLOWEEN, LABOR, MLK, MEMORIAL, MOTHERS, NEWYEAR, THANKSGIVING, THANKSGIVINGCANADA, USINDEPENDENCE, USPRESIDENTS, VALENTINES, VETERANS, VETERANSUSG, VETERANSUSPS, and VICTORIA. Values for *holiday* can be in

uppercase or lowercase. For additional information, see *SAS DS2 Language Reference*.

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 4.2 *Holiday Values and Their Descriptions*

Holiday Value	Description	Date Celebrated
BOXING	Boxing Day	December 26
CANADA	Canadian Independence Day	July 1
CANADAOBSERVED	Canadian Independence Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25
COLUMBUS	Columbus Day	2nd Monday in October
EASTER	Easter Sunday	date varies
FATHERS	Father's Day	3rd Sunday in June
HALLOWEEN	Halloween	October 31
LABOR	Labor Day	1st Monday in September
MLK	Martin Luther King, Jr. 's birthday	3rd Monday in January beginning in 1986
MEMORIAL	Memorial Day	last Monday in May (since 1971)
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

Data type CHAR

year
is a numeric constant, variable, or expression that specifies a four-digit year. If you use a two-digit year, then you must specify the YEARCUTOFF= system option.

Data type DOUBLE

Details

The HOLIDAY function computes the date on which a specific holiday occurs in a specified year. Only certain common U.S. and Canadian holidays are defined for use with this function.

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.

Example

The following statements illustrate the HOLIDAY function:

```
proc
cas;

   thanks = holiday('thanksgiving',
2021);

   format thanks
date9.;

   print "thanks="
thanks;



   boxing = holiday('boxing',
2021);

   format boxing
date9.;

   print "boxing="
boxing;



   easter = holiday('easter',
2021);

   format easter
date9.;

   print "easter="
easter;



   canada = holiday('canada',
2021);

   format canada
date9.;

   print "canada="
canada;
```

```
   fathers = holiday('fathers',
2021);

   format fathers
date9.;

   print "fathers="
fathers;



   valentines = holiday('valentines',
2021);

   format valentines
date9.;

   print "valentines="
valentines;



   victoria = holiday('victoria',
2021);

   format victoria
date9.;

   print "victoria="
victoria;

run;
```

SAS writes these results to the log:

```
thanks=25NOV2021
boxing=26DEC2021
easter=04APR2021
canada=01JUL2021
fathers=20JUN2021
valentines=14FEB2021
victoria=24MAY2021
```

HOUR Function

Returns the hour from a SAS time or datetime value.

Returned data type: DOUBLE

Syntax

HOUR(*time* | *datetime*)

Arguments

time
specifies any valid expression that represents a SAS time value.

Data type DOUBLE

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

Details

The HOUR function returns a numeric value that represents the hour from a SAS time or datetime value. Numeric values can range from 0 through 23. HOUR always returns a positive number.

Example

The following statement illustrates the HOUR function:

```
proc cas;
   a=hour(time());
   print "a="
a;

run;
```

These statements produce this result:

```
a=17
```

IBESSEL Function

Returns the value of the modified Bessel function.

Returned data type: DOUBLE

Syntax

IBESSEL(*nu*, *x*, *kode*)

Required Arguments

nu

specifies a numeric constant, variable, or expression.

Range *nu* ≥ 0

Data type DOUBLE

x

specifies a numeric constant, variable, or expression.

Range *x* ≥ 0

Data type DOUBLE

kode

is a numeric constant, variable, or expression that specifies a nonnegative whole number.

Data type DOUBLE

Details

The IBESSEL function returns the value of the modified Bessel function of order *nu* evaluated at *x* (Abramowitz, Stegun 1964; Amos, Daniel, Weston 1977). When *kode* equals 0, the Bessel function is returned. Otherwise, the value of the following function is returned:

$$\varepsilon^{-x} I_{nu}(x)$$

Example

The following statements illustrate the IBESSEL function:

```
proc
cas;

   x=ibessel(2, 2,
0);

   y=ibessel(2, 2,
1);
```

```
   print "x=" x "y="
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.

Returned data type: STRING

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.

```
proc cas;
   data _null_;
      input name $ grade;
      performance = ifc(grade>80, 'Pass', 'Needs Improvement');
      put name= performance=;
      datalines;
   John 74
   Kareem 89
   Kati 100
   Maria 92
;
run;
```

```
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.

```
proc cas;
   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
;
run;
```

```
name=John performance=Needs Improvement
name=Sam performance=Pass
name=Kati performance=Pass
name=Maria performance=Pass
```

IFN Function

Returns a numeric value based on whether an expression is true, false, or missing.

Returned data type: DOUBLE

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 exceed $10,000, then the sales commission is 5% of the total sales. If total sales are less than $10,000, then the sales commission is 2% of the total sales.

```
proc cas;
   data _null_;
      input TotalSales;
      commission=ifn(TotalSales > 10000, TotalSales*.05,
TotalSales*.02);
      put commission=;
      datalines;
   25000
   10000
   500
   10300
;
```

```
run;
```

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 exceed $10,000, then the sales commission is 5% of the total sales. If total sales are less than $10,000, then the sales commission is 2% of the total sales.

```
proc cas;
   data _null_;
      input TotalSales;
      if TotalSales > 10000 then commission = .05 * TotalSales;
         else commission = .02 * TotalSales;
      put commission=;
      datalines;
   25000
   10000
   500
   10300
;
run;
```

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 exceed $10,000, then the sales commission is 5% of the total sales. If total sales are 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.

```
proc cas;
   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;
```

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.

Returned data type: DOUBLE

Syntax

INDEX(*target-expression*, *search-expression*)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

search-expression
specifies any valid expression that evaluates or can be coerced to a character string that is used to search for in *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in single quotation marks.

Details

The INDEX function searches *target-expression*, from left to right, for the first occurrence of the string specified in *search-expression*, and returns the position in *target-expression* of the string's first character. If the string is not found in *target-expression*, INDEX returns a value of 0. If there are multiple occurrences of the string, INDEX returns only the position of the first occurrence.

Comparisons

The VERIFY function returns the position of the first character in *target-expression* that does not contain *search-expression* where the INDEX function returns the position of the first occurrence of *search-expression* that is present in *target-expression*.

Example

The following statements illustrate the INDEX statement:

```
proc cas;
   a='ABC.DEF (X=Y)';
   b='X=Y';
   c=index(a,b);
   print "c=" c;
run;
```

SAS writes the following output to the log:

```
   c=10
```

INDEXC Function

Searches a character expression for specified characters and returns the position of the first occurrence of any of the characters.

Returned data type: DOUBLE

Syntax

INDEXC(*target-expression*, *search-expression*<, ...*search-expression*>)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string that is searched.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

search-expression
specifies the characters to search for in *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip Enclose a literal string of characters in single quotation marks.

Details

The INDEXC function searches *target-expression*, from left to right, for the first occurrence of any character present in the search expressions and returns the position in *target-expression* of that character. If none of the characters in the search expressions are found in *target-expression*, INDEXC returns a value of 0.

Comparisons

The INDEXC function searches for the first occurrence of any individual character that is present within the search expression, whereas the INDEX function searches for the first occurrence of the search expression as a pattern.

Example

The following statements illustrate the INDEXC function:

```
proc
cas;

   a='ABC.DEP
(X2=Y1)';

   x=indexc(a, '0123', ';
()=.');

   print "x="
x;



   b='have a good
day';

   x=indexc(b, 'pleasant',
'very');

   print "x="
x;

run;
```

These statements produce these results:

```
x=4
x=2
```

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.

Returned data type: DOUBLE

Syntax

INDEXW(*target-expression*, *search-expression* <, *delimiter*>)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string and that is searched.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

search-expression
specifies any valid expression that evaluates or can be coerced to a character string and that is searched for in *target-expression*. SAS removes the leading and trailing delimiters from *search-expression*.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

delimiter
specifies a character expression that you want INDEXW to use as a word separator in the character strings. The default delimiter is the blank character.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: If the blank character is a delimiter, order it so that it is not the last character in *delimiter*. Trailing blanks are ignored because *delimiter* is trimmed of trailing blanks.

Details

The INDEXW function searches *target-expression*, from left to right, for the first occurrence of *search-expression* and returns the position in *target-expression* of the substring's first character. If the substring is not found in *target-expression*, then INDEXW returns a value of 0. If there are multiple occurrences of the string, then INDEXW returns only the position of the first occurrence.

The substring pattern must begin and end on a word boundary. For INDEXW, word boundaries are delimiters, the beginning of *target-expression*, and the end of *target-expression*.

TIP INDEXW has the following behavior when *search-expression* contains blank spaces or has a length of 0:

- If both *target-expression* and *search-expression* contain only blank spaces or have a length of 0, then INDEXW returns a value of 1.
- If *search-expression* contains only blank spaces or has a length of 0, and *target-expression* contains character or numeric data, then INDEXW returns a value of 0.

Comparisons

The INDEXW function searches for strings that are words, whereas the INDEX function searches for patterns as separate words or as parts of other words. INDEXC searches for any characters that are present in the excerpts.

Example

The following statements illustrate the INDEXW function.

```
proc
cas;

   s='asdf adog
dog';

   p='dog
';

   a=indexw(s,
p);

   print "a="
a;



   s='abcdef
x=y';


p='def';
```

```
   b=indexw(s,
p);

   print "b="
b;



   c="abc,def@
xyz";

   abc=indexw(c, " abc ",
"@");

   print "abc="
abc;



   d="abc,def@
xyz";

   comma=indexw(d, ",",
"@");

   print "comma="
comma;



   e='abc,def%
xyz';

   def=indexw(e, 'def',
'%,');

   print "def="
def;



   f="abc,def@
xyz";

   at=indexw(f, "@",
"@");

   print "at="
at;



   g="abc,def@
xyz";
```

```
   xyz=indexw(g, " xyz",
"@");

   print "xyz="
xyz;



   h=indexw(trimn(' '),
' ');

   print "h="
h;



   i=indexw(' x y ', trimn('
'));

   print "i="
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
```

INPUTC Function

Enables you to specify a character informat at run time.

Returned data type: STRING, VARCHAR

Syntax

INPUTC(*source*, *informat*<, *w*>)

Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

informat
is a character constant, variable, or expression that contains the character informat that you want to apply to *source*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

w
specifies any valid expression that evaluates to a numeric width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type DOUBLE

Details

If the INPUTC function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

Comparisons

The INPUTN function enables you to specify a numeric informat at run time.

Example

This example shows how to specify character informats.

```
proc cas;
      type=inputc('positive', '$upcase15.');
      type2=inputc('positive', '$upcase15.', 3);
      print "type=" type;
      print "type2=" type2;
   end;
run;
```

The following line is written to the SAS log.

```
type=POSITIVE
type2=POS
```

INPUTN Function

Enables you to specify a numeric informat at run time.

Returned data type: DOUBLE

Syntax

INPUTN(*source*, *informat*<, *w*<, *d*>>)

Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR

informat
is a character constant, variable, or expression that contains the numeric informat that you want to apply to *source*.

Data type CHAR

w
is a numeric constant, variable, or expression that specifies a width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type DOUBLE

d
is a numeric constant, variable, or expression that specifies the number of decimal places to use.

Interaction If you specify a number here, it overrides any decimal-place specification in the informat.

Data type DOUBLE

Comparisons

The INPUTC function enables you to specify a character informat at run time. Using the PUT function is faster because you specify the informat at compile time.

Example

This example shows how to specify numeric informats.

```
proc cas;
    salary = inputn('20,000.00', comma10.2);
    print "salary=" salary;
run;
```

SAS writes the following output to the log:

```
salary=20000
```

INT Function

Returns the whole number, fuzzed to avoid unexpected floating-point results.

Returned data type: DOUBLE

Syntax

INT(*expression*)

Arguments

expression
specifies any expression that evaluates to a numeric value.

Data type DOUBLE

Comparisons

Unlike the INTZ function, the INT function fuzzes the result. If the argument is within 1E-12 of a whole number, the INT function fuzzes the result to be equal to that whole number. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

Example

The following statements illustrate the INT function:

```
proc cas;
   var1=2.1;
   a=int(var1);
   print a;
   a=int(-2.4);
   print a;
   a=int(1+1.e-11);
   print a;
   a=int(-1.6);
   print a;
run;
```

These statements produce these results:

```
2
-2
1
-1
```

INTCINDEX Function

Returns the cycle index when a date, time, or timestamp interval and value are specified.

Returned data type: DOUBLE

Syntax

INTCINDEX({*interval*<*multiple*><*.shift-index*>}, *date-time-value*)

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Note: The possible values of *interval*

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

date-time-value
specifies a date, time, or timestamp value that represents a time period of a specified interval.

Data type DOUBLE

Details

The Basics

The INTCINDEX function returns the index of the seasonal cycle when you specify an interval and a DATE, TIME, or TIMESTAMP value. For example, if the interval is MONTH, each observation in the data corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12. WEEK is the seasonal cycle for an interval that is equal to DAY. This example returns a value of 19 because May 8, 2018, is the third day of the 19th week of the year.

```
cycle_index1 = intcindex('day', 21312);
```

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<multiple><.shift-index>

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', 21185);
```

In this example, the INTINDEX function returns the day of the week.

```
index = intindex('day',21185);
```

In this example, the INTCINDEX function returns the hour of the day.

```
a= intcindex('minute', 50883.620553);
```

In this example, the INTINDEX function returns the minute of the hour.

```
a= intindex('minute', 50883.620553);
```

Example

The following statements illustrate the INTCINDEX function:

```
proc cas;
   cycle_index1=intcindex('day', 45910);
   print "cycle_index1=" cycle_index1;
run;
```

SAS writes these results to the log:

```
cycle_index1=37
```

INTCK Function

Returns the number of interval boundaries of a given kind that lie between two SAS dates, times, or timestamp values encoded as DOUBLE.

Returned data type: DOUBLE

Syntax

INTCK({*interval*<*multiple*><*.shift-index*>}, *start-date*, *end-date*<, '*method*'>)

INTCK(*start-date*, *end-date*<, '*method*'>)

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Data type DOUBLE

start-date
specifies an expression that represents the starting SAS date, time, or timestamp value.

Data type DOUBLE

end-date
specifies an expression that represents the ending SAS date, time, or timestamp value.

Data type DOUBLE

'method'
specifies that intervals are counted using either a discrete or a continuous method. Enclose *method* in quotation marks. *Method* can be one of these values: CONTINUOUS and DISCRETE (counts interval boundaries).

specifies that intervals are counted using either a discrete or a continuous method.

You must enclose *method* in quotation marks. *Method* can be one of these values:

CONTINUOUS
specifies that continuous time is measured. The interval is shifted based on the starting date.

For example, the distance in months between January 15, 2013, and February 15, 2013, is one month.

Alias C or CONT

DISCRETE
specifies that discrete time is measured. The discrete method counts interval boundaries (for example, end of month).

The default discrete method is useful to sort time series observations into bins for processing. For example, daily data can be accumulated to monthly data for processing as a monthly series.

For the DISCRETE method, the distance in months between January 31, 2013, and February 1, 2013, is one month.

Alias D or DISC

Default DISCRETE

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<*multiple*><*.shift-index*>

Calendar Interval Calculations

All values within a discrete time interval are interpreted as being equivalent. This means that the dates of January 1, 2018 and January 15, 2018 are equivalent when you specify a monthly interval. Both of these dates represent the interval that begins on January 1, 2018 and ends on January 31, 2018. You can use the date for the beginning of the interval (January 1, 2018) or the date for the end of the interval (January 31, 2018) to identify the interval. These dates represent all of the dates within the monthly interval.

In the following example, the *start-date* is the SAS value 21185 (Jan. 1, 2018) is equivalent to the first quarter of 2018.

```
qtr=intck('qtr', 21185, 21366);
```

The *end-date* has the SAS date value of 21366 (July 1, 2018) is equivalent to the second quarter of 2018. The interval count, that is, the number of times the beginning of an interval is reached in moving from the *start-date* to the *end-date* is 2.

The INTCK function using the default discrete method counts the number of times the beginning of an interval is reached in moving from the first date to the second. It does not count the number of complete intervals between two dates:

- The following example returns 0, because the two dates are within the same month.

```
proc cas;
  month=intck("month", 21185, 21202);
  print "month=" month;
run;
```

- The following example returns 1, because the two dates lie in different months that are one month apart.

```
proc cas;
```

```
  month=intck("month", 21185, 21216);
  print "month=" month;
run;
```

- The following example returns –1 because the first date is in a later discrete interval than the second date. (INTCK returns a negative value whenever the first date is later than the second date and the two dates are not in the same discrete interval.)

```
proc cas;
  month=intck("month", 21216, 21185);
  print "month=" month;
run;
```

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.

Intervals by Category

Table 4.3 *Intervals Used with Date and Time Functions*

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
Date	DAY	Daily intervals	Each day	Days	DAY3	Three-day intervals starting on Sunday
	WEEK	Weekly intervals of seven days	Each Sunday	Days (1=Sunday ... 7=Saturday)	WEEK.7	Weekly with Saturday as the first day of the week
	WEEKDAY <*days*W>	Daily intervals with Friday-Saturday-Sunday	Each day	Days	WEEKDAY1W	Six-day week with Sunday as a weekend day
		counted as the same day (five-			WEEKDAY35W	Five-day week with Tuesday

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
		day work week with a Saturday-Sunday weekend). *days* identifies the weekend days by number (1=Sunday . .. 7=Saturday). By default, *days*=17.				and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)
	TENDAY	Ten-day intervals (a U.S. automobile industry convention)	First, 11th, and 21st of each month	Ten-day periods	TENDAY4.2	Four ten-day periods starting at the second TENDAY period
	SEMIMONTH	Half-month intervals	First and 16th of each month	Semi-monthly periods	SEMIMONTH2.2	Intervals from the 16th of one month through the 15th of the next month
	MONTH	Monthly intervals	First of each month	Months	MONTH2.2	February-March, April-May, June-July, August-September, October-November, and December-January of the

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
						following year
	QTR	Quarterly (three-month) intervals	January 1	Months	QTR3.2	Three-month intervals starting on April 1, July 1, October 1, and January 1
			April 1			
			July 1			
			October 1			
	SEMIYEAR	Semiannual (six-month) intervals	January 1	Months	SEMIYEAR.3	Six-month intervals, March-August, and September-February
			July 1			
	YEAR	Yearly intervals	January 1	Months		
Datetime	Add DT to any of the date intervals	Interval that corresponds to the associated date interval	Midnight of January 1, 1960		DTMONTH	
					DTWEEKDAY	
Time	SECOND	Second intervals	Start of the day (midnight)	Seconds		
	MINUTE	Minute intervals	Start of the day (midnight)	Minutes		
	HOUR	Hourly intervals	Start of the day (midnight)	Hours		

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4.

The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively.

Example

The following statements illustrate the INTCK function:

```
proc cas;
  qtr=intck('qtr', 21185, 21366);
  print "qtr=" qtr;

  year=intck('year', 21185, 21550);
  print "year=" year;

  year=intck('year', 21185, 21489);
  print "year=" year;

  semiyear=intck('semiyear', 21185, 21915);
  print "semiyear=" semiyear;

  weekvar=intck('week2.2', 21185, 21550);
  print "weekvar=" weekvar;

  wdvar=intck('weekday7w', 21185, 21216);
  print "wdvar=" wdvar;

  newyears=intck('year', 17900, 21550);
  print "newyears="
newyears;

run;
```

These statements produce these results:

```
qtr=2
year=1
year=0
semiyear=4
weekvar=26
wdvar=27
newyears=10
```

INTCYCLE Function

Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.

Returned data type:	STRING, VARCHAR

Syntax

INTCYCLE({*interval*<*multiple*><.*shift-index*>}<, *seasonality*>)

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval*

Tip *Interval* can appear in uppercase or lowercase.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Data type DOUBLE

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.

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default 52

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example The *seasonality* argument in the following example

```
INTCYCLE('MONTH', 3);
```

causes the function call to return the value QTR. The function call

```
INTCYCLE('MONTH');
```

does not have a *seasonality* argument and returns the value YEAR.

Details

The Basics

The INTCYCLE function returns the interval of the seasonal cycle, depending on a date, time, or datetime interval. For example, `INTCYCLE('MONTH');` returns the value YEAR because the months from January through December constitute a yearly cycle. `INTCYCLE('DAY');` returns the value WEEK because the days from Sunday through Saturday constitute a weekly cycle.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<multiple><.shift-index>

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTCYCLE function uses the concept of seasonality and returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the *SAS/ETS User's Guide*.

Example

The following statements illustrate the INTCYCLE function:

```
proc cas;
  cycle_year=intcycle('year');
  print "cycle_year=" cycle_year;

  cycle_quarter=intcycle('qtr');
  print "cycle_quarter=" cycle_quarter;

  cycle_month=intcycle('month', 3);
  print "cycle_month=" cycle_month;

  cycle_month=intcycle('month');
  print "cycle_month=" cycle_month;

  cycle_weekday=intcycle('weekday');
```

```
  print "cycle_weekday=" cycle_weekday;

  cycle_weekday2=intcycle('weekday', 5);
  print "cycle_weekday2=" cycle_weekday2;

  cycle_day=intcycle('day');
  print "cycle_day=" cycle_day;

  cycle_day2=intcycle('day', 10);
  print "cycle_day2=" cycle_day2;

  cycle_second=intcycle('second');
  print "cycle_second=" 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
```

INTFIT Function

Returns a time interval that is aligned between two dates.

Returned data type: STRING, VARCHAR

Syntax

INTFIT(*expresssion-1*, *expression-2*, '*type*')

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string and that represents a SAS date or datetime value.

Data type DOUBLE

'type'
specifies whether the arguments are SAS date values, datetime values, or a row. The following values for *type* are valid: d (date), dt (datetime), and obs (rows).

specifies whether the arguments are SAS date values, datetime values, or a row.

The following values for *type* are valid:

d specifies that *expression-1* and *expression-2* are date values.

dt specifies that *expression-1* and *expression-2* are datetime values.

obs specifies that *expression-1* and *expression-2* are rows.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Example

The following example shows the interval that is aligned between two dates. The *type* argument in this example identifies the input as date values.

```
proc cas;
  sasdate1=17900;
  sasdate2=21550;
  intfit=intfit(sasdate1, sasdate2, 'd');
  print "intfit=" intfit;
run;
```

The following line is written to the SAS log.

```
intfit=DAY3650.3301
```

INTGET Function

Returns a time interval based on three date or datetime values.

Returned data type: STRING, VARCHAR

Syntax

INTGET(*date-1*, *date-2*, *date-3*)

Argument

date

specifies any valid expression that evaluates to a SAS date or datetime value.

Data type DOUBLE

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 .

Note: If INTGET cannot determine a matching interval, then the function returns a missing value. No message is written to the SAS log.

Example

The following statements illustrate the INTGET function:

```
proc cas;
```

```
   date1=inputn('29Sep2024',date9.); date2=inputn('06Oct2024',date9.);
date3=inputn('13Oct2024',date9.); /*1*/
   print(NOTE) "interval1( " date1 "," date2 "," date3 " )="
intget(date1,date2,date3);                    /*2*/
   date1=inputn('30Sep2024',date9.); date2=inputn('07Oct2024',date9.);
date3=inputn('14Oct2024',date9.);
   print(NOTE) "interval1( " date1 "," date2 "," date3 " )="
intget(date1,date2,date3);
   date1=inputn('01jan2024',date9.); date2=inputn('01Feb2024',date9.);
date3=inputn('01Mar2024',date9.);
   print(NOTE) "interval1( " date1 "," date2 "," date3 " )="
intget(date1,date2,date3);
   date1=inputn('01Jul2024',date9.); date2=inputn('01Sep2024',date9.);
date3=inputn('01Nov2024',date9.);
   print(NOTE) "interval1( " date1 "," date2 "," date3 " )="
intget(date1,date2,date3);
   date1=inputn('01jan2020',date9.); date2=inputn('01jan2021',date9.);
date3=inputn('01Jan2022',date9.);
   print(NOTE) "interval1( " date1 "," date2 "," date3 " )="
intget(date1,date2,date3);
   date1=inputn('29feb2020',date9.); date2=inputn('28feb2022',date9.);
date3=inputn('29feb2024',date9.);
   print(NOTE) "interval1( " date1 "," date2 "," date3 " )="
intget(date1,date2,date3);
run;
quit;
```

1 The INTGET function requires three parameters, which must be double-precision SAS data values. CASL dates are stored as ANSI DATE type values and cannot be presented to INTGET in hat format. Here, the INPUTN function produces SAS double-precision date values for each input text string.

2 The parameters presented to INTGET are double-precision numeric SAS date values.

The program writes the following output to the log:

```
NOTE: interval1( 23648 ,23655 ,23662  )=WEEK
NOTE: interval1( 23649 ,23656 ,23663  )=WEEK.2
NOTE: interval1( 23376 ,23407 ,23436  )=MONTH
NOTE: interval1( 23558 ,23620 ,23681  )=MONTH2
NOTE: interval1( 21915 ,22281 ,22646  )=YEAR
NOTE: interval1( 21974 ,22704 ,23435  )=YEAR2.2
```

INTINDEX Function

Returns the seasonal index when a date, time, or timestamp interval and value are specified.

Returned data type: DOUBLE

Syntax

INTINDEX({*interval*<*multiple*><.*shift-index*>}, *date-value*<, *seasonality*>)

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval*

Tip *Interval* can appear in uppercase or lowercase.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

date-value

specifies a date, time, or timestamp value that represents a time period of the given interval.

Data type DOUBLE

seasonality

specifies a number or a cycle. This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

specifies a number or a cycle.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example In this example, the following functions produce the same result.

```
INTINDEX('MONTH', sasdate, 3);
INTINDEX('MONTH', sasdate, 'QTR');
```

Seasonality in the first example is a number (the number of months), and in the second example *seasonality* is a cycle (QTR).

Details

The Basics

The INTINDEX function returns the seasonal index when you supply an interval and an appropriate date, time, or timestamp value. The seasonal index is a number that represents the position of the date, time, or timestamp value in the seasonal cycle of the specified interval. This example returns a value of 12 because there are 12 months in a yearly cycle and December is the 12th month of the year.

```
sasdate=to_double(date'2012-12-01');
x=intindex('month', sasdate);
put x;
```

In the following examples, INTINDEX returns the same value (1) because both statements have dates that occur in the first quarter of the year 2013.

```
sasdate=to_double(date'2013-01-01');
x=intindex('qtr', sasdate);
put x;

sasdate=to_double(date'2013-03-31');
y=intindex('qtr', sasdate);
put y;
```

The following example returns a value of 6 because daily data is weekly periodic and December 7, 2012, is a Friday, the sixth day of the week.

```
sasdate=to_double(date'2012-12-07');
x=intindex('day', sasdate);
put x;
```

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<multiple><.shift-index>

How *Interval* and *Date-Time-Value* Are Related

To correctly identify the seasonal index, the interval should agree with the date, time, or timestamp value. For example, `intindex('month', '01DEC2012'd);` returns a value of 12 because there are 12 months in a yearly interval and December is the 12th month of the year. The MONTH interval requires a SAS date value. The following example, returns a value of 6 because there are seven days in a weekly interval and December 7, 2012, is a Friday, the sixth day of the week.

```
sasdate=to_double(date'2012-12-07');
x=intindex('day', sasdate);
put x;
```

The DAY interval requires a SAS date value.

This example returns a missing value because the QTR interval expects the date to be a SAS date value rather than a TIMESTAMP value.

```
sasdate=to_double(timestamp'2013-01-01 00:00:00');
x=intindex('qtr', sasdate);
put x;
```

This example returns a value of 12. The DTMONTH interval requires a TIMESTAMP value.

```
sasdate=to_double(timestamp'2013-12-01 00:00:00');
x=intindex('dtmonth', sasdate);
put x;
```

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTINDEX function uses the concept of seasonality and returns the seasonal index when a date, time, or timestamp interval and value are specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the *SAS/ETS User's Guide*.

Comparisons

The INTINDEX function returns the seasonal index whereas the INTCINDEX function returns the cycle index.

In the following example, the INTINDEX function returns 5 because April 4, 2013 is on a Thursday, the fifth day of the week.

```
sasdate=to_double(date'2013-04-04');
x = intindex('day', sasdate);
put x;
```

Using the same date, the INTCINDEX function returns 14 because April 4, 2013 is the 14th week of the year.

```
sasdate=to_double(date'2013-04-04');
x = intcindex('day', sasdate);
put x;
```

In this example, the INTINDEX function returns the minute of the hour.

```
sasdate=to_double(timestamp'2012-09-01 06:05:04');
x = intindex('minute', sasdate);
put x;
```

Using the same date and time, the INTCINDEX function returns the hour of the day.

```
sasdate=to_double(timestamp'2012-09-01 06:05:04');
y = intcindex('minute', sasdate);
put y;
```

In the example `intseas('interval');`, INTSEAS returns the maximum number that could be returned by `intindex('interval', date);`.

Example

The following statements illustrate the INTINDEX function:

```
proc cas;
   index1 = intindex('day', 19584);
   print "index1=" index1;
   index2 = intindex('minute', 50883.620553);
   print "index2=" index2;
run;
```

The program writes the following output to the log:

```
index1=4
index2=9
```

See Also

Other References:

- *SAS/ETS User's Guide*

INTNX Function

Increments a SAS date, time, or datetime value encoded as a DOUBLE, and returns a SAS date, time, or datetime value encoded as a DOUBLE.

Returned data type: DOUBLE

Syntax

INTNX({*interval*<*multiple*><*.shift-index*>}, *start-from*, *increment*<, '*alignment*'>)

INTNX(*start-from*, *increment*<, '*alignment*'>)

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type: DOUBLE

Example: YEAR2 specifies a two-year, or biennial, interval type.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

start-from
specifies an expression that represents a SAS date, time, or datetime value encoded as a DOUBLE and that identifies a starting point.

Data type: DOUBLE

increment
specifies a negative, positive, or zero whole number that represents the number of date, time, or datetime intervals. *Increment* is the number of intervals to shift the value of *start-from*.

Data type DOUBLE

'alignment'

controls the position of SAS dates within the interval. You must enclose *alignment* in quotation marks. *Alignment* can be one of these values: BEGINNING, MIDDLE, END, and SAME. SAME specifies that the date that is returned has the same alignment as the input date. For more information, see SAS DS2 Language Reference.

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

Default BEGINNING

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

The INTNX function increments a date, time, or datetime value by intervals such as DAY, WEEK, QTR, and MINUTE, or a custom interval that you define. The increment is based on a starting date, time, or datetime value, and on the number of time intervals that you specify.

The INTNX function returns the SAS date value for the beginning date, time, or datetime value of the interval that you specify in the *start-from* argument. (To convert the date value to a calendar date, use any valid DS2 date format, such as the DATE9. format.) The following example shows how to determine the date of the start of the week that is six weeks from the week of October 17, 2011.

```
sasdate=to_double(date'2011-10-17');
x=intnx('week', sasdate, 6);
print put x date9.;
```

INTNX returns the value 27NOV2011.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<multiple><.shift-index>

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.

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.

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.

Adjusting Dates

The INTNX function automatically adjusts for the date if the date in the interval that is incremented does not exist.

Example: Using the INTNX Function

The following statements illustrate the INTNX function.

```
proc cas;
   date1=inputn('29Sep2024','date9.');
date2=inputn('06Oct2024','date9.');                /* 1 */
   print(NOTE) 'date1=' putn(date1,'yymmddd10.')'.';
   print(NOTE) 'INTNX("WEEK",date1,2)='
putn(INTNX("WEEK",date1,2),yymmddd10.);           /* 2 */

   date1=inputn('30Sep2024',date9.);
   print(NOTE) 'date1=' putn(date1,'yymmddd10.')'.';
   print(NOTE) 'INTNX("WEEK2.2",date1,2)='
putn(INTNX("WEEK2.2",date1,2),yymmddd10.);

   date1=inputn('01Jan2024',date9.);
   print(NOTE) 'date1=' putn(date1,'yymmddd10.')'.';
   print(NOTE) 'INTNX("MONTH,date1,2)='
putn(INTNX("MONTH",date1,2),yymmddd10.);

   date1=inputn('01Jan2024',date9.);
   print(NOTE) 'date1=' putn(date1,'yymmddd10.')'.';
   print(NOTE) 'INTNX("MONTH2",date1,2)='
putn(INTNX("MONTH2",date1,2),yymmddd10.);

   date1=inputn('01Jan2020',date9.);
   print(NOTE) 'date1=' putn(date1,'yymmddd10.')'.';
   print(NOTE) 'INTNX("YEAR",date1,2)='
putn(INTNX("YEAR",date1,2),yymmddd10.);

   date1=inputn('29Feb2020',date9.);
   print(NOTE) 'date1=' putn(date1,'yymmddd10.')'.';
   print(NOTE) 'INTNX("YEAR2.2",date1,2)='
putn(INTNX("YEAR2.2",date1,2),'yymmddd10.');
run;
```

The program writes the following output to the log:

```
NOTE: date1=2024-09-29.
NOTE: INTNX("WEEK",date1,2)=2024-10-13
NOTE: date1=2024-09-30.
NOTE: INTNX("WEEK2.2",date1,2)=2024-10-21
NOTE: date1=2024-01-01.
```

```
NOTE: INTNX("MONTH,date1,2)=2024-03-01
NOTE: date1=2024-01-01.
NOTE: INTNX("MONTH2",date1,2)=2024-05-01
NOTE: date1=2020-01-01.
NOTE: INTNX("YEAR",date1,2)=2022-01-01
NOTE: date1=2020-02-29.
NOTE: INTNX("YEAR2.2",date1,2)=2024-02-01
```

INTRR Function

Returns the internal rate of return as a decimal value.

Returned data type: DOUBLE

Syntax

INTRR(*freq*, *c0, c1*<*..., cn*>)

Arguments

freq
is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *freq* > 0

Data type DOUBLE

Tip The case *freq* = 0 is a flag to allow continuous compounding.

***c0, c1*< *..., cn*>**
are numeric, the optional cash payments.

Data type DOUBLE

Details

The INTRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c0*, *c1*, ..., *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument *freq* > 0 describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

The internal rate of return is the interest rate such that the sequence of payments has a 0 net present value.It is given by the following equation.

$$r = \begin{cases} \frac{1}{x^{freq}} - 1 & freq > 0 \\ -log_e(x) & freq = 0 \end{cases}$$

In this equation, *x* is the real root of the polynomial.

$$\sum_{i=0}^{n} c_i x^i = 0$$

In the case of multiple roots, one real root is returned and a warning is issued concerning the non-uniqueness of the returned internal rate of return. Depending on the value of payments, a root for the equation does not always exist. In that case, a missing value is returned.

Missing values in the payments are treated as 0 values. When *freq* > 0, the computed rate of return is the effective rate over the specified base period. To compute a quarterly internal rate of return (the base period is three months) with monthly payments, set *freq* to 3.

If *freq* is 0, continuous compounding is assumed and the base period is the time interval between two consecutive payments. The computed internal rate of return is the nominal rate of return over the base period. To compute with continuous compounding and monthly payments, set *freq* to 0. The computed internal rate of return 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 as

```
proc
cas;

   rate=intrr(1, -400, 100, 200,
300);

   print
rate;

run;
```

These statements produce this result:

```
0.1943770996
```

The value that is returned is 0.19437709967.

INTSEAS Function

Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.

Returned data type: DOUBLE

Syntax

INTSEAS({*interval*<*multiple*><.*shift-index*>}<, *seasonality*>)

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies year-based intervals.

multiple

specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type: DOUBLE

Example: YEAR2 specifies a two-year, or biennial, interval type.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Restrictions: The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2

specifies bimonthly periods starting on the first day of even-numbered months.

Data type DOUBLE

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

seasonality

specifies a numeric value. This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Default 52

Data type DOUBLE, CHAR, NCHAR, NVARCHAR, VARCHAR

Example The *seasonality* argument in the following example

```
INTSEAS('MONTH', 'qtr');
```

causes the function call to return the value 3. The function call

```
INTSEAS('MONTH');
```

does not have a *seasonality* argument and returns the value 12.

Details

The Basics

The INTSEAS function returns the number of intervals in a seasonal cycle. For example, when the interval for a time series is described as monthly, then many procedures use the option INTERVAL=MONTH. Each observation in the data then corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12.

Quarterly data is also considered to be periodic for a one-year period. A year contains four quarters, so the number of intervals in a seasonal cycle is four.

The periodicity is not always one year. For example, INTERVAL=DAY is considered to have a period of one week. Because there are seven days in a week, the number of intervals in the seasonal cycle is seven.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<multiple><.shift-index>

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTSEAS function uses the concept of seasonality and returns the length of the seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and forecasting, see the *SAS/ETS User's Guide*.

Example

The following statements illustrate the INTCYCLE function:

```
proc cas;
   cycle_years=intseas('year');
   print "cycle_years=" cycle_years;
   cycle_smiyears=intseas('semiyear');
   print "cycle_smiyears=" cycle_smiyears;
   cycle_quarters=intseas('quarter');
   print "cycle_quarters=" cycle_quarters;
   cycle_number=intseas('month', 'qtr');
   print "cycle_number=" cycle_number;
   cycle_months=intseas('month');
   print "cycle_months=" cycle_months;
   cycle_smimonths=intseas('semimonth');
   print "cycle_smimonths=" cycle_smimonths;
   cycle_tendays=intseas('tenday');
   print "cycle_tendays=" cycle_tendays;
   cycle_weeks=intseas('week');
   print "cycle_weeks=" cycle_weeks;
   cycle_wkdays=intseas('weekday');
   print "cycle_wkdays=" cycle_wkdays;
   cycle_hours=intseas('hour');
   print "cycle_hours=" cycle_hours;
   cycle_minutes=intseas('minute');
   print "cycle_minutes=" cycle_minutes;
   cycle_month2=intseas('month2.2');
   print "cycle_month2=" 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

Other References:

- *SAS/ETS User's Guide*

INTSHIFT Function

Returns the shift interval that corresponds to the base interval.

Returned data type: STRING, VARCHAR

Syntax

INTSHIFT(*interval*<*multiple*><.*shift-index*>)

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: *Interval* can appear in uppercase or lowercase.

Example: YEAR specifies yearly intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type: DOUBLE

Example: YEAR2 consists of two-year, or biennial, periods.

shift-index

specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Data type DOUBLE

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

Details

The Basics

The INTSHIFT function returns the shift interval that corresponds to the base interval. For custom intervals, the value that is returned is the base custom interval name. INTSHIFT ignores multiples of the interval and interval shifts.

The INTSHIFT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<multiple><.shift-index>

Example

The following statements illustrate the INTSHIFT function:

```
proc cas;
   shift1=intshift('year');
   print shift1;
   shift2=intshift('dtyear');
   print shift2;
   shift3=intshift('minute');
   print shift3;
   shift4 = intshift('weekdays');
   print shift4;
   shift5=intshift('weekday5.4');
   print shift5;
   shift6=intshift('qtr');
   print shift6;
   shift7=intshift('dttenday');
   print shift7;
run;
```

The following output is printed to the SAS Log:

```
MONTH
DTMONTH
DTMINUTE
WEEKDAY
WEEKDAY
MONTH
DTTENDAY
```

INTTEST Function

Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.

Returned data type: DOUBLE

Syntax

INTTEST(*interval*<*multiple*><.*shift-index*>)

Arguments

interval
specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Tip *Interval* can appear in uppercase or lowercase.

Example YEAR specifies year-based intervals.

multiple
specifies an optional multiplier that sets the interval equal to a multiple of the period of the basic interval type.

Data type DOUBLE

Example YEAR2 specifies a two-year, or biennial, interval type.

shift-index
specifies an optional shift index that shifts the interval to start at a specified subperiod starting point.

Data type DOUBLE

Example YEAR.3 specifies yearly periods shifted to start on the first of March of each calendar year and to end in February of the following year.

Details

The Basics

The INTTEST function checks for a valid interval name. This function is useful when checking for valid values of *multiple* and *shift-index*.

The INTTEST function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant.

Intervals

Intervals can be basic or complex. The basic interval is a unit of measurement that SAS can count within an elapsed period of time, such as a DAY, MONTH, or HOUR. Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications.

The interval syntax is as follows:

interval<multiple><.shift-index>

Example

In the following examples, SAS returns a value of 1 if the *interval* argument is valid, and 0 if the interval argument is invalid.

```
proc cas;
   test1 = inttest('month');
   print "test1=" test1;
   test2 = inttest('week6.13');
   print "test2=" test2;
   test3 = inttest('tenday');
   print "test3=" test3;
   test4 = inttest('twoweeks');
   print "test4=" test4;
   test5 = inttest('hour2.2');
   print "test5=" test5;
run;
```

The following output is printed to the SAS Log:

```
test1=1
test2=1
test3=1
test4=0
test5=1
```

INTZ Function

Returns the whole number portion of the argument, using zero fuzzing.

Returned data type: DOUBLE

Syntax

INTZ(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The following rules apply:

- If the value of the argument is an exact whole number, INTZ returns that whole number.
- If the argument is positive and not a whole number, INTZ returns the largest whole number that is less than the argument.
- If the argument is negative and not a whole number, INTZ returns the smallest whole number that is greater than the argument.

Comparisons

Unlike the INT function, the INTZ function uses zero fuzzing. If the argument is within 1E-12 of a whole number, the INT function fuzzes the result to be equal to that whole number. The INTZ function does not fuzz the result. Therefore, with the INTZ function, you might get unexpected results.

Example

The following statements illustrate the INTZ function:

```
proc cas;
  x=intz(5/2);
  y=intz(5.9999999999);
  z=intz(CONSTANT('PI'));
  print "X=" X "Y=" Y "Z=" Z;
run;
```

The program writes the following output to the log:

```
X=2 Y=5 Z=3
```

IPMT Function

Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.

Returned data type: DOUBLE

Syntax

IPMT(*rate*, *period*, *number-of-periods*, *principal-amount*<, *future-amount*><, *type*>)

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

period
specifies the payment period for which the interest payment is computed.

Requirement *Period* must be a positive whole number that is less than or equal to the value of *number-of-periods*.

Data type INTEGER

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type INTEGER

principal-amount
specifies the principal amount of the loan.

Data type DOUBLE

Note Zero is assumed if a missing value is specified.

future-amount
specifies the future amount.

Data type DOUBLE

Notes *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings.

Zero is assumed if *future-amount* is omitted or if a missing value is specified.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments.

Data type INTEGER

Note 0 is assumed if *type* is omitted or if a missing value is specified.

Example

The interest payment on the first periodic payment of an $8,000 loan, where the nominal annual interest rate is 10% and the end-of-period monthly payments are 36, is computed as follows:

```
proc cas;
   InterestPaid1 = ipmt(0.1/12, 1, 36, 8000);
   print InterestPaid1;
run;
```

This computation returns a value of 66.6666666666666.

If the same loan has beginning-of-period payments, then the interest payment can be computed as follows:

- ```
  proc cas;
     InterestPaid2 = ipmt(0.1/12, 1, 36, 8000, 0, 1);
     print InterestPaid2;
  run;
  ```

  This computation returns a value of 0.

- ```
  proc cas;
     InterestPaid3 = ipmt(0.1, 3, 3, 8000);;
     print InterestPaid3;
  run;
  ```

  This computation returns a value of 292.447129909366.

- ```
  proc cas;
     InterestPaid4 = ipmt(0.09/12, 359, 360, 125000, 0, 1);
     print InterestPaid4;
  ```
  ```

```
run;
```

This computation returns a value of 14.8075736630449.

IQR Function

Returns the interquartile range.

Returned data type: DOUBLE

Syntax

IQR(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

If all arguments have null or missing values, the result is a null or missing value depending on whether you are in ANSI mode or SAS mode.

Otherwise, the result is the interquartile range of the non-null or nonmissing values. The formula for the interquartile range is the same as the one that is used in the Base SAS UNIVARIATE procedure.

Example

The following statement illustrates the IQR function:

```
proc cas;
   a=iqr(2,4,1,3,999999);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   2
```

IRR Function

Returns the internal rate of return as a percentage.

Returned data type: DOUBLE

Syntax

IRR(*freq*, *c1, c2* <..., *cn*>)

Arguments

freq
is numeric, the number of payments over a specified base period of time that is associated with the desired internal rate of return.

Range *freq* > 0.

Data type DOUBLE

Tip The case *freq* = 0 is a flag to allow continuous compounding.

***c1, c2*< ..., *cn*>**
are numeric, the optional cash payments.

Requirement At minimum, two cash payment values are required.

Data type DOUBLE

Details

The IRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c1*, *c2*, ..., *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument *freq* > 0 describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

Comparisons

The IRR function is identical to INTRR, except that in the IRR function, the internal rate of return is a percentage.

Example

For an initial outlay of $400 and expected payments of $100, $200, and $300 over the following three years, the annual internal rate of return as a percentage can be expressed as

```
proc cas;
   rate=irr(1,-400,100,200,300);
   print rate;
run;
```

The value that is returned is 19.437709962747.

JBESSEL Function

Returns the value of the Bessel function.

Returned data type: DOUBLE

Syntax

JBESSEL(*nu*, *x*)

Required Arguments

nu

specifies a numeric constant, variable, or expression.

Range $nu \geq 0$

x

specifies a numeric constant, variable, or expression.

Range $x \geq 0$

Details

The JBESSEL function returns the value of the Bessel function of order *nu* evaluated at *x* (For more information, see Abramowitz and Stegun 1964; Amos, Daniel, and Weston 1977).

Example

The following statements illustrate the JBESSEL function:

```
proc cas;
  x=jbessel(2, 2);
  print x;
run;
```

The following output is printed to the SAS Log:

```
   0.3528340286
```

JULDATE Function

Returns the Julian date from a SAS date value.

Returned data type: DOUBLE

Syntax

JULDATE(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE function converts a SAS date value to a Julian date. If *date* falls within the 100-year span defined by the system option YEARCUTOFF=, the result has three, four or five digits: In a five-digit result, the first two digits represent the year, and the next three digits represent the day of the year (1 to 365, or 1 to 366 for leap years). As leading zeros are dropped from the result, the year portion of a Julian date can be omitted (for years ending in 00) or it can have only one digit (for years ending 01–09). Otherwise, the result has seven digits: the first four digits represent the year, and the next three digits represent the day of the year.

For years that end between 00–09, you can format the five-digit Julian date by using the Z5. format.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following statements illustrate the JULDATE function:

```
proc cas;
  julian=juldate(mdy(12,31,2013));
  print julian;
run;
```

The following output is printed to the SAS Log:

```
   13365
```

JULDATE7 Function

Returns a seven-digit Julian date from a SAS date value.

Returned data type: DOUBLE

Syntax

JULDATE7(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE7 function returns a seven-digit Julian date from a SAS date value. The first four digits represent the year, and the next three digits represent the day of the year.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four-digit year. Thus, JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two-digit year.

Example

The following statements illustrate the JULDATE7 function:

```
proc cas;
  julian=juldate7(mdy(12,31,2006));
  print julian;
run;
```

The following output is printed to the SAS Log:

```
   2006365
```

KURTOSIS Function

Returns the kurtosis.

Returned data type: DOUBLE

Syntax

KURTOSIS(*expression-1, expression-2, expression-3, expression-4* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least four non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

Details

Kurtosis is primarily a measure of the heaviness of the tails of a distribution. Large kurtosis values indicate that the distribution has heavy tails.

Null values and missing values are ignored and are not included in the computation.

If all non-null or nonmissing arguments have equal values, the kurtosis is mathematically undefined and the KURTOSIS function returns a null or missing value.

Example

The following statements illustrate the KURTOSIS function:

```
proc cas;
   a=kurtosis(5,9,3,6);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   0.928
```

LARGEST Function

Returns the *k*th largest non-null or nonmissing value.

Returned data type: DOUBLE

Syntax

LARGEST(*k*, *expression* <, ...*expression*>)

Arguments

k
specifies any valid expression that evaluates to a numeric value that represents the largest value to return. For example, if *k* is 2, the LARGEST function returns the second largest value from the list of expressions.

Data type DOUBLE

expression
specifies any valid expression that evaluates to a numeric value and that is to be searched.

Data type DOUBLE

Details

If *k* is null or missing, less than zero, or greater than the number of values, the result is a null or missing value. Otherwise, if *k* is greater than the number of non-null or nonmissing values, the result is a null or missing value.

Example

The following statements illustrate the LARGEST function:

```
proc cas;
   k=1;
   largest1=largest(k, 456, 789, .Q, 123);
   print largest1;
run;
```

The following output is printed to the SAS Log:

```
   789
```

LCM Function

Returns the least common multiple for a set of whole numbers.

Returned data type: DOUBLE

Syntax

LCM(*expression*-1, *expression*-2 <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a whole number.

Requirement At least two arguments are required.

Data type DOUBLE

Details

The least common multiple is the smallest number that two or more numbers will divide into evenly.

Example

The following statements illustrate the LCM function:

```
proc cas;
   b=lcm(25,70,85,130);
   print b;
run;
```

The following output is printed to the SAS Log:

```
   77350
```

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.

Returned data type: DOUBLE

Syntax

LCOMB(*n*, *r*)

Required Arguments

n

is a nonnegative whole number that represents the total number of elements from which the sample is chosen.

r

is a nonnegative whole number that represents the number of chosen elements.

Restriction $r \leq n$

Comparisons

The LCOMB function computes the logarithm of the COMB function.

Example

The following statements illustrate the LCOMB function:

```
proc cas;
   x=lcomb(5000, 500);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   1621.4411361
```

LEFT Function

Left aligns a character expression.

Returned data type: STRING, VARCHAR

Syntax

LEFT(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

LEFT returns a character string with leading blanks moved to the end of the value.

Example

The following statements illustrate the LEFT function:

```
proc cas;
   a='  END-OF-YEAR';
   b=left(a);
   print b;
run;
```

The following output is printed to the SAS Log:

```
END-OF-YEAR
```

LENGTH Function

Returns the length of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string.

Alias: LENGTHN

Returned data type: BIGINT, INTEGER

Syntax

LENGTH(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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

The following statements illustrate the LENGTH function:

```
proc cas;
   a=length('ABCDEF    ');
   print a;
run;
```

The following output is printed to the SAS Log:

```
   6
```

LENGTHC Function

Returns the length of a character string, including trailing blanks, in bytes.

Returned data type: BIGINT, INTEGER

Syntax

LENGTHC(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

For fixed-length variables, LENGTHC returns the character length of a character string, including trailing blanks. If *expression* is a fixed-length variable, the value returned by LENGTHC is equal to the declared length of the variable. If *expression* is a varying-length variable, the value returned by LENGTHC is less than or equal to the declared length of the variable. If the value of *expression* is missing and contains blanks, LENGTHC returns the number of blanks in *expression*. If *expression*

is a numeric expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string.

Comparisons

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTH function returns the length of a character string, excluding trailing blanks. LENGTHC always returns a value that is greater than or equal to the value returned by LENGTH.
- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. For fixed-length character strings, LENGTHC and LENGTHM always return the same value. For varying-length character strings, LENGTHC always returns a value that is less than or equal to the value returned by LENGTHM. However, with an expression argument, LENGHTM always returns a null value, whereas LENGTHC returns the length, in characters, of the character value.

Example

The following statements illustrate the LENGTHC function::

```
proc cas;
   a=lengthc('string with trailing blanks    ');
   print "results=" a;
run;

proc cas;
   string1='The power to know.  ';
   a=lengthc(string1);
   print "results=" a;
run;

proc cas;
   a=lengthc('');
   print "results=" a;
run;
```

The following output is printed to the SAS Log:

```
   results=31
   results=20
   results=0
```

LENGTHM Function

Returns the amount of memory, in bytes, that could or might be allocated for a character string.

Returned data type: BIGINT, INTEGER

Syntax

LENGTHM(*variable*)

Arguments

variable
specifies any valid string variable. An expression that evaluates or can be coerced to a string value is an invalid argument.

Restriction: Only variable arguments are allowed. Expressions and literals are not valid and will return a null byte length

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The LENGTHM function returns the maximum amount of memory, in bytes, that might be allocated for a character variable. This is true regardless of the character data type qualifiers such as NATIONAL, VARYING, CHARACTER SET, and so on.

The value returned by the LENGTHM function is independent of the current value stored in the variable and might be greater than the currently allocated byte length of the variable. This is true for both fixed-length and varying-length character variables. For example, if you have a CHAR(100) CHARACTER SET UTF-8 variable or a VARCHAR(100) CHARACTER SET UTF-8 variable, the LENGTHM function returns 400 bytes regardless of the value stored in the variable.

For a literal or an expression, the value returned by the LENGTHM function is a null byte length and a warning is issued that the argument is invalid for the function.

Comparisons

The LENGTHM function returns the amount of memory in bytes that is allocated for a character string, whereas the LENGTH and LENGTHC functions return the length, in characters, of a character value. With a variable argument, LENGTHM always returns a value that is greater than or equal to the values returned by LENGTH and LENGTHC. With an expression argument, LENGHTM always returns a null value, whereas LENGTH and LENGTHC return the length, in characters, of the character value.

Example

The following statements illustrate the LENGTHM function:

```
proc cas;
   a='ABCDEF    ';
   b=lengthm(a);
   print b;
run;
```

The following output is printed to the SAS Log:

```
   2147483647
```

```
proc cas;
   string1='the power to know.  ';
   a=lengthm(string1);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   2147483647
```

LENGTHN Function

Returns the length of a character string, excluding trailing blanks.

Alias: LENGTH

Returned data type: BIGINT, INTEGER

Syntax

LENGTHN(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The LENGTHN function returns an integer that represents the position of the rightmost non-blank character in *string*. 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 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 in characters, 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

The following statements illustrate the LENGTHN function:

```
proc cas;
   c=lengthn('   abc   ');
   print c;

   d=lengthn('abc   ');
   print d;
run;
```

The following output is printed to the SAS Log:

```
   6
   3
```

LFACT Function

Computes the logarithm of the FACT (factorial) function.

Returned data type: DOUBLE

Syntax

LFACT(*n*)

Required Argument

n
is a whole number that represents the total number of elements from which the sample is chosen.

Details

The LFACT function computes the logarithm of the FACT function.

Example

The following statements illustrate the LFACT function:

```
proc cas;
   x=lfact(5000);
   print x;
   y=lfact(100);
   print y;
run;
```

The following output is printed to the SAS Log:

```
   37591.143509
   363.73937556
```

LGAMMA Function

Returns the natural logarithm of the Gamma function.

Returned data type: DOUBLE

Syntax

LGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: Must be a positive number.

Data type: DOUBLE

Example

The following statements illustrate the LGAMMA function:

```
proc cas;
   a=lgamma(2);
   print a;
   a=lgamma(1.5);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   0
   -0.120782238
```

LOG Function

Returns the natural logarithm (base e) of a numeric value expression.

Returned data type: DOUBLE

Syntax

LOG(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: Must be a positive number.

Data type: DOUBLE

Example

The following statements illustrate the LOG function:

```
proc cas;
   a=log(1.0);
   print a;
   a=log(10.0);
   print a;
run;
```

The following output is printed to the SAS Log:

```
0
2.302585093
```

LOG10 Function

Returns the base-10 logarithm of a numeric value expression.

Returned data type: DOUBLE

Syntax

LOG10(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: Must be a positive number.

Data type: DOUBLE

Example

The following statements illustrate the LOG10 function:

```
proc cas;
   a=log10(1.0);
   print a;
   a=log10(10.0);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   0
   1
```

LOG1PX Function

Returns the log of 1 plus the argument.

Returned data type: DOUBLE

Syntax

LOG1PX(*x*)

Arguments

x

specifies a numeric variable, constant, or expression.

Data type DOUBLE

Details

The LOG1PX function computes the log of 1 plus the argument. The LOG1PX function is mathematically defined by the following equation, where –1 < x:

$$LOG1PX(x) = log(1 + x)$$

When x is close to 0, `LOG1PX(x)` can be more accurate than `LOG(1+x)`.

Examples

Example 1: Computing the Log with the LOG1PX Function

The following example computes the log of 1 plus the value 0.5.

```
proc cas;
      x=log1px(0.5);
      print "x=" x;
run;
```

SAS writes the following output to the log:

```
x=0.40546510810816
```

Example 2: Comparing the LOG1PX Function with the LOG Function

In the following example, the value of X is computed by using the LOG1PX function. The value of Y is computed by using the LOG function.

```
proc cas;
      x=log1px(1.e-5);
      print x hex16.;
      y=log(1+1.e-5);
```

```
        print y hex16.;
run;
```

SAS writes the following output to the log:

```
9.99995E-6 hex16.
9.99995E-6 hex16.
```

LOG2 Function

Returns the base 2 logarithm of a numeric value expression.

Returned data type: DOUBLE

Syntax

LOG2(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: Must be a positive number.

Data type: DOUBLE

Example

The following statements illustrate the LOG2 function:

```
proc cas;
   a=log2(8.0);
   print a;
   a=log2(4);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   3
   2
```

LOGBETA Function

Returns the logarithm of the beta function.

Returned data type: DOUBLE

Syntax

LOGBETA(*a*, *b*)

Arguments

a

is the first shape parameter, where *a*>0.

Data type DOUBLE

b

is the second shape parameter, where *b*>0.

Data type DOUBLE

Details

The LOGBETA function is mathematically given by the equation

$$log(\beta(a,b)) = log(\Gamma(a)) + log(\Gamma(b)) - log(\Gamma(a+b))$$

In the equation, $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, LOGBETA returns a missing value.

Example

The following DS2 statements compute the logarithm of the beta function. The first shape parameter is 5 and the second shape parameter is 3.

```
proc cas;
      y=logbeta(5,3);
      print "y=" y;
run;
```

The following line is written to the SAS log.

```
y=-4.65396035015752
```

LOGISTIC Function

Returns the logistic transformation of the argument.

Syntax

LOGISTIC(*argument*)

Arguments

argument
is a numeric variable, constant, or expression that specifies the value of a numeric random variable. When *argument* is a missing or null value, the LOGISTIC function returns a missing or null value.

Details

The LOGISTIC function returns the logistic transformation of an argument. It is typically used to convert a log odds value to a value on the probability scale. The function is mathematically expressed by the following equation:

$$logistic=\frac{e^x}{1+e^x}$$

If the argument contains a missing value, then the LOGISTIC function returns a missing or null value.

Example

The following statement illustrates the LOGISTIC function:

```
proc cas;
   x = LOGISTIC(.5);
   print x;
   x = LOGISTIC(7.3);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.6224593312
   0.9993249173
```

LOWCASE Function

Converts all letters in a character expression to lowercase.

Alias: LOWER

Returned data type: STRING, VARCHAR

Syntax

LOWCASE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The LOWCASE function copies a character expression, converts all uppercase letters to lowercase letters, and returns the altered value as a result.

Comparisons

The UPCASE function converts all letters in an argument to uppercase letters. The LOWCASE function converts all letters in an argument to lowercase letters.

Example

The following statement illustrates the LOWCASE function:

```
proc cas;
   a=lowcase('INTRODUCTION');
   print a;
run;
```

The following output is printed to the SAS Log:

```
introduction
```

MAD Function

Returns the median absolute deviation from the median.

Returned data type: DOUBLE

Syntax

MAD(*expression-1*<, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value of which the median absolute deviation from the median is to be computed.

Data type DOUBLE

Details

If all arguments have missing or null values, the result is a missing or null value. Otherwise, the result is the median absolute deviation from the median of the nonmissing or non-null values. The formula for the median is the same as the one that is used in the UNIVARIATE procedure. For more information, see Base SAS Procedures Guide: Statistical Procedures.

Example

The following statement illustrates the MAD function:

```
proc cas;
  c=mad(2, 4, 1, 3, 5, 999999);
  print c;
run;
```

The following output is printed to the SAS Log:

```
   1.5
```

MARGRCLPRC Function

Calculates call prices for European options on stocks, based on the Margrabe model.

Returned data type: DOUBLE

Syntax

MARGRCLPRC(*X_1*, *t*, *X_2*, *sigma1*, *sigma2*, *rho12*)

Arguments

X_1
is a nonmissing, positive value that specifies the price of the first asset.

Requirement: Specify X_1 and X_2 in the same units.

Data type: DOUBLE

t
is a nonmissing value that specifies the time to expiration, in years.

Data type: DOUBLE

X_2
is a nonmissing, positive value that specifies the price of the second asset.

Requirement: Specify X_2 and X_1 in the same units.

Data type: DOUBLE

sigma1
is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type: DOUBLE

sigma2
is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type: DOUBLE

rho12
specifies the correlation between the first and second assets.

specifies the correlation between the first and second assets, $\rho_{x_1 x_2}$.

Range	between –1 and 1
Data type	DOUBLE

Details

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$\text{CALL} = X_1 N(d_1) - X_2 N(d_2)$$

Arguments

X_1
specifies the price of the first asset.

X_2
specifies the price of the second asset.

N
specifies the cumulative normal density function.

$$d_1 = \frac{\left(\ln\left(\frac{N_1}{N_2}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1,x_2}\sigma_{x_1}\sigma_{x_2}$$

The following arguments apply to the preceding equation:

t
specifies the time to expiration.

$\sigma_{x_1}^2$
specifies the variance of the first asset.

$\sigma_{x_2}^2$
specifies the variance of the second asset.

σ_{x_1}
specifies the volatility of the first asset.

σ_{x_2}
specifies the volatility of the second asset.

ρ_{x_1, x_2}
specifies the correlation between the first and second assets.

For the special case of *t*=0, the following equation is true:

$$\text{CALL} = \max((X_1 - X_2), 0)$$

Note: This function assumes that there are no dividends from the two assets.

Comparisons

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following statements illustrate the MARGRCLPRC function:

```
proc cas;
  a=margrclprc(15, .5, 13, .06, .05, 1);
  print a;
  b=margrclprc(2, .25, 1, .3, .2, 1);
  print b;
run;
```

The following output is printed to the SAS Log:

```
   2
   1
```

MARGRPTPRC Function

Calculates put prices for European options on stocks, based on the Margrabe model.

Returned data type: DOUBLE

Syntax

MARGRPTPRC(*X*$_1$, *t*, *X*$_2$, *sigma1*, *sigma2*, *rho12*)

Arguments

***X*$_1$**

is a nonmissing, positive value that specifies the price of the first asset.

Requirement Specify X_1 and X_2 in the same units.

Data type DOUBLE

t
is a nonmissing value that specifies the time to expiration, in years.

Data type DOUBLE

X_2
is a nonmissing, positive value that specifies the price of the second asset.

Requirement Specify X_2 and X_1 in the same units.

Data type DOUBLE

sigma1
is a nonmissing, positive fraction that specifies the volatility of the first asset.

Data type DOUBLE

sigma2
is a nonmissing, positive fraction that specifies the volatility of the second asset.

Data type DOUBLE

rho12
specifies the correlation between the first and second assets.

specifies the correlation between the first and second assets, $\rho_{x_1 y_1}$.

Range between –1 and 1

Data type DOUBLE

Details

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$\text{PUT} = X_2 N(pd_1) - X_1 N(pd_2)$$

Arguments

X_1
specifies the price of the first asset.

X_2
specifies the price of the second asset.

N
specifies the cumulative normal density function.

$$pd_1 = \frac{\left(\ln\left(\frac{N_1}{N_2}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$pd_2 = pd_1 - \sigma\sqrt{t}$$

$$\sigma^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2 - 2\rho_{x_1, x_2}\sigma_{x_1}\sigma_{x_2}$$

The following arguments apply to the preceding equation:

t
is a nonmissing value that specifies the time to expiration, in years.

$\sigma_{x_1}^2$
specifies the variance of the first asset.

$\sigma_{x_2}^2$
specifies the variance of the second asset.

σ_{x_1}
specifies the volatility of the first asset.

σ_{x_2}
specifies the volatility of the second asset.

ρ_{x_1, x_2}
specifies the correlation between the first and second assets.

For the special case of t=0, the following equation is true:

$$\text{PUT} = \max((X_2 - X_1), 0)$$

Note: This function assumes that there are no dividends from the two assets.

Comparisons

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. These functions return a scalar value.

Example

The following statements illustrate the MARGRPTPRC function:

```
proc cas;
  a=margrptprc(2, .25, 3, .06, .2, 1);
  print a;
  b=margrptprc(3, .25, 4, .05, .3, 1);
  print b;
run;
```

The following output is printed to the SAS Log:

```
1.0000000001
1.0015762491
```

MAX Function

Returns the largest value from a list of arguments.

Returned data type: BIGINT, DECIMAL, DOUBLE

Syntax

MAX(*expression-1, expression-2* <, ...*expression-n*>)

Arguments

expression
is any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: BIGINT, DECIMAL, DOUBLE

Details

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Comparisons

The MAX function returns the largest value from a list of arguments. The MAX operator (<>) returns the largest of two operands.

The MAX function returns a null or missing value only if all arguments are null or missing. The MAX operator (<>) returns a null or missing value only if both operands are null or missing. In this case, it returns the value of the operand that is higher in the sort order for null or missing values.

Example

The following statements illustrate the MAX function:

```
proc cas;
   x=max(8,3);
   print x;
   x=max(2, 6, .);
   print x;
   x=max(2, -3, 1, -1);
   print x;
   x=max(3, ., -3);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   8
   6
   2
   3
```

MDY Function

Returns a SAS date value from month, day, and year values.

Returned data type: DOUBLE

Syntax

MDY(*month*, *day*, *year*)

Arguments

month
specifies a numeric expression that represents a whole number from 1 through 12.

Data type DOUBLE

day
specifies a numeric expression that represents a whole number from 1 through 31.

Data type DOUBLE

year
specifies a numeric expression that represents a two-digit or four-digit year. The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

Example

The following statements illustrate the MDY function:

```
proc cas;
  mn=8;
  dy=27;
  yr=12;
  birthday=mdy(mn, dy, yr);
  print "birthday=" birthday;
run;
```

The following output is printed to the SAS Log:

```
   birthday=19232
```

```
proc cas;
  mn=7;
  dy=11;
  yr=12;
  anniversary=mdy(mn, dy, yr);
  print "anniversary=" anniversary;
run;
```

The following output is printed to the SAS Log:

```
   anniversary=19185
```

Note: CASL outputs the birthday date as a SAS date value.

MEAN Function

Returns the arithmetic mean (average) of the non-null or nonmissing arguments.

Returned data type: DOUBLE

Syntax

MEAN(*expression-1*<, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.
Data type	DOUBLE

Comparisons

The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, whereas the MEAN function returns the arithmetic mean (average).

Example

The following statements illustrate the MEAN function:

```
proc cas;
   a=mean(2,.,.,6);
   print a;
   a=mean(1,2,3,2);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   4
   2
```

MEDIAN Function

Returns the median value.

Category:	CAS
Returned data type:	DOUBLE

Syntax

MEDIAN(*expression-1*<, ...*expression-n* >)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The MEDIAN function returns the median of the nonmissing or nonnull values. If all arguments have missing or null values, the result is a missing or null value.

Note: The formula that is used in the MEDIAN function is the same as the formula that is used in PROC UNIVARIATE in Base SAS Procedures Guide: Statistical Procedures.

Comparisons

The MEDIAN function returns the median of nonmissing or nonnull values, whereas the MEAN function returns the arithmetic mean (average).

Example

The following statements illustrate the MEDIAN function:

```
proc cas;
   x=median(2, 4, 1, 3);
   print x;
   y=median(5, 8, 0, 3, 4);
   print y;
run;
```

The following output is printed to the SAS Log:

```
   2.5
   4
```

MIN Function

Returns the smallest value.

Returned data type: BIGINT, DECIMAL, DOUBLE

Syntax

MIN(*expression-1, expression-2* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type BIGINT, DECIMAL, DOUBLE

Details

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Comparisons

The MIN function returns the smallest value from a list of values. The MIN operator (><) returns the smallest value of two operands.

The MIN function returns a null or missing value only if all arguments are null or missing. The MIN operator returns a null or missing value only if either operand is null or missing. In this case, it returns the value of the operand that is lower in the sort order for null or missing values.

Example

The following statements illustrate the MIN function:

```
proc cas;
   a=min(2,.,6);
   print a;
   a=min(2,-3,1,-1);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   2
   -3
```

MINUTE Function

Returns the minute from a SAS time or datetime value.

Returned data type: DOUBLE

Syntax

MINUTE(*time* | *datetime*)

Arguments

time
specifies any valid expression that represents a SAS time value.

Data type DOUBLE

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

Details

The MINUTE function returns a whole number that represents a specific minute of the hour. MINUTE always returns a positive number in the range of 0 through 59. Null or missing values are ignored.

Example

The following statement illustrates the MINUTE function:

```
proc cas;
   a=minute(time());
   print a;
run;
```

The following output is printed to the SAS Log:

```
   55
```

MISSING Function

Returns a number that indicates whether the argument contains a missing value.

Returned data type: INTEGER

Syntax

MISSING(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a value.

Data type All data types

Details

- The MISSING function checks if a value is a null or missing value and returns a numeric result. If the argument does not contain a missing value, SAS returns a value of 0. If the argument contains a missing value, SAS returns a value of 1.
- In SAS mode, a blank-filled character value is defined to be the SAS missing value. In ANSI mode, a blank-filled character value is defined as nonmissing and non-null.
- In SAS mode, a DOUBLE value could be a SAS missing value (., .A through .Z). The other numeric types do not support SAS missing values.
- The MISSING function returns a 1 if a package instance does not exist. That is, the package variable is a missing package reference. The MISSING function returns a 0 if the package variable references a package instance.

Comparisons

The MISSING function can have only one argument. The NMISS function requires numeric arguments and returns the number of missing values in the list of arguments.

Example: Using the MISSING Function

The following example illustrates the MISSING function.

```
proc cas;
  a[1]=2;
  a[2]=4;
  a[3]=.;
  do i=1 to 3;
    if missing (a[i]) then put 'Missing';
      else print 'Not Missing';
      end;
    end;
run;
```

The following lines are written to the SAS log.

```
Not Missing
Not Missing
Missing
```

MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.

Returned data type: DOUBLE

Syntax

MOD(*dividend-expression*, *divisor-expression*)

Arguments

dividend-expression
specifies a dividend that is any valid expression that evaluates to a numeric value.

Data type DOUBLE

divisor-expression
specifies a divisor that is any valid expression that evaluates to a numeric value.

Restriction *divisor-expression* cannot be 0

Data type DOUBLE

Details

The MOD function returns the remainder from the division of *dividend-expression* by *divisor-expression*. When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MOD function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representations.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. In this case, the following occurs.

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more. In this case, SAS also writes an error message to the log.

Comparisons

Here are some comparisons between the MOD and MODZ functions:

- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- The MODZ function performs no fuzzing.
- Both the MOD and MODZ functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

The following statements illustrate the MOD function:

```
proc cas;
   a=mod(10,3);
   print a;
   a=mod(.35,-.1);
   print a;
   a=mod(17,3);
   print a;
   a=mod(.3,-.9);
   print a;
run;
```

The following output is printed to the SAS Log:

```
1
0.05
2
0.3
```

MODZ Function

Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.

Returned data type: DOUBLE

Syntax

MODZ(*dividend-expression*, *divisor-expression*)

Arguments

dividend-expression
specifies a dividend that is any valid expression that evaluates to a numeric value.

Data type DOUBLE

divisor-expression
specifies a divisor that is any valid expression that evaluates to a numeric value.

Restriction *divisor-expression* cannot be 0

Data type DOUBLE

Details

The MODZ function returns the remainder from the division of *dividend-expression* by *divisor-expression*. When the result is nonzero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MODZ function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representation.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. For example, when you use exact arithmetic and the result is zero, MODZ might return a very small positive value or a value slightly less than the second argument.

Comparisons

Here are some comparisons between the MODZ and MOD functions:

- The MODZ function performs no fuzzing.
- The MOD function performs fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- Both the MODZ and MOD functions return a null or missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

The following statements illustrate the differences between the MOD and MODZ function:

```
proc cas;
   a=mod(10,3);
   b=modz(10,3);
   print "a=" a "b=" b;
run;
```

The following output is printed to the SAS Log:

```
   a=1 b=1
```

MONTH Function

Returns a number that represents the month from a SAS date value.

Returned data type: DOUBLE

Syntax

MONTH(*date*)

Arguments

date

specifies any valid expression that represents a SAS date value.

Range 1–12

Data type DOUBLE

Example

```
proc cas;
   theDate=inputn('16JUL1969','DATE9.');
   theDay=day(theDate);
   theMonth=Month(theDate);
   theYear=year(theDate);
   print(NOTE) "Date=" putn(theDate,'date9.');
   print(NOTE) "Day=" theDay ' Month=' theMonth 'Year=' theYear;
run;
```

The program writes the following output to the log:

```
NOTE: Date=16JUL1969
NOTE: Day=16  Month=7 Year=1969
```

MORT Function

Returns amortization parameters.

Returned data type: DOUBLE

Syntax

MORT(*a*, *p*, *r*, *n*)

Arguments

a

specifies any valid expression that evaluates to the initial amount.

Data type DOUBLE

p

specifies any valid expression that evaluates to the periodic payment.

Data type DOUBLE

r

specifies any valid expression that evaluates to the periodic interest rate that is expressed as a fraction.

Data type DOUBLE

n

specifies any valid expression that evaluates to the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

Details

Calculating Results

The MORT function returns the missing argument in the list of four arguments from an amortization calculation with a fixed interest rate that is compounded each period. The arguments are related by the following equation:

$$p = \frac{ar(1+r)^n}{(1+r)^n - 1}$$

One missing argument must be provided. The value is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Restrictions in Calculating Results

The MORT function returns an invalid argument note to the SAS log and sets _ERROR_ to 1 if one of the following argument combinations is true:

- rate < –1 or n < 0
- principal <= 0 or payment <= 0 or n <= 0
- principal <= 0 or payment <= 0 or rate <= –1
- principal * rate > payment
- principal > payment * n

Example

In the following example, an amount of $50,000 is borrowed for 30 years at an annual interest rate of 10% compounded monthly.

```
proc cas;
  payment=mort(50000, . , .10/12, 30*12);
  print payment;
run;
```

The value that is returned is 438.79 (rounded). The second argument is set to missing, which indicates that the periodic payment is to be calculated. The 10% nominal annual rate has been converted to a monthly rate of 0.10/12. The rate is the fractional (not the percentage) interest rate per compounding period. The 30 years are converted to 360 months.

N Function

Returns the number of non-null or nonmissing numeric values.

Returned data type: INTEGER

Syntax

N(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one argument is required.

Data type: DECIMAL, DOUBLE, NUMERIC

Details

Null values are converted to missing values and are counted as missing values.

Comparisons

The N function counts non-null and nonmissing values, whereas the NMISS function counts missing values. The N function requires numeric arguments.

Example

The following statements illustrate the N function:

```
proc cas;
   a=n(1,0,.,2,5,.);
   print a;
```

```
   a=n(1,2);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   4
   2
```

NETPV Function

Returns the net present value as a percent.

Returned data type: DOUBLE

Syntax

NETPV(*r*, *freq*, *c0, c1, ..., cn*)

Arguments

r

is numeric, the interest rate over a specified base period of time expressed as a fraction.

Range *r* >= 0

Data type DOUBLE

freq

is numeric, the number of payments during the base period of time that is specified with the rate *r*.

Range *freq* > 0

Data type DOUBLE

Note The case *freq* = 0 is a flag to allow continuous discounting.

c0, c1,..., cn

are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ...n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The *c0* argument and the *c1* argument are required.

Data type DOUBLE

Details

The NETPV function returns the net present value at time 0 for the set of cash payments *c0*,*c1*, ...,*cn*, with a rate *r* over a specified base period of time. The argument *freq*>0 describes the number of payments that occur over the specified base period of time.

The net present value is given by the equation:

$$\mathrm{NETPV}(r, freq, c_0, c_1, ..., c_n) = \sum_{i=0}^{n} c_i x^i$$

The following relationship applies to the preceding equation:

$$x = \begin{cases} \frac{1}{(1+r)^{(1/freq)}} & freq > 0 \\ \varepsilon^{-r} & freq = 0 \end{cases}$$

Missing values in the payments are treated as 0 values. When *freq*>0, the rate *r* is the effective rate over the specified base period. To compute with a quarterly rate (the base period is three months) of 4% with monthly cash payments, set *freq* to 3 and set *r* to .04.

If *freq* is 0, continuous discounting is assumed. The base period is the time interval between two consecutive payments, and the rate *r* is a nominal rate.

To compute with a nominal annual interest rate of 11% discounted continuously with monthly payments, set *freq* to 0 and set *r* to .11/12.

Example

For an initial investment of $500 that returns biannual payments of $200, $300, and $400 over the succeeding 6 years and an annual discount rate of 10%, the net present value of the investment can be expressed as follows:

```
proc cas;
  value=netpv(.10, .5, -500, 200, 300, 400);
  print value;
run;
```

The value that is returned is 95.982864829379.

See Also

Functions:

- "NPV Function" in *SAS DS2 Language Reference*

NMISS Function

Returns the number of null and SAS missing numeric values.

Returned data type: INTEGER

Syntax

NMISS(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one argument is required.

Data type: DECIMAL, DOUBLE, NUMERIC

Details

Null values are converted to SAS missing values and are counted as missing values.

Comparisons

The NMISS function returns the number of null or SAS missing values, whereas the N function returns the number of non-null and nonmissing values. NMISS requires numeric values and works with multiple numeric values, whereas MISSING works with only one value that can be either numeric or character.

Example

The following statements illustrate the NMISS function:

```
proc cas;
   a=nmiss(1,0,.,2,5,.);
   print a;
   a=nmiss(1,0);
   print a;
run;
```

The following output is printed to the SAS Log:

```
2
0
```

NOMRATE Function

Returns the nominal annual interest rate.

Returned data type: DOUBLE

Syntax

NOMRATE(*compounding-interval*, *rate*)

Arguments

compounding-interval
is a SAS interval. This value represents how often the returned value is compounded.

Data type CHAR

rate
is numeric. *Rate* is the effective annual interest rate (expressed as a percentage) that is compounded at each interval.

Data type DOUBLE

Details

The NOMRATE function returns the nominal annual interest rate. NOMRATE computes the nominal annual interest rate that corresponds to an effective annual interest rate.

The following details apply to the NOMRATE function:

- The values for rates must be at least –99.
- In considering an effective interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by NOMRATE equals $\log_e(1+[rate/100])$.

  If *compounding-interval* is not 'CONTINUOUS', and *m* intervals occur in a year, the value that is returned by NOMRATE equals the following:

$$m\left((1+\frac{rate}{100})^{\frac{1}{m}}-1\right)$$

- The following values are valid for *compounding-interval*:
  - 'CONTINUOUS'
  - 'DAY'
  - 'SEMIMONTH'
  - 'MONTH'
  - 'QUARTER'
  - 'SEMIYEAR'
  - 'YEAR'
- If the interval is 'DAY', then *m*=365.

Example

- If an effective rate is 10% when compounded monthly, the corresponding nominal rate can be expressed as follows:

```
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);
```

NOTALNUM Function

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

Returned data type: DOUBLE

Syntax

NOTALNUM('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The results of the NOTALNUM function depend directly on the translation table that is in effect and indirectly on the system options.

The NOTALNUM function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase letter. If such a character is found, NOTALNUM returns the position in the string of that character. If no such character is found, NOTALNUM returns a value of 0.

If you use only one argument, NOTALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALNUM function searches a character string for a non-alphanumeric character. The ANYALNUM function searches a character string for an alphanumeric character.

Example

The following example uses the NOTALNUM function to search a string from left to right for non-alphanumeric characters.

```
proc cas;
      string='Next = Last + 1;';
      j=1;
      do until(j=0);
         j=notalnum(string, j+1);
         if j=0 then put 'The end';
         else do;
            c=substr(string, j, 1);
            print  "j=" j "c=" c;
         end;
      end;
run;
```

SAS writes the following output to the log:

```
j=5 c==
j=10 c=+
j=12 c=;
The End
```

NOTALPHA Function

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

Returned data type: DOUBLE

Syntax

NOTALPHA('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTALPHA function searches a string for the first occurrence of any character that is not an uppercase or lowercase letter. If such a character is found, NOTALPHA returns the position in the string of that character. If no such character is found, NOTALPHA returns a value of 0.

If you use only one argument, NOTALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALPHA function searches a character string for a nonalphabetic character. The ANYALPHA function searches a character string for an alphabetic character.

Example: Searching a String for Nonalphabetic Characters

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

```
proc cas;
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notalpha(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j "c=" 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=;
The end
```

NOTCNTRL Function

Searches a character string for a character that is not a control character, and returns the first character position at which that character is found.

Returned data type: DOUBLE

Syntax

NOTCNTRL('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTCNTRL function searches a string for the first occurrence of a character that is not a control character. If such a character is found, NOTCNTRL returns the

position in the string of that character. If no such character is found, NOTCNTRL returns a value of 0.

If you use only one argument, NOTCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTCNTRL returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTCNTRL function searches a character string for a character that is not a control character. The ANYCNTRL function searches a character string for a control character.

Example

```
proc cas;
   strings={'01'x||'09'x||'0D'x||'0A'x||'43'x||'41'x||'53'x||'4C'x
           ,'B 1AB '
           ,'01'x||'02'x||'03'x||'04'x||'09'x||'09'x||'07'x||'08'x||'0D
'x||'0A'x};
   do thisString over strings;
      if notcntrl(thisString)>0 then
         print "Non-control character found in '" thisString "' at "
notcntrl(thisString);
         else print "thisString contains only control characters. "
thisString;
   end;
run;
```

The program writes the following output to the log:

```
Non-control character found in 'CASL' at 5
Non-control character found in 'B 1AB ' at 1
thisString contains only control characters.
```

NOTDIGIT Function

Searches a character string for any character that is not a digit, and returns the first character position at which that character is found.

Returned data type: DOUBLE

Syntax

NOTDIGIT('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTDIGIT function searches a string for the first occurrence of any character that is not a digit. If such a character is found, NOTDIGIT returns the position in the string of that character. If no such character is found, NOTDIGIT returns a value of 0.

If you use only one argument, NOTDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTDIGIT function searches a character string for any character that is not a digit. The ANYDIGIT function searches a character string for a digit.

Example

The following example uses the NOTDIGIT function to search for a character that is not a digit.

```
proc cas;
    string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notdigit(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" 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=16 c=E
j=18 c=;
The end
```

NOTFIRST Function

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

Returned data type: DOUBLE

Syntax

NOTFIRST('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTFIRST function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are any except the underscore (_) and uppercase or lowercase English letters. If such a character is found, NOTFIRST returns the position in the string of that character. If no such character is found, NOTFIRST returns a value of 0.

If you use only one argument, NOTFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.

- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Example

The following 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.

```
proc cas;
     string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notfirst(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" 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=;
The end
```

NOTGRAPH Function

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

Returned data type: DOUBLE

Syntax

NOTGRAPH('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTGRAPH function searches a string for the first occurrence of a non-graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, NOTGRAPH returns the position in the string of that character. If no such character is found, NOTGRAPH returns a value of 0.

If you use only one argument, NOTGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTGRAPH function searches a character string for a non-graphical character. The ANYGRAPH function searches a character string for a graphical character.

Example: Searching a String for Non-Graphical Characters

The following example uses the NOTGRAPH function to search a string for a non-graphical character.

```
proc cas;
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notgraph(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" c;
         end;
      end;
run;
```

SAS writes the following output to the log:

```
j=5 c=
j=7 c=
j=11 c=
j=13 c=
The end
```

NOTLOWER Function

Searches a character string for a character that is not a lowercase letter, and returns the first character position at which that character is found.

Returned data type: DOUBLE

Syntax

NOTLOWER('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTLOWER function searches a string for the first occurrence of any character that is not a lowercase letter. If such a character is found, NOTLOWER returns the position in the string of that character. If no such character is found, NOTLOWER returns a value of 0.

If you use only one argument, NOTLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTLOWER function searches a character string for a character that is not a lowercase letter. The ANYLOWER function searches a character string for a lowercase letter.

Example

The following example uses the NOTLOWER function to search a string for any character that is not a lowercase letter.

```
proc cas;
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notlower(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" 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=16 c=E
j=17 c=3
j=18 c=;
The end
```

NOTNAME Function

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

Returned data type: DOUBLE

Syntax

NOTNAME('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTNAME function does not depend on the TRANTAB, ENCODING, or LOCALE system options.

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. These characters are any except underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, NOTNAME returns the position in the string of that character. If no such character is found, NOTNAME returns a value of 0.

If you use only one argument, NOTNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7.

Example

The following example uses the NOTNAME function to search a string for any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

```
proc cas;
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notname(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" 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=;
The end
```

NOTPRINT Function

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

Returned data type: DOUBLE

Syntax

NOTPRINT('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type: CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTPRINT function searches a string for the first occurrence of a non-printable character. If such a character is found, NOTPRINT returns the position in the string of that character. If no such character is found, NOTPRINT returns a value of 0.

If you use only one argument, NOTPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTPRINT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTPRINT function searches a character string for a non-printable character. The ANYPRINT function searches a character string for a printable character.

Example

```
proc cas;
   strings={'SAS'||'09'x||'0D'x||'0A'x
           ,'B 1AB '
           ,'43'x||'41'x||'53'x||'4C'x};
   do thisString over strings;
      if notprint(thisString)>0 then
         print "Non-printable character found in '" thisString "' at "
notprint(thisString);
```

```
            else print "thisString contains only printable characters. "
   thisString;
      end;
   run;
```

The program writes the following output to the log:

```
Non-printable character found in 'SAS' at 4
thisString contains only printable characters. B 1AB
thisString contains only printable characters. CASL
```

NOTPUNCT Function

Searches a character string for a character that is not a punctuation character, and returns the first character position at which that character is found.

Returned data type: DOUBLE

Syntax

NOTPUNCT('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTPUNCT function searches a string for the first occurrence of a character that is not a punctuation character. If such a character is found, NOTPUNCT returns the position in the string of that character. If no such character is found, NOTPUNCT returns a value of 0.

If you use only one argument, NOTPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTPUNCT function searches a character string for a character that is not a punctuation character. The ANYPUNCT function searches a character string for a punctuation character.

Example: 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.

```
proc cas;
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notpunct(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" 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=9 c=n
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
The end
```

NOTSPACE Function

Searches a character string for a character that is not a whitespace character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first character position at which that character is found.

Returned data type: DOUBLE

Syntax

NOTSPACE('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTSPACE function searches a string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, NOTSPACE returns the position in the string of that character. If no such character is found, NOTSPACE returns a value of 0.

If you use only one argument, NOTSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Example: 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.

```
proc cas;
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notspace(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" 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=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c=+
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
The end
```

NOTUPPER Function

Searches a character string for a character that is not an uppercase letter, and returns the first character position at which that character is found.

Returned data type: DOUBLE

Syntax

NOTUPPER('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTUPPER function searches a string for the first occurrence of a character that is not an uppercase letter. If such a character is found, NOTUPPER returns the position in the string of that character. If no such character is found, NOTUPPER returns a value of 0.

If you use only one argument, NOTUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTUPPER function searches a character string for a character that is not an uppercase letter. The ANYUPPER function searches a character string for an uppercase letter.

Example

The following example uses the NOTUPPER function to search a string for any character that is not an uppercase letter.

```
proc cas;
      string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notupper(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" 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=;
The end
```

NOTXDIGIT Function

Searches a character string for a character that is not a hexadecimal character, and returns the first character position at which that character is found.

Returned data type: DOUBLE

Syntax

NOTXDIGIT('*expression*'<, *start*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, VARCHAR, NVARCHAR

start
specifies any valid expression that evaluates or can be coerced to a numeric value and specifies the character position at which the search should start and the direction in which to search.

Data type DOUBLE

Details

The NOTXDIGIT function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, NOTXDIGIT returns the position in the string of that character. If no such character is found, NOTXDIGIT returns a value of 0.

If you use only one argument, NOTXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character. The ANYXDIGIT function searches a character string for a character that is a hexadecimal character.

Example

The following example uses the NOTXDIGIT function to search a string for a character that is not a hexadecimal character.

```
proc cas;
     string='Next = _n_ + 12E3;';
      j=1;
      do until(j=0);
         j=notxdigit(string, j+1);
         if j=0 then print 'The end';
         else do;
            c=substr(string, j, 1);
            print "j=" j;
            print "c=" c;
         end;
      end;
run;
```

SAS writes the following output to the log:

```
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=18 c=;
The end
```

NPV Function

Returns the net present value with the rate expressed as a percentage.

Returned data type: DOUBLE

Syntax

NPV(*r*, *freq*, *c0*, *c1*, ..., *cn*)

Arguments

r

is numeric, the interest rate over a specified base period of time expressed as a percentage.

Data type DOUBLE

freq

is numeric, the number of payments during the base period of time specified with the rate *r*.

Range *freq* > 0

Data type DOUBLE

Note The case *freq* = 0 is a flag to allow continuous discounting.

c0, c1, ..., cn

are numeric cash flows that represent cash outlays (payments) or cash inflows (income) occurring at times 0, 1, ...n. These cash flows are assumed to be equally spaced, beginning-of-period values. Negative values represent payments, positive values represent income, and values of 0 represent no cash flow at a given time. The *c0* argument and the *c1* argument are required.

Data type DOUBLE

Comparisons

The NPV function is identical to NETPV, except that the *r* argument is provided as a percentage.

Example

This example uses the NPV function to return the net present value with rate expressed as a percentage.

```
proc cas;
   a=NPV(10,1,100,100,100);
   print a;
run;
```

```
273.55371901
```

NWKDOM Function

Returns the date for the *n*th occurrence of a weekday for the specified month and year.

Returned data type: DOUBLE

Syntax

NWKDOM(*n*, *weekday*, *month*, *year*)

Arguments

n

specifies the numeric week of the month that contains the specified day.

Range 1–5

Data type DOUBLE

Tip *N*=5 indicates that the specified day occurs in the last week of that month. Sometimes *n*=4 and *n*=5 produce the same results.

weekday

specifies the number that corresponds to the day of the week.

Range 1–7

Data type DOUBLE

Tip Sunday is considered the first day of the week and has a *weekday* value of 1.

month
specifies the number that corresponds to the month of the year.

Range 1–12

Data type DOUBLE

year
specifies a four-digit calendar year.

Data type DOUBLE

Details

The NWKDOM function returns a SAS date value for the *n*th weekday of the month and year that you specify. Use any valid SAS date format, such as the DATE9. format, to display a calendar date. You can specify *n*=5 for the last occurrence of a particular weekday in the month.

Sometimes *n*=5 and *n*=4 produce the same result. These results occur when there are only four occurrences of the requested weekday in the month. For example, if the month of January begins on a Sunday, there 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);`

Example: 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.

```
proc cas;
   /* Return the date of the third Monday in May 2012. */
   a=nwkdom(3, 2, 5, 2012);
      /* Return the date of the fourth Wednesday in November 2012. */
   b=nwkdom(4, 4, 11, 2012);
      /* Return the date of the fourth Saturday in November 2012. */
   c=nwkdom(4, 7, 11, 2012);
      /* Return the date of the first Sunday in January 2013. */
   d=nwkdom(1, 1, 1, 2013);
      /* Return the date of the second Tuesday in September 2012. */
   e=nwkdom(2, 3, 9, 2012);
      /* Return the date of the fifth Thursday in December 2012. */
   f=nwkdom(5, 5, 12, 2012);
   print "a=" a;
   print "b=" b;
   print "c=" c;
   print "d=" d;
   print "e=" e;
   print "f=" f;
end;
run;
```

SAS writes the following output to the log:

```
a=19134
b=19325
c=19321
d=19364
e=19247
f=19354
```

ORDINAL Function

Orders a list of values, and returns a value that is based on a position in the list.

Returned data type: DOUBLE

Syntax

ORDINAL(*position*, *expression-1, expression-2* <, ...*expression-n*>)

Arguments

position
specifies a whole number that is less than or equal to the number of elements in the list of arguments.

Requirement *position* must be a positive number.

Data type DOUBLE

expression
specifies any valid expression that evaluates to a numeric value.

Requirement At least two arguments are required.

Data type DOUBLE

Details

The ORDINAL function sorts the list and returns the argument in the list that is specified by *position*. Missing values are sorted low and are placed before any numeric values.

Comparisons

The ORDINAL function counts both null, missing, non-null, and nonmissing values, whereas the SMALLEST function counts only non-null and nonmissing values.

Example

The following statement illustrates the ORDINAL function:

```
proc cas;
   a=ordinal(4,1,.,2,3,-4,5,6,7);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   2
```

PCTL Function

Returns the percentile that corresponds to the percentage.

Returned data type: DOUBLE

Syntax

PCTL<*n*>(*percentage*, *expression*<, ...*expression*>)

Arguments

n
is a digit from 1 to 5 that specifies the definition of the percentile to be computed.

Default: definition 5

Data type: DOUBLE

See: QNTLDEF= (alias PCTLDEF=) descriptions in the table “Methods for Computing Quantile Statistics” in the *Base SAS Procedures Guide*.

percentage
specifies the percentile to be computed.

Data type: DOUBLE

Tips: *percentage* is numeric where, 0≤*percentage*≤100.

percentage is numeric where, 0 ≤ *percentage* ≤ 100.

expression
specifies any valid expression that evaluates to a numeric value, whose value is computed in the percentile calculation.

Data type: DOUBLE

Details

The PCTL function returns the percentile of the non-null or nonmissing values corresponding to the percentage. If *percentage* is null or missing, less than zero, or greater than 100, the PCTL function generates an error message.

Example

The following statements illustrate the PCTL function:

```
proc cas;
   lower_quartile=PCTL(25,2,4,1,3);
   print "lower_quartile=" lower_quartile;
```

```
   percentile_def2=PCTL2(25,2,4,1,3);
   print "percentile_def2=" percentile_def2;
   lower_tertile=PCTL(100/3,2,4,1,3);
   print "lower_tertile=" lower_tertile;
   percentile_def3=PCTL3(100/3,2,4,1,3);
   print "percentile_def3=" percentile_def3;
   median=PCTL(50,2,4,1,3);
   print "median=" median;
   upper_tertile=PCTL(200/3,2,4,1,3);
   print "upper_tertile=" upper_tertile;
   upper_quartile=PCTL(75,2,4,1,3);
   print "upper_quartile=" upper_quartile;
run;
```

The following output is printed to the SAS Log:

```
   lower_quartile=1.5
   percentile_def2=1
   lower_tertile=2
   percentile_def3=2
   median=2.5
   upper_tertile=3
   upper_quartile=3.5
```

PERM Function

Computes the number of permutations of *n* items that are taken *r* at a time.

Returned data type: DOUBLE

Syntax

PERM(*n*<, *r*>)

Arguments

n

specifies any valid expression that represents the total number of elements from which the sample is chosen.

Data type DOUBLE

r

specifies any valid expression that represents the number of chosen elements.

Restriction $r \leq n$

Data type DOUBLE

Note If *r* is omitted, the function returns the factorial of *n*.

Details

The mathematical representation of the PERM function is given by the following equation:

$$PERM(n, r) = \frac{n!}{(n-r)!}$$

with $n \geq 0$, $r \geq 0$, and $n \geq r$.

If the expression cannot be computed, a missing value is returned. For moderately large values, it is sometimes not possible to compute the PERM function.

Example

The following statements illustrate the PERM function:

```
proc cas;
   x=perm(5, 1);
   print x;
   x=perm(5);
   print x;
   x=perm(5, 2);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   5
   120
   20
```

PMT Function

Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.

Returned data type: DOUBLE

Syntax

PMT(*rate*, *number-of-periods*, *principal-amount*<, *future-amount*><, *type*>)

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a null or missing value is specified.

Data type DOUBLE

future-amount
specifies the future amount. *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a missing value is specified.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a null or missing value is specified.

Data type DOUBLE

Example

- The monthly payment for a $10,000 loan with a nominal annual interest rate of 8% and 10 end-of-month payments can be computed in the following ways:

```
proc cas;
   Payment1 = PMT(0.08/12., 10, 10000, 0, 0);
   print Payment1
run;
```

```
proc cas;
   Payment1 = PMT(0.08/12., 10, 10000);
   print Payment1;
run;
```

These computations return a value of 1037.03208935915.

- If the same loan has beginning-of-period payments, then payment can be computed as follows:

```
proc cas;
   Payment2 = PMT(0.08/12., 10, 10000, 0, 1);
   print Payment2;
run;
```

  This computation returns a value of 1030.16432717796.

- The payment for a $5,000 loan earning a 12% nominal annual interest rate, that is to be paid back in five monthly payments, is computed as follows:

```
proc cas;
   Payment3 = PMT(.01/12., 5, 5000);
   print Payment3;
run;
```

  This computation returns a value of 1002.50138831008.

- The payment for monthly periodic savings that accrue more than 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:

```
proc cas;
   Payment4 = PMT(0.06/12., 216, 0, 50000, 0);
   print Payment4;
run;
```

  This computation returns a value of -129.081160867993.

POISSON Function

Returns the probability from a Poisson distribution.

Returned data type: DOUBLE

Syntax

POISSON(*m*, *n*)

Arguments

m

specifies any valid expression that evaluates to a numeric mean parameter.

Range $m \geq 0$

Data type DOUBLE

n

specifies any valid expression that evaluates to a random variable.

Range $n \geq 0$

Data type DOUBLE

Details

The POISSON function returns the probability that an observation from a Poisson distribution, with mean *m*, is less than or equal to *n*. To compute the probability that an observation is equal to a given value, *n*, compute the difference of two probabilities from the Poisson distribution for *n* and *n*−1.

Example

The following statement illustrates the POISSON function:

```
x=poisson(1, 2);
```

The following output is printed to the SAS Log:

```
0.9196986029
```

PPMT Function

Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.

Returned data type: DOUBLE

Syntax

PPMT(*rate*, *period*, *number-of-periods*, *principal-amount*<, *future-amount*><, *type*>)

Arguments

rate
specifies the interest rate per payment period.

Data type DOUBLE

period
specifies the payment period for which the principal payment is computed.

Requirement *Period* must be a whole number that is less than or equal to the value of *number-of-periods*

Data type DOUBLE

number-of-periods
specifies the number of payment periods.

Requirement *Number-of-periods* must be a positive whole number.

Data type DOUBLE

principal-amount
specifies the principal amount of the loan. Zero is assumed if a null or missing value is specified.

Data type DOUBLE

future-amount
specifies the future amount. *Future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a null or missing value is specified.

Data type DOUBLE

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a null or missing value is specified.

Data type DOUBLE

Example

- The principal payment amount of the first monthly periodic payment for a 2-year, $2,000 loan with a nominal annual interest rate of 10%, is computed as follows:

```
proc cas;
   PrincipalPayment = PPMT(0.1/12., 1, 24, 2000);
   print PrincipalPayment;
run;
```

  This computation returns a value of 75.6231860083663.

- The principal payment for a 3-year, $20,000 loan with beginning-of-month payments is computed as follows:

```
proc cas;
   PrincipalPayment2 = PPMT(0.1/12., 1, 36, 20000, 0, 1);
   print PrincipalPayment;
run;
```

This computation returns a value of 640.010324505867 as the principal that was paid with the first payment.

- The principal payment of an end-of-month payment loan with an outstanding balance of $5,000 at the end of three years, is computed as follows:

```
proc cas;
   PrincipalPayment3 = PPMT(0.1/12., 1, 36, 20000, 5000, 0);
   print PrincipalPayment;
run;
```

This computation returns a value of 359.007807907562 as the principal that was paid with the first payment.

PROBBETA Function

Returns the probability from a beta distribution.

Returned data type: DOUBLE

Syntax

PROBBETA(*x*, *a*, *b*)

Arguments

x

is a numeric random variable.

Range $0 \leq x \leq 1$

Data type DOUBLE

a

is a numeric shape parameter.

Range $a > 0$

Data type DOUBLE

b

is a numeric shape parameter.

Range $b > 0$

Data type DOUBLE

Details

The PROBBETA function returns the probability that an observation from a beta distribution, with shape parameters *a* and *b*, is less than or equal to *x*.

Example

The following statement illustrates the PROBBETA function:

```
proc cas;
   x=probbeta(.2,3,4);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.09888
```

PROBBNML Function

Returns the probability from a binomial distribution.

Returned data type: DOUBLE

Syntax

PROBBNML(*p*, *n*, *m*)

Arguments

p

is a numeric probability of success parameter.

Range $0 \leq p \leq 1$

Data type DOUBLE

n

is the number of independent Bernoulli trials parameter. This argument must be a whole number.

Range $n > 0$

Data type DOUBLE

m
is the number of successes random variable. This argument must be a whole number.

Range $0 \le m \le n$

Data type DOUBLE

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success *p*, number of trials *n*, and number of successes *m*, is less than or equal to *m*. To compute the probability that an observation is equal to a given value *m*, compute the difference of two probabilities from the binomial distribution for *m* and *m*−1 successes.

Example

The following statement illustrates the PROBBNML function:

```
proc cas;
   x=probbnml(0.5,10,4);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.376953125
```

PROBCHI Function

Returns the probability from a chi-square distribution.

Returned data type: DOUBLE

Syntax

PROBCHI(*x*, *df*<, *nc*>)

Arguments

x
is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

nc
is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Data type DOUBLE

Details

The PROBCHI function returns the probability that an observation from a chi-square distribution, with degrees of freedom *df* and noncentrality parameter *nc*, is less than or equal to *x*. This function accepts a noninteger degrees of freedom parameter *df*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central chi-square distribution.

Example

The following statement illustrates the PROBCHI function:

```
proc cas;
   x=probchi(11.264,11);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.5785813293
```

PROBF Function

Returns the probability from an *F* distribution.

Returned data type: DOUBLE

Syntax

PROBF(*x*, *ndf*, *ddf*<, *nc*>)

Arguments

x
is a numeric random variable.

Range *x* ≥ 0

Data type DOUBLE

ndf
is a numeric numerator degrees of freedom parameter.

Range *ndf* > 0

Data type DOUBLE

ddf
is a numeric denominator degrees of freedom parameter.

Range *ddf* > 0

Data type DOUBLE

nc
is an optional numeric noncentrality parameter.

Range *nc* ≥ 0

Data type DOUBLE

Details

The PROBF function returns the probability that an observation from an *F* distribution, with numerator degrees of freedom *ndf*, denominator degrees of freedom *ddf*, and noncentrality parameter *nc*, is less than or equal to *x*. The PROBF function accepts noninteger degrees of freedom parameters *ndf* and *ddf*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central *F* distribution.

The significance level for an *F* test statistic is given by the following equation.

```
p=1-probf(x,ndf,ddf);
```

Example

The following statement illustrates the PROBF function:

```
proc cas;
   x=probf(3.32,2,3);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.8263933602
```

PROBGAM Function

Returns the probability from a gamma distribution.

Returned data type: DOUBLE

Syntax

PROBGAM(*x*, *a*)

Arguments

x

is a numeric random variable.

Range $x \geq 0$

Data type DOUBLE

a

is a numeric shape parameter.

Data type DOUBLE

Details

The PROBGAM function returns the probability that an observation from a gamma distribution, with shape parameter *a*, is less than or equal to *x*.

Example

The following statement illustrates the PROBGAM function:

```
proc cas;
   x=probgam(1,3);
```

```
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.0803013971
```

PROBHYPR Function

Returns the probability from a hypergeometric distribution.

Returned data type: DOUBLE

Syntax

PROBHYPR(*N*, *K*, *n*, *x*<, *r*>)

Arguments

N

is a population size parameter. This argument must be a whole number.

Range $N \geq 1$

Data type DOUBLE

K

is the number of items in the category of interest parameter. This argument must be a whole number.

Range $0 \leq K \leq N$

Data type DOUBLE

n

is the sample size parameter. This argument must be a whole number.

Range $0 \leq n \leq N$

Data type DOUBLE

x

is the random variable. This argument must be a whole number.

Range $max(0, K + n-N) \leq x \leq min(K,n)$

r

is a numeric odds ratio parameter.

Range $r \geq 0$

Data type DOUBLE

Details

The PROBHYPR function returns the probability that an observation from an extended hypergeometric distribution, with population size *N*, number of items *K*, sample size *n*, and odds ratio *r*, is less than or equal to *x*. If the optional parameter *r* is not specified or is set to 1, the value returned is from the usual hypergeometric distribution.

Example

The following statement illustrates the PROBHYPR function:

```
proc cas;
   x=probhypr(200,50,10,2);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.5236734081
```

PROBIT Function

Returns a quantile from the standard normal distribution.

Returned data type: DOUBLE

Syntax

PROBIT(*p*)

Arguments

p
is a numeric probability.

Range $0 < p < 1$

Data type DOUBLE

Details

The PROBIT function returns the p^{th} quantile from the standard normal distribution. The probability that an observation from the standard normal distribution is less than or equal to the returned quantile is *p*.

CAUTION
The result could be truncated to lie between -8.222 and 7.941.

Note: PROBIT is the inverse of the PROBNORM function.

Example

The following statements illustrate the PROBIT function:

```
proc cas;
   x=probit(.025);
   print x;
   x=probit(1.e-7);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   -1.959963985
   -5.199337582
```

PROBMC Function

Returns a probability or a quantile from various distributions for multiple comparisons of means.

Returned data type: DOUBLE

Syntax

PROBMC(*distribution*, *q*, *prob*, *df*, *nparms*<, *parameters*>)

Arguments

distribution
is a character constant, variable, or expression that evaluates or can be coerced to a character string and that identifies the distribution. The following distributions are valid: ANOM (Analysis of Means), DUNNETT1, DUNNETT2,

MAXMOD (Maximum Modulus), PARTRANGE (Partitioned Range), RANGE (Studentized Range), and WILLIAMS.

is a character constant, variable, or expression that identifies the distribution. The following distributions are valid:

Distribution	Argument
Analysis of Means	`ANOM`
One-sided Dunnett	`DUNNETT1`
Two-sided Dunnett	`DUNNETT2`
Maximum Modulus	`MAXMOD`
Partitioned Range	`PARTRANGE`
Studentized Range	`RANGE`
Williams	`WILLIAMS`

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

q

is the quantile from the distribution.

Restriction Either *q* or *prob* can be specified, but not both.

Data type DOUBLE

prob

is the left probability from the distribution.

Restriction Either *prob* or *q* can be specified, but not both.

Data type DOUBLE

df

is the degrees of freedom.

Note: A missing value is interpreted as an infinite value.

nparms

is the number of treatments.

Data type DOUBLE

Note For DUNNETT1 and DUNNETT2, the control group is not counted.

parameters

is a set of *nparms* parameters that must be specified to handle the case of unequal sample sizes. The meaning of *parameters* depends on the value of *distribution*. If *parameters* is not specified, equal sample sizes are assumed, which is usually the case for a null hypothesis.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Overview

The PROBMC function returns the probability or the quantile from various distributions with finite and infinite degrees of freedom for the variance estimate.

The *prob* argument is the probability that the random variable is less than *q*. Therefore, *p*-values can be computed as 1– *prob*. For example, to compute the critical value for a 5% significance level, set *prob*= 0.95. The precision of the computed probability is $O(10^{-8})$ (absolute error); the precision of computed quantile is $O(10^{-5})$.

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes.

Note: Williams' test is computed only for equal sample sizes.

Formulas and Parameters

The equations listed here define expressions that are used in equations that relate the probability, *prob*, and the quantile, *q*, for different distributions and different situations within each distribution. For these equations, let *v* be the degrees of freedom, *df*.

$$d\mu_\nu(x) = \frac{\nu^{\frac{\nu}{2}}}{\Gamma\left(\frac{\nu}{2}\right)2^{\frac{\nu}{2}-1}} x^{\nu-1}\varepsilon^{-\frac{\nu x^2}{2}}dx$$

$$\phi(x) = \frac{1}{\sqrt{2\,\pi}}\varepsilon^{-\frac{x^2}{2}}$$

$$\Phi(x) = \int_{-\infty}^{x} \phi(u)du$$

Computing the Analysis of Means

Analysis of Means (ANOM) applies to data that is organized as *k* (Gaussian) samples, the i^{th} sample being of size n_i. Let $I = \sqrt{-1}$. The distribution function [1, 2, 3, 4, 5] is the CDF for the maximum absolute of a *k*-dimensional multivariate T

vector, with ν degrees of freedom, and an associated correlation matrix $\rho_{ij} = -\alpha_i\alpha_j$. This equation can be written as follows.

$$prob = r(|t_1| < h, |t_2| < h, ..., |t_k| < h)$$
$$= \int_0^\infty \left\{ \int_0^\infty \prod_{j=0}^{j=k} g(sh, y, \alpha_j)\phi(y)dy \right\} d\mu_\nu(s)$$

The following relationship applies to the preceding equation:

$$g(sh, y, \alpha_j) = \Phi\left(\frac{sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

In this equation, $\Gamma(.)$, $\phi(.)$, and $\Phi(.)$, are the gamma function, the density, and the CDF from the standard normal distribution, respectively.

For $\nu = \infty$, the distribution reduces to this equation.

$$r(|t_1| < h, |t_2| < h, ..., |t_k| < h) = \int_0^\infty \prod_{j=0}^{j=k} g(h, y, \alpha_j)\phi(y)dy$$

The following relationship applies to the preceding equation:

$$g(h, y, \alpha_j) = \Phi\left(\frac{h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

For the balanced case, the distribution reduces to the following equation:

$$r(|t_1| < h, |t_2| < h, ..., |t_n| < h) = \int_0^\infty f(h, y, \rho)^n \phi(y)dy$$

The following relationships apply to the preceding equation:

$$f(h, y, \rho) = \Phi\left(\frac{h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right) - \Phi\left(\frac{-h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right)$$

$$\rho = \frac{1}{n-1}$$

Here is the syntax for this distribution:

x=probmc('anom', q, p, nu, n[, alpha$_1$, ..., alpha$_n$]);

Arguments

x
is a numeric value with the returned result.

q
is a numeric value that denotes the quantile.

p
is a numeric value that denotes the probability. One of *p* and *q* must be missing.

nu
is a numeric value that denotes the degrees of freedom.

n
is a numeric value that denotes the number of samples.

alpha$_i$, i = 1, ...,k

are optional numeric values denoting the alpha values from the first equation of this distribution. See "Computing the Analysis of Means".

Many-One *t*-Statistics: Dunnett's One-Sided Test

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\lambda_1, ..., \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + qx}{\sqrt{1-\lambda_i^2}}\right) dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed $\left(\lambda = \sqrt{\frac{1}{2}}\right)$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y)[\Phi(y + \sqrt{2qx})]^k dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\lambda_1, ..., \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \Phi\left(\frac{\lambda_i y + q}{\sqrt{1-\lambda_i^2}}\right) dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed $\left(\lambda = \sqrt{\frac{1}{2}}\right)$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^\infty \phi(y)[\Phi(y + \sqrt{2q})]^k dy$$

Many-One *t*-Statistics: Dunnett's Two-sided Test

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\lambda_1, ..., \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y) \prod_{i=1}^k \left[\Phi\left(\frac{\lambda_i y + qx}{\sqrt{1-\lambda_i^2}}\right) - \Phi\left(\frac{\lambda_i y - qx}{\sqrt{1-\lambda_i^2}}\right)\right] dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^\infty \int_{-\infty}^\infty \phi(y)[\Phi(y + \sqrt{2qx}) - \Phi(y - \sqrt{2qx})]^k dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are $\lambda_1, ..., \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + q}{\sqrt{1-\lambda_i^2}} \right) - \Phi\left(\frac{\lambda_i y - q}{\sqrt{1-\lambda_i^2}} \right) \right] dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \phi(y)[\Phi(y+\sqrt{2q}) - \Phi(y-\sqrt{2q})]^k dy$$

Computing the Partitioned Range

RANGE applies to the distribution of the studentized range for *n* group means. PARTRANGE applies to the distribution of the partitioned studentized range. Let the *n* groups be partitioned into *k* subsets of size $n_1 + \ldots + n_k = n$. Then the partitioned range is the maximum of the studentized ranges in the respective subsets. The studentization factor is the same in all cases.

$$prob = \int_{0}^{\infty} \prod_{i=1}^{i=k} \left(\int_{-\infty}^{\infty} k\phi(y)(\Phi(y) - \Phi(y-qx))^{k-1} dy \right)^{n_i} d\mu_\nu(x)$$

Here is the syntax for this distribution:

x=probmc('partrange', q, p,nu, k, n_1,...,n_k);

Arguments

x
is a numeric value with the returned result (either the probability or the quantile).

q
is a numeric value that denotes the quantile.

p
is a numeric value that denotes the probability. One of *p* and *q* must be missing.

nu
is a numeric value that denotes the degrees of freedom.

k
is a numeric value that denotes the number of groups.

n_i i=1,...,k
are optional numeric values that denote the *n* values from the equation in this distribution. See "Computing the Partitioned Range".

The Studentized Range

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty} \int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y-qx)]^{k-1} dy\, du_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are σ_1, ..., σ_k, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \sum_{j=1}^{k} \left\{ \prod_{i=1}^{k} \left[\Phi\left(\frac{y}{\sigma_i}\right) - \Phi\left(\frac{y-q}{\sigma_i}\right) \right] \right\} \phi\left(\frac{y}{\sigma_j}\right) \frac{1}{\sigma_j} dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y-q)]^{k-1} dy$$

The Studentized Maximum Modulus

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are σ_1, ..., σ_k, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty} \prod_{i=1}^{k} \left[2\Phi\left(\frac{qx}{\sigma_i}\right) - 1 \right] d\mu_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty} [2\Phi(qx) - 1]^k d\mu_\nu(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are σ_1, ..., σ_k, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \prod_{i=1}^{k} \left[2\Phi\left(\frac{q}{\sigma_i}\right) - 1 \right]$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = [2\Phi(q) - 1]^k$$

Williams' Test

The need for the Williams' Test arises when you compare the dose treatment means with a control mean to determine the lowest effective dose of treatment.

Note: Williams' Test is computed only for equal sample sizes.

Let X_1, X_2, ..., X_k be identical independent N(0,1) random variables. Let Y_k denote their average given by the following equation.

$$Y_k = \frac{X_1 + X_2 + ... + X_k}{k}$$

It is required to compute the distribution of the following value.

$$(Y_k - Z)/S$$

Arguments

Y_k

is as defined previously.

Z

is an N(0,1) independent random variable.

S

is such that ½vS^2 is a χ^2 variable with v degrees of freedom.

1 Compute the distribution of Y_k. It is the fundamental (expensive) part of this operation and it can be used to find both the density and the probability of Y_k. Let U_i be defined in this equation.

$$U_i = \frac{X_1 + X_2 + ... + X_i}{i},\ i = 1, 2, ..., k$$

You can write a recursive expression for the probability of $Y_k > d$. The value of d can be any real number.

$$\begin{aligned}\Pr(Y_k > d) &= \Pr(U_1 > d)\\ &\quad +\Pr(U_2 > d, U_1 < d)\\ &\quad +\Pr(U_3 > d, U_2 < d, U_1 < d)\\ &\quad +\ldots\\ &\quad +\Pr(U_k > d, U_{k-1} < d, \ldots, U_1 < d)\\ &= \Pr(Y_{k-1} > d) + \Pr(X_k + (k-1)U_{k-1} > kd)\end{aligned}$$

To compute this probability, start from an N(0,1) density function.

$$D(U_1 = x) = \phi(x)$$

And recursively compute the convolution.

$$D(U_k = x, U_{k-1} < d, \ldots, U_1 < d) =$$

$$\int_{-\infty}^{d} D(U_{k-1} = y, U_{k-2} < d, \ldots, U_1 < d)(k-1)\phi(kx - (k-1)y)dy$$

From this sequential convolution, it is possible to compute all the elements of the recursive equation for $\Pr(Y_k < d)$, shown previously.

2 Compute the distribution of $Y_k - Z$. This computation involves another convolution to compute the probability.

$$\Pr((Y_k - Z) > d) = \int_{-\infty}^{\infty} \Pr(Y_k > \sqrt{2d} + y)\phi(y)dy$$

3 Compute the distribution of $(Y_k - Z)/S$. This computation involves another convolution to compute the probability.

$$\Pr((Y_k - Z) > tS) = \int_0^\infty \Pr((Y_k - Z) > ty) d\mu_\nu(y)$$

The third stage is not needed when v = ∞. Due to the complexity of the operations, this lengthy algorithm is replaced by a much faster one when $k \le 15$ for both finite and infinite degrees of freedom v. For $k \ge 16$, the lengthy computation is carried out. It is extremely expensive and very slow due to the complexity of the algorithm.

Comparisons

The MEANS statement in the GLM Procedure of SAS/STAT Software computes the following tests:

- Dunnett's one-sided test
- Dunnett's two-sided test
- Studentized Range

Example: 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
X_2	10.0
X_3	10.6
X_4	11.4
X_5	11.9
X_6	11.7

The mean square, with (7 – 1)(8 – 1) = 42 degrees of freedom, is $s^2 = 1.16$.

Determine the maximum likelihood estimates M_i through the averaging process.

- Because $X_0 > X_1$, form $X_{0,1} = (X_0 + X_1)/2 = 10.15$.
- Because $X_{0,1} > X_2$, form $X_{0,1,2} = (X_0 + X_1 + X_2)/3 = (2X_{0,1} + X_2)/3 = 10.1$.

- $X_{0,1,2} < X_3 < X_4 < X_5$
- Because $X_5 > X_6$, form $X_{5,6} = (X_5 + X_6)/2 = 11.8$.

Now the order restriction is satisfied.

The maximum likelihood estimates under the alternative hypothesis are:

- $M_0 = M_1 = M_2 = X_{0,1,2} = 10.1$
- $M_3 = X_3 = 10.6$
- $M_4 = X_4 = 11.4$
- $M_5 = M_6 = X_{5,6} = 11.8$

Now compute $t = (11.8 - 10.4)/\sqrt{2s^2/8} = 2.60$, and the probability that corresponds to $k = 6$, v = 42, and $t = 2.60$ is .9924467341, which shows strong evidence that there is a response to the substance. You can also compute the quantiles for the upper 5% and 1% tails.

```
proc cas;
      prob=probmc('WILLIAMS',2.6,.,42,6);
      print "prob=" prob;
      quant5=probmc('WILLIAMS',.,.95,42,6);
      print "quant5=" quant5;
      quant1=probmc('WILLIAMS',.,.99,42,6);
      print "quant1=" quant1;
run;
```

The following output is printed to the SAS Log:

```
prob=0.9924466872
quant5=1.806562536
quant1=2.490908273
```

References

Guirguis, G. H., and R. D. Tobias. 2004. "On the Computation of the Distribution for the Analysis of Means." *Communications in Statistics: Simulation and Computation* 33: 861–887.

Nelson, P. R. 1981. "Numerical evaluation of an equicorrelated multivariate non-central t distribution." *Communications in Statistics: Part B - Simulation and Computation* 10: 41–50.

Nelson, P.R. 1982a. "An Approximation for the Complex Normal Probability Integral." *BIT* 22(1): 94–100.

Nelson, P. R. 1982b. "Exact critical points for the analysis of means." *Communications in Statistics: Part A - Theory and Methods* 11: 699–709.

Nelson, P. R. 1988. "Application of the Analysis of Means." *Proceedings of the SAS Users Group International Conference* 13: 225–230.

Nelson, P. R. 1991. "Numerical evaluation of multivariate normal integrals with correlations." *The Frontiers of Statistical Scientific Theory and Industrial Applications* 2: 97–114.

Nelson, P. R. 1993. "Additional Uses for the Analysis of Means and Extended Tables of Critical Values." *Technometrics* 35: 61–71.

PROBNEGB Function

Returns the probability from a negative binomial distribution.

Returned data type: DOUBLE

Syntax

PROBNEGB(*p*, *n*, *m*)

Arguments

p

is a numeric probability of success parameter.

Range $0 \leq p \leq 1$

Data type DOUBLE

n

is the number of successes parameter. This argument must be a whole number.

Range $n \geq 1$

Data type DOUBLE

m

is the random variable, the number of failures. This argument must be a positive, whole number.

Range $m \geq 0$

Data type DOUBLE

Details

The PROBNEGB function returns the probability that an observation from a negative binomial distribution, with probability of success *p* and number of successes *n*, is less than or equal to *m*.

To compute the probability that an observation is equal to a given value *m*, compute the difference of two probabilities from the negative binomial distribution for *m* and *m*−1.

Example

The following statement illustrates the PROBNEGB function:

```
proc cas;
   x=probnegb(0.5,2,1);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   0.5
```

PRXCHANGE Function

Performs a pattern-matching replacement.

Returned data type: STRING

Syntax

PRXCHANGE(*perl-regular-expression* | *regular-expression-id*, *times*, *source*)

Arguments

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

Data type CHAR

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned from the PRXPARSE function.

Restriction If you use this argument, you must also use the PRXPARSE function.

Data type INTEGER

times
is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern.

Data type INTEGER

Tip If the value of *times* is –1, then matching patterns continue to be replaced until the end of *source* is reached.

source
specifies a character constant, variable, or expression that you want to search.

Data type CHAR

Details

The Basics

If you use *regular-expression-id*, the PRXCHANGE function searches the *source* with the *regular-expression-id* that is returned by PRXPARSE. It returns the value in *source* with the changes that were specified by the regular expression. If there is no match, PRXCHANGE returns the unchanged value in *source*.

If you use *perl-regular-expression*, PRXCHANGE searches the *source* with the *perl-regular-expression*, and you do not need to call PRXPARSE. You can use PRXCHANGE with a *perl-regular-expression* in a WHERE clause and in PROC SQL.

Note: The following restrictions apply to PRX functions in DS2:

- Only **m**, **i**, **s**, and **x** can be used in the PRX form /.../.../ that can be preceded or followed by a single character.
- The matching mode modifiers **p**, **o**, **c**, **a**, and **l** are not supported.
- The matching mode modifier **g** is supported.

Example: Changing the Order of First and Last Names By Using the DATA Step

The following example changes the order of first and last names.

```
proc cas;
/*Create four varibles that contain four different names. Include both
first and last names*/
  name1="Jones, Fred";
  name2="Kavich, Kate";
  name3="Turley, Ron";
  name4="Dulix, Yolanda";
/*Reverse the first and last name*/
  nameA=prxchange('s/(\w+), (\w+)/$2 $1/', -1, name1);
  nameB=prxchange('s/(\w+), (\w+)/$2 $1/', -1, name2);
  nameC=prxchange('s/(\w+), (\w+)/$2 $1/', -1, name3);
  nameD=prxchange('s/(\w+), (\w+)/$2 $1/', -1, name4);
```

```
  columns={'name(before)', 'name(after)'};
  coltypes={'string', 'string'};
  row1={name1, nameA};
  row2={name2, nameB};
  row3={name3, nameC};
  row4={name4, nameD};
  mytable=newtable('mytable', columns, coltypes, row1, row2, row3,
row4);
  print mytable;
run;
```

Output 4.5 *Results from the PRXCHANGE Function*

mytable: Results

name(before)	name(after)
Jones, Fred	Fred Jones
Kavich, Kate	Kate Kavich
Turley, Ron	Ron Turley
Dulix, Yolanda	Yolanda Dulix

PRXMATCH Function

Searches for a pattern match and returns the position at which the pattern is found.

Returned data type: INTEGER

Syntax

PRXMATCH(*perl-regular-expression*, *source*)

Arguments

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

Data type CHAR

source
specifies a character constant, variable, or expression that you want to search.

Data type CHAR

Details

The Basics

When you use *perl-regular-expression*, the PRXMATCH function searches *source* with the *perl-regular-expression* and returns the position at which the string begins. If there is no match, PRXMATCH returns a zero..

You can use PRXMATCH with a Perl regular expression in a WHERE clause and in PROC SQL.

Note: The following restrictions apply to PRX functions in DS2:

- Only `m`, `i`, `s`, and `x` can be used in the PRX form /.../.../ that can be preceded or followed by a single character.
- The matching mode modifiers `p`, `o`, `c`, `a`, and `l` are not supported.

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.

Example: 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.

```
proc cas;
   position=prxmatch('/world/', 'Hello world!');
   print "position=" position;
run;
```

SAS writes the following output to the log:

```
position=7
```

PRXPAREN Function

Returns the last bracket match for which there is a match in a pattern.

Restriction: Use with the PRXPARSE function.

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 Basics

The PRXPAREN function is useful in finding the largest capture-buffer number that can be passed to the PRXPOSN function, or in identifying which part of a pattern matched.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " in *SAS Functions and CALL Routines: Reference*.

Example

The following example uses Perl regular expressions and writes the results to the SAS log.

```
proc cas;
   ExpressionID=prxparse('/(magazine)|(book)|(newspaper)/');
   position=prxmatch(ExpressionID, 'find book here');
   if position then paren=prxparen(ExpressionID);
   print 'Matched paren ' paren;
   position=prxmatch(ExpressionID, 'find magazine here');
   if position then paren=prxparen(ExpressionID);
   print 'Matched paren ' paren;
   position=prxmatch(ExpressionID, 'find newspaper here');
   if position then paren=prxparen(ExpressionID);
   print 'Matched paren ' paren;
run;
```

SAS writes the following output to the log:

```
Matched paren 2
Matched paren 1
Matched paren 3
```

PRXPARSE Function

Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.

Restriction: Use with other Perl regular expressions.

Returned data type: INTEGER

Syntax

regular-expression-id=**PRXPARSE**(*perl-regular-expression*)

Arguments

regular-expression-id
is a numeric pattern identifier that is returned by the PRXPARSE function.

Data type INTEGER

perl-regular-expression
specifies a character, constant, variable, or expression with a value that is a Perl regular expression.

Data type CHAR

Details

The Basics

The PRXPARSE function returns a pattern identifier number that is used by other Perl functions to match patterns. If an error occurs in parsing the regular expression, SAS returns a missing value.

PRXPARSE uses metacharacters in constructing a Perl regular expression.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant, the Perl regular expression is compiled only once. Successive calls to PRXPARSE will not cause a recompile, but will return 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.

Example: Compiling a Perl Regular Expression

The following example uses PRXPARSE to compile the Perl regular expression.

```
proc cas;
     /* 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!');
     print "position=" position;
run;
```

The following output is printed to the SAS Log:

```
position=7
```

PRXPOSN Function

Returns a character string that contains the value for a capture buffer.

Returned data type: STRING

Syntax

PRXPOSN(*regular-expression-id*, *capture-buffer*, *source*)

Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned by the PRXPARSE function.

Restriction

Data type INTEGER

capture-buffer
is a numeric constant, variable, or expression that identifies the capture buffer for which to retrieve a value. If the value of *capture-buffer* is zero, PRXPOSN returns the entire match. If the value of *capture-buffer* is between 1 and the number of open parentheses in the regular expression, then PRXPOSN returns the value for that capture buffer. If the value of *capture-buffer* is greater than the number of open parentheses, then PRXPOSN returns a missing value.

is a numeric constant, variable, or expression that identifies the capture buffer for which to retrieve a value:

- If the value of *capture-buffer* is zero, PRXPOSN returns the entire match.

- If the value of *capture-buffer* is between 1 and the number of open parentheses in the regular expression, then PRXPOSN returns the value for that capture buffer.
- If the value of *capture-buffer* is greater than the number of open parentheses, then PRXPOSN returns a missing value.

Data type INTEGER

source
specifies the text from which to extract capture buffers.

Data type CHAR

Details

The PRXPOSN function uses the results of PRXMATCH or PRXCHANGE to return a capture buffer. A match must be found by one of these functions for PRXPOSN to return meaningful information.

Note: The following restrictions apply to PRX functions in DS2:

- Only **m**, **i**, **s**, and **x** can be used in the PRX form /.../.../ that can be preceded or followed by a single character.
- The matching mode modifiers **p**, **o**, **c**, **a**, and **l** are not supported.

Example

```
proc cas;
   rxID=prxparse('m/([\-\(]?\d{3}[\s\-\)\.]?\s?\d{3}[\s\-\.]?
\d{4})/');  /*1*/
   textArray={'Call me at (311) 555-2368 or (123) 555-2121 '
        ,'212.664.7665 is my number'
        'Phone 123-555-2121 for more information.'};
    do thisText over textArray;

Phone=PRXPOSN(rxID,1,thisText);                                    /
*2*/
       print "Text string: " thisText;
       print "Phone number: " Phone;
    end;
run;
```

1 The regular expression matches US phone numbers written in a variety of formats. It is syntax checked and compiled by the PRXPARSE function, and a numeric ID for the compiled version is assigned to the variable `rxID`. Any text matching the pattern will be accessible in the first capture buffer.

2 This call to PRXPOSN applies the regular (`rxID`) to the value of `thisText` and returns the text found in the first capture buffer.

The program writes the following output to the log:

```
Text string: Call me at (311) 555-2368 or (123) 555-2121
Phone number: (311) 555-2368
Text string: 212.664.7665 is my number
Phone number: 212.664.7665
Text string: Phone 123-555-2121 for more information.
Phone number: 123-555-2121
```

PUTC Function

Enables you to specify a character format at run time.

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.

Example: Aligning Character Values

This example shows how to use a format and an alignment character to align the value of A.

```
proc cas;
    a='experiment';
    y=putc(a, 'upcase.-r', 20);
    print '*' y '*';
    print '*' a '*';
run;
```

The following output is printed to the SAS Log:

```
*EXPERIMENT          *
*experiment*
```

PUTN Function

Enables you to specify a numeric format at run time.

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*.

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.

Example: Aligning Output from the PUTN Function

This example shows how to use a format and an alignment character to left-align a value.

```
proc cas;
    y=putn(today(), 'weekdate.-1', 30);
    print '*' y '*';
run;
```

Example Code 4.1 *Alignment Results*

```
*          Friday, May 11, 2018*
```

PVP Function

Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.

Returned data type: DOUBLE

Syntax

PVP(*A*, *c*, *n*, *K*, k_0, *y*)

Arguments

A

specifies the par value.

Ranges $A > 0$

$A > 0$

Data type DOUBLE

c

specifies the nominal per-year coupon rate, expressed as a fraction.

Ranges $0 \leq c < 1$

$0 \leq c < 1$

Data type DOUBLE

n

specifies the number of coupons per year.

Ranges $n > 0$

$n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons.

Ranges $K > 0$

$K > 0$

Data type DOUBLE

k_0

specifies the time from the present date to the first coupon date, expressed in terms of the number of years.

Ranges $0 < k_0 \leq \frac{1}{n}$

$0 < k_0 \leq 1/n$

Data type DOUBLE

y

specifies the nominal per-year yield-to-maturity, expressed as a fraction.

Ranges $y > 0$

$y > 0$

Data type DOUBLE

Details

The PVP function is based on the following relationship:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

Example

```
proc cas;
     p=pvp(1000,.01,4,14,.33/2,.10);
     print "p=" p;
run;
```

The value that is returned is 743.167613519067.

QTR Function

Returns the quarter of the year from a SAS date value.

Returned data type: DOUBLE

Syntax

QTR(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

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

The following statements illustrate the QTR function:.

```
proc cas;
   theDate=inputn('07SEP2024','DATE9.');
   theQuarter=qtr(theDate);
   print(NOTE) "Date=" putn(theDate,'date9.') " Quarter=" theQuarter;
run;
```

The following output is printed to the SAS Log:

```
   NOTE: Date=07SEP2024 Quarter=3
```

QUOTE Function

Adds double quotation marks to a character value.

Returned data type: STRING, VARCHAR

Syntax

QUOTE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The QUOTE function adds double quotation marks, the default character, to a character value. If double quotation marks are found within the argument, they are doubled in the output.

The length of the receiving variable must be long enough to contain the argument (including trailing blanks), leading and trailing quotation marks, and any embedded quotation marks that are doubled. For example, if the argument is ABC followed by three trailing blanks, then the receiving variable must have a length of at least eight to hold “ABC###”. (The character # represents a blank space.) If the receiving field is not long enough, the QUOTE function returns a blank string, and writes an invalid argument note to the SAS log.

A string of characters enclosed in double quotation marks is a DS2 identifier and not a character constant. The double quotation marks become part of the identifier. Quoted identifiers cannot be used to create column names in an output table.

Example

The following statements illustrate the QUOTE function:

```
proc cas;
   a='A"B';
   b=quote(a);
   print b;
run;
```

The following output is printed to the SAS Log:

```
   "A""B"
```

RAND Function

Generates random numbers from a distribution that you specify.

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
	`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)

Distribution	Function Call	Parameter Values
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
	`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))

Distribution	Function Call	Parameter Values
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
	`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.

Note: If you do not specify a value for the *distribution* argument, the uniform distribution is used.

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.

Example

This example uses the RAND function and the STREAMINIT function.

```
/*** Simulate flipping a coin 2000 times
     and count heads and tails ***/

proc cas;
   streaminit(1234);   /* Use any positive integer */
   nheads = 0;
   ntails = 0;
   do flip = 1 to 2000;
      if rand( 'uniform' ) > 0.5   /* 0.5 simulates a fair coin */
         then nheads = nheads + 1;
         else ntails = ntails + 1;
   end;
   print 'There were ' nheads 'heads and ' ntails 'tails.';
run;
```

Output 4.6 *SAS Log*

```
There were 960 heads and 1040 tails.
```

See Also

"RAND Function" in *SAS Functions and CALL Routines: Reference*

RANGE Function

Returns the difference between the largest and the smallest values.

Returned data type: DOUBLE

Syntax

RANGE(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

Details

The RANGE function returns the difference between the largest and the smallest of the non-null or nonmissing arguments.

Example

The following statements illustrate the RANGE function:

```
proc cas;
   a=range(-2,6,3);
   print a;
   a=range(2,6,3,.);
   print a;
   a=range(1,6,3,1);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   8
   4
   5
```

RANK Function

Returns the position of a character in the ASCII or EBCDIC collating sequence.

Returned data type: DOUBLE

Syntax

RANK(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The RANK function returns a whole number that represents the position of the first character in the character expression.

Example

The following statement illustrates the RANK function:

```
proc cas;
   a=rank('A');
   print a;
run;
```

The following output is printed to the SAS Log:

```
   65
```

REPEAT Function

Repeats a character expression.

Returned data type: STRING

Syntax

REPEAT(*expression*, *n*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

n
specifies the number of times to repeat *expression*.

Restriction *n* must be greater than or equal to 0.

Data type BIGINT, DOUBLE

Details

The REPEAT function returns a character value consisting of the first argument repeated *n* times. Thus, the first argument appears *n*+1 times in the result.

Example

The following statement illustrates the REPEAT function:

```
proc cas;
   a=repeat('ONE',2);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   ONEONEONE
```

REVERSE Function

Reverses a character expression.

Returned data type: STRING, VARCHAR

Syntax

REVERSE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The REVERSE function returns a character value with the last character in the expression is the first character in the result, the next-to-last character in the expression is the second character in the result, and so on.

Note: Trailing blanks in the expression become leading blanks in the result.

Example

The following statement illustrates the REVERSE function:

```
proc cas;
   a=reverse('xyz  ');
   print a;
run;
```

The following output is printed to the SAS Log:

```
     zyx
```

RIGHT Function

Right aligns a character expression.

Returned data type: STRING, VARCHAR

Syntax

RIGHT(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The RIGHT function returns an argument with trailing blanks moved to the start of the value. The argument's length does not change.

Example

The following example uses the RIGHT function to move the three trailing blanks to the start of the value.

```
proc cas;
   a=RIGHT("SAS Institute   ");
   print a;
run;
```

Output 4.7 *Log Output*

```
   SAS Institute
```

RMS Function

Returns the root mean square.

Returned data type: DOUBLE

Syntax

RMS(*expression* <, ...*expression*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are null or missing values, then the result is a null or missing value. Otherwise, the result is the root mean square of the non-null or nonmissing values.

Let n be the number of arguments with non-null or nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The root mean square is calculated as follows.

$$\sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}$$

Example

The following statements illustrate the RMS function:

```
proc cas;
   x1=rms(1,7);
   print x1;
   x2=rms(.,1,5,11);
   print x2;
run;
```

The following output is printed to the SAS Log:

```
   5
   7
```

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.

Returned data type: DOUBLE

Syntax

ROUND(*expression* <, *rounding-unit*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value, to be rounded.

Data type DOUBLE

rounding-unit
specifies a positive numeric expression that specifies the rounding unit.

Data type DOUBLE

Details

Basic Concepts

The ROUND function rounds the first argument to a value that is very close to a multiple of the second argument. The results might not be an exact multiple of the second argument.

Differences between Binary and Decimal Arithmetic

Computers use binary arithmetic with finite precision. If you work with numbers that do not have an exact binary representation, computers often produce results that differ slightly from the results that are produced with decimal arithmetic.

For example, the decimal values 0.1 and 0.3 do not have exact binary representations. In decimal arithmetic, 3*0.1 is exactly equal to 0.3, but this equality is not true in binary arithmetic.

As the following example shows, if **a** is a float and **b** is a REAL, there is a difference between the two values.

```
proc cas;
      a=0.3;
      b=3*0.1;
      diff=a-b;
      print "a=" a;
      print "b=" b;
      print "diff=" diff;
run;
```

The following lines are written to the SAS log:

```
a=0.3
b=0.3
diff=-5.55112E-17
```

Operating Environment Information: The example above was executed in the Windows environment. If you use other operating environments, the results will be slightly different.

The Effects of Rounding

Rounding by definition finds an exact multiple of the rounding unit that is closest to the value to be rounded. For example, 0.33 rounded to the nearest tenth equals 3*0.1 or 0.3 in decimal arithmetic. In binary arithmetic, 0.33 rounded to the nearest tenth equals 3*0.1, and not 0.3, because 0.3 is not an exact multiple of one tenth in binary arithmetic.

The ROUND function returns the value that is based on decimal arithmetic, even though this value is sometimes not the exact, mathematically correct result. In the example `ROUND(0.33,0.1)`, ROUND returns 0.3 and not 3*0.1.

Expressing Binary Values

If the characters "0.3" appear as a constant in a DS2 program, the value is computed as 3/10. To be consistent with the standard informat, `ROUND(0.33,0.1)` computes the result as 3/10, and the following statement produces the results that you would expect.

```
if round(x,0.1) = 0.3 then
   ... more DS2 statements ...
```

However, if you use the variable Y instead of the constant 0.3, as the following statement shows, the results might be unexpected depending on how the variable Y is computed.

```
if round(x,0.1) = y then
   ... more DS2 statements ...
```

If ROUND reads Y as the characters "0.3" using the standard informat, the result is the same as if a constant 0.3 appeared in the IF statement. If ROUND reads Y with a different informat, or if a program other than SAS reads Y, then there is no guarantee that the characters "0.3" would produce a value of exactly 3/10. Imprecision can also be caused by computation involving numbers that do not have exact binary representations, or by porting tables from one operating environment to another that has a different floating-point representation.

If you know that Y is a decimal number with one decimal place, but are not certain that Y has exactly the same value as would be produced by the standard informat, it is better to use the following statement:

```
if round(x,0.1) = round(y,0.1) then
   ... more DS2 statements ...
```

Producing Expected Results

In general, `ROUND(expression, rounding-unit)` produces the result that you expect from decimal arithmetic if the result has no more than nine significant digits and any of the following conditions are true:

- The rounding unit is an integer.
- The rounding unit is a power of 10 greater than or equal to 1e-15. [1]

1. If the rounding unit is less than one, ROUND treats it as a power of 10 if the reciprocal of the rounding unit differs from a power of 10 in at most the three or four least significant bits.

- The result that you expect from decimal arithmetic has no more than four decimal places.

When the Rounding Unit Is the Reciprocal of an Integer

When the rounding unit is the reciprocal of an integer [1], the ROUND function computes the result by dividing by the integer. Therefore, you can safely compare the result from ROUND with the ratio of two integers, but not with a multiple of the rounding unit.

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.

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.

Comparisons

The ROUND function is the same as the ROUNDE function except that when the first argument is halfway between the two nearest multiples of the second argument, ROUNDE returns an even multiple. ROUND returns the multiple with the larger absolute value.

The ROUNDZ function returns a multiple of the rounding unit without trying to make the result match the result that is computed with decimal arithmetic.

Example

```
proc cas;
   Value=55.352;
   Print "Original Value =" Value;
   Print "ROUND(Value)=" round(Value);
   Print "ROUND(Value,10)=" round(Value,10);
```

1. ROUND treats the rounding unit as a reciprocal of an integer if the reciprocal of the rounding unit differs from an integer in at most the three or four least significant bits.

```
   Print "ROUND(Value,.1)=" round(Value,.1);
   Print "ROUND(value,.25)=" round(value,.25);
run;
```

The program writes the following output to the log:

```
Original Value =55.352
ROUND(Value)=55
ROUND(Value,10)=60
ROUND(Value,.1)=55.4
ROUND(value,.25)=55.25
```

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.

Returned data type: DOUBLE

Syntax

ROUNDE(*expression* <, *rounding-unit*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value and that is to be rounded.

Data type DOUBLE

rounding-unit
is a positive, numeric expression that specifies the rounding unit.

Default 1

Data type DOUBLE

Details

The ROUNDE function rounds the first argument to the nearest multiple of the second argument.

Comparisons

The ROUNDE function is the same as the ROUND function except that when the first argument is halfway between the two nearest multiples of the second argument, ROUNDE returns an even multiple. ROUND returns the multiple with the larger absolute value.

Example

The following example compares the results that are returned by the ROUNDE function with the results that are returned by the ROUND function.

```
proc cas;
  x=0;
    do x=0 to 4 by .25;
    rounde=rounde(x);
    round=round(x);
  print "rounde=" rounde;
  print "round=" round;
    end;
run;
```

The following is printed to the SAS log

```
rounde=0     round=0
rounde=0     round=0
rounde=0     round=1
rounde=1     round=1
rounde=1     round=1
rounde=1     round=1
rounde=2     round=2
rounde=2     round=2
rounde=2     round=2
rounde=2     round=2
rounde=2     round=3
rounde=3     round=3
rounde=3     round=3
rounde=3     round=3
rounde=4     round=4
rounde=4     round=4
rounde=4     round=4
```

ROUNDZ Function

Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.

Returned data type: DOUBLE

Syntax

ROUNDZ(*expression* <, *rounding-unit*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

rounding-unit
specifies any valid expression that evaluates to a numeric expression and that specifies the rounding unit.

Default 1

Requirement Only positive values are valid.

Data type DOUBLE

Details

The ROUNDZ function rounds the first argument to the nearest multiple of the second argument.

Comparisons

The ROUNDZ function is the same as the ROUND function with these exceptions:

- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument. ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples.
- When the rounding unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUNDZ does not fuzz the result. ROUND performs extra computations, called fuzzing, to try to make the result agree with decimal arithmetic.

Example: Comparing Results from the ROUNDZ and ROUND Functions

The following example compares the results that are returned by the ROUNDZ and the ROUND function.

```
proc cas;
      do i=10 to 17;
         Value=2.5 - 10**(-i);
         Roundz1=roundz(value);
         Round1=round(value);
         print "Roundz1=" Roundz1;
         print "Round1=" Round1;
      end;
      do i=16 to 12 by -1;
         value=2.5 + 10**(-i);
         roundz=roundz(value);
         round=round(value);
         print "roundz=" roundz;
         print "round=" round;
      end;
run;
```

The following output shows the results.

```
Roundz1=2     Round1=3
Roundz1=2     Round1=3
Roundz1=2     Round1=3
Roundz1=2     Round1=3
Roundz1=2     Round1=3
Roundz1=2     Round1=3
Roundz1=2     Round1=3
Roundz1=2     Round1=3
roundz=2      round=3
roundz=2      round=3
roundz=2      round=3
roundz=2      round=3
roundz=2      round=3
```

SAVING Function

Returns the future value of a periodic saving.

Returned data type: DOUBLE

Syntax

SAVING(*f*, *p*, *r*, *n*)

Arguments

f

is numeric, the future amount (at the end of *n* periods).

Range $f \geq 0$

Data type DOUBLE

p

is numeric, the fixed periodic payment.

Range $p \geq 0$

Data type DOUBLE

r

is numeric, the periodic interest rate expressed as a decimal.

Range $r \geq 0$

Data type DOUBLE

n

is an integer, the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

Details

The SAVING function returns the missing argument in the list of four arguments from a periodic saving. The arguments are related by the following equation:

$$f = \frac{p(1+r)\left((1+r)^n - 1\right)}{r}$$

One missing argument must be provided. It is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Example

A savings account pays a 5% nominal annual interest rate, compounded monthly. For a monthly deposit of $100, the number of payments that are needed to equal at least $12,000, can be expressed as follows:

```
proc cas;
   number=saving(12000, 100, .05/12, .);
   print number;
run;
```

The value that is returned is 97.18 months. The fourth argument is set to missing, which indicates that the number of payments needs 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.

SAVINGS Function

Returns the balance of a periodic savings by using variable interest rates.

Returned data type: DOUBLE

Syntax

SAVINGS(*base-date*, *initial-deposit-date*, *deposit-amount*, *deposit-number*, *deposit-interval*, *compounding-interval*, *date-1*, *rate-1*<, ...*date-n*, *rate-n*>)

Arguments

base-date
specifies the value that is returned is the balance of the savings at the base date.

Requirement *Base-date* is a SAS date.

Data type DOUBLE

initial-deposit-date
specifies the date of the first deposit. Subsequent deposits are at the beginning of subsequent deposit intervals.

Requirement *Initial-deposit-date* is a SAS date.

Data type DOUBLE

deposit-amount
specifies the value of each deposit. All deposits are assumed constant.

Data type DOUBLE

deposit-number
specifies the number of deposits.

Data type DOUBLE

deposit-interval
specifies the frequency at which deposits are made.

Requirement *Deposit-interval* is a SAS interval.

Data type CHAR

compounding-interval
specifies the compounding interval.

Requirement *Compounding-interval* is a SAS interval.

Data type CHAR

date
specifies the time at which *rate* takes effect. Each date is paired with a rate.

Requirement *Date* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate as numeric percentage that starts on *date*. Each rate is paired with a date.

Data type DOUBLE

Details

The following details apply to the SAVINGS function:

- The values for rates must be between –99 and 120.
- *Deposit-interval* cannot be 'CONTINUOUS'.
- The list of date-rate pairs does not need to be in chronological order.
- When multiple rate changes occur on a single date, the SAVINGS function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the *initial-deposit-date* and the *base-date*.

Example

- If you deposit $300 monthly for two years into an account that compounds quarterly at an annual rate of 4%, the balance of the account after five years can be expressed as follows:

```
proc cas;
      bd=16437;
      idd=14612;
      d=14612;
      amount_base1=savings(bd, idd, 300, 24, 'month', 'qtr', d, 4.00);
      print "amount_base1=" amount_base1;
run;
```

The following line is written to the SAS log.

```
amount_base1=8458.7145034
```

- To determine the balance after one year of deposits, the following statement sets amount_base3 to the desired balance:

```
proc cas;
      bd=14976;
      idd=14610;
      d=14610;
    amount_base3 = savings(bd, idd, 300, 24, 'month', 'qtr', d, 4);
      print "amount_base3=" amount_base3;
run;
```

  The following line is written to the SAS log.

```
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 expression.

Returned data type: STRING, VARCHAR

Syntax

SCAN(*expression*, *n* <, *delimiters*<, *modifier*>>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

n
is a nonzero numeric expression that specifies the number of the word in the character expression that you want SCAN to select. The following rules apply: If *n* is positive, SCAN counts words from left to right in the character string. If *n* is negative, SCAN counts words from right to left in the character string. If *n* is greater than the number of words in *expression*, SCAN returns a blank value.

is a nonzero numeric expression that specifies the number of the word in the character expression that you want SCAN to select. The following rules apply:

- If *n* is positive, SCAN counts words from left to right in the character string.
- If *n* is negative, SCAN counts words from right to left in the character string.

- If *n* is greater than the number of words in *expression*, SCAN returns a blank value.

delimiters

specifies any valid expression that evaluates or can be coerced to a character string and that SCAN uses as word separators in the expression.

Default

Requirement: If *delimiter* is a constant, enclose *delimiter* in single quotation marks.

Interactions: ASCII default delimiters are: blank ! $ % & ( ) * + , – . / ; < |. In environments without the ^ character, SCAN uses the ~ character instead.

EBCDIC default delimiters are: blank ! $ % & ( ) * + , – . / ; < ¬ | ¢.

Specifying a modifier can change the characters in *delimiter*. For example, if you specify the K modifier in the *modifier* argument, all characters that are not in *delimiter* are used as delimiters.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Tip: You can add more characters to *delimiter* by using other modifiers.

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the SCAN function. Blanks are ignored. Use the following characters as modifiers:

a or A	adds alphabetic characters to the list of characters.
b or B	scans backward from right to left instead of from left to right, regardless of the sign of the *count* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters to the list of characters.
g or G	adds graphic characters to the list of characters. Graphic characters are characters that, when printed, produce an image on paper.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters.
k or K	causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.
l or L	adds lowercase letters to the list of characters.

m or M specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.

n or N adds digits, an underscore, and English letters to the list of characters.

o or O processes the *charlist* and *modifier* arguments only once, rather than every time the SCAN function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure can make SCAN run faster when you call it in a loop where the *character-list* and *modifier* arguments do not change. The O modifier applies separately to each instance of the SCAN function in your SAS code, and does *not* cause all instances of the SCAN function to use the same delimiters and modifiers.

p or P adds punctuation marks to the list of characters.

q or Q ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.

r or R removes leading and trailing blanks from the word that SCAN returns. If you specify the Q and R modifiers, the SCAN function first removes leading and trailing blanks from the word. Then, if the word begins with a quotation mark, SCAN also removes one layer of quotation marks from the word.

s or S adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T trims trailing blanks from the *string* and *charlist* arguments. If you want to remove trailing blanks from only one character argument instead of both character arguments, use the TRIM function instead of the SCAN function with the T modifier.

u or U adds uppercase letters to the list of characters.

w or W adds printable (writable) characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Restriction This argument is supported only on the SAS Viya platform and on the CAS server.

Tip If the *modifier* argument is a character constant, enclose the argument in quotation marks. Specify multiple modifiers in a single set of quotation marks. A *modifier* argument can also be expressed as a character variable or expression.

Details

Definitions of “Delimiter” and “Word”

A *delimiter* is any of several characters that are used to separate words. You can specify the delimiters in the *delimiter* and *modifier* arguments.

If you specify the Q modifier, delimiters inside substrings that are enclosed in quotation marks are ignored.

In the SCAN function, “word” refers to a substring that has all of these characteristics:

- It is bounded on the left by a delimiter or the beginning of the string.
- It is bounded on the right by a delimiter or the end of the string.
- It contains no delimiters.

A word can have a length of zero if there are delimiters at the beginning or end of the string, or if the string contains two or more consecutive delimiters. However, the SCAN function ignores words that have a length of zero unless you specify the M modifier.

Note: The definition of “word” is the same in the SCAN and COUNTW functions.

Using Default Delimiters in ASCII and EBCDIC Environments

If you use the SCAN function with only two arguments, then the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.

- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

If you use the *modifier* argument without specifying any characters as delimiters, then the only delimiters that are used are delimiters that are defined by the *modifier* argument. In this case, the lists of default delimiters for ASCII and EBCDIC environments are not used. In other words, modifiers add to the list of delimiters that are explicitly specified by the *delimiter* argument. Modifiers do not add to the list of default modifiers.

The Length of the Result

Leading delimiters before the first word in the expression do not affect SCAN. If there are two or more contiguous delimiters, SCAN treats them as one.

In DS2, if the SCAN function returns a value to a variable that has not yet been given a length, and then that variable is given the length of the first argument. If

you need the SCAN function to assign to a variable a value that is different from the length of the first argument, use a DECLARE statement for that variable before the statement that uses the SCAN function.

The minimum length of the word that is returned by the SCAN function depends on whether the M modifier is specified.

Using the SCAN Function with the M Modifier

If you specify the M modifier, the number of words in a string is defined as one plus the number of delimiters in the string. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you specify the M modifier, the SCAN function returns a word with a length of zero if one of these conditions is true:

- The string begins with a delimiter and you request the first word.
- The string ends with a delimiter and you request the last word.
- The string contains two consecutive delimiters and you request the word that is between the two delimiters.

Using the SCAN Function without the M Modifier

If you do not specify the M modifier, the number of words in a string is defined as the number of maximal substrings of consecutive non-delimiters. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you do not specify the M modifier, the SCAN function acts in these ways:

- It ignores delimiters at the beginning or end of the string.
- It treats two or more consecutive delimiters as if they were a single delimiter.

If the string contains no characters other than delimiters, or if you specify a count that is greater in absolute value than the number of words in the string, then the SCAN function returns one of the following items:

- a single blank when you call the SCAN function from a DATA step
- a string with a length of zero when you call the SCAN function from the macro processor

Using Null Arguments

The SCAN function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Processing SBCS and DBCS Data

The SCAN function is designed to process SBCS data, but it can also process DBCS data. Here are the criteria:

- If *expression* is not declared as VARCHAR and you are processing single-byte data, then SCAN processes SBCS.

- If *string* is declared as VARCHAR and you are processing multibyte data, then SCAN processes DBCS.

Example: Basic SCAN Function Usage

The following statements illustrate the SCAN function:

```
proc cas;
   expr='ABC.DEF(X=y)';
   word=scan(expr,3);
   print word;
run;
```

The following output is written to the SAS Log:

```
   X=y
```

SEC Function

Returns the secant.

Returned data type: DOUBLE

Syntax

SEC(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value that expressed in radians.

Restriction *expression* cannot be an odd multiple of PI/2.

Comparisons

The SEC function is related to the COS function:

```
sec(x)=1/cos(x)
```

Example

The following statements illustrate the SEC function:

```
proc cas;
   x=sec(0.5);
   print x;
   y=sec(0);
   print y;
   z=sec(3.14159/3);
   print z;
run;
```

The following output is printed to the SAS Log:

```
   1.1394939273
   1
   1.9999969359
```

SECOND Function

Returns the second from a SAS time or datetime value.

Returned data type: DOUBLE

Syntax

SECOND(*time* | *datetime*)

Arguments

time
specifies any valid expression that represents a SAS time value.

Data type DOUBLE

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

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.

Example

The following statements illustrate the SECOND function:

```
proc cas;
   a=hms(3,19,24);
   b=second(a);
   print a;
   print b;
run;
```

The following output is printed to the SAS Log:

```
   11964
   24
```

SIGN Function

Returns a number that indicates the sign of a numeric value expression.

Returned data type: DOUBLE

Syntax

SIGN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type All numeric types

Details

The SIGN function returns the following values:

-1
if *expression* < 0

0
if *expression* = 0

1
if *expression* > 0.

Example

The following statements illustrate the SIGN function:

```
proc cas;
   a=sign(-5);
   print a;
   a=sign(5);
   print a;
   a=sign(0);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   -1
   1
   0
```

SIN Function

Returns the trigonometric sine.

Returned data type: DOUBLE

Syntax

SIN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Example

The following statements illustrate the SIN function:

```
proc cas;
   a=sin(25.6);
   print a;
   a=sin(5);
   print a;
run;
```

The following output is printed to the SAS Log:

```
0.4504405943
-0.958924275
```

SINH Function

Returns the hyperbolic sine.

Returned data type: DOUBLE

Syntax

SINH(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

The SINH function returns the hyperbolic sine of the argument, which is given by the following equation.

$$e^{argument} - e^{-argument}/2$$

Example

The following statements illustrate the SINH function:

```
proc cas;
   a=sinh(0);
   print a;
   a=sinh(1);
   print a;
   a=sinh(-1.0);
   print a;
run;
```

The following output is printed to the SAS Log:

```
0
```

```
1.1752011936
-1.175201194
```

SKEWNESS Function

Returns the skewness.

Returned data type: DOUBLE

Syntax

SKEWNESS(*expression-1, expression-2, expression-3* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least three non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

Details

If all non-null or nonmissing arguments have equal values, the skewness is mathematically undefined and the SKEWNESS function returns a null or missing value.

Example

The following statements illustrate the SKEWNESS function:

```
proc cas;
   x1=skewness(0,1,1);
   print x1;
   x2=skewness(2,4,6,3,1);
   print x2;
   x3=skewness(2,0,0);
   print x3;
run;
```

The following output is printed to the SAS Log:

```
-1.732050808
0.5901286564
1.7320508076
```

SLEEP Function

For a specified period of time, suspends the execution of a program that invokes this function.

Returned data type: DOUBLE

Syntax

SLEEP(*number-of-time-units* <, *time-unit*>)

Arguments

number-of-time-units
specifies any valid expression that evaluates to a numeric value and that specifies the number of units of time for which you want to suspend execution of a program.

Range $n \geq 0$

Data type DOUBLE

time-unit
specifies the unit of time, as a power of 10, which is applied to *number-of-time-units*. For example, 1 corresponds to a second, and .001 to a millisecond.

Default 1 in a Windows PC environment, .001 in other environments

Data type DOUBLE

Details

The SLEEP function suspends the execution of a program that invokes this function for a period of time that you specify. The maximum sleep period for the SLEEP function is 46 days.

Examples

Example 1: Suspending Execution for a Specified Period of Time

The following example delays the execution for 20 seconds:

```
proc cas;
   ...CASL statements...
   time_slept=sleep(20,1);
   ...more CASL statements...
   print "time_slept=" time_slept;
run;
```

Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following example tells SAS to suspend the execution until June 15, 2011, at midnight. DS2 calculates the length of the suspension based on the target date and the date and time that the code begins to execute.

```
   proc cas;
      ...CASL statements...;
      sleeptime=dhms(mdy(06,15,2011),00,00,00)-datetime();
      time_calc=sleep(sleeptime,1);
      ...more CASL statements...;
      print "time_calc=" time_calc;
   run;
```

SMALLEST Function

Returns the *k*th smallest non-null or nonmissing value.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

SMALLEST(*k*, *expression* <, ...*expression*>)

Arguments

k

specifies any valid expression that evaluates to a numeric value to return.

Data type DOUBLE

expression
specifies any valid expression that evaluates to a numeric value to be processed.

Data type DOUBLE

Details

If *k* is null or missing, less than zero, or greater than the number of values, the result is a null or missing value.

Comparisons

The SMALLEST function differs from the ORDINAL function in that SMALLEST ignores null and missing values, but ORDINAL counts null and missing values.

Example

This example compares the values that are returned by the SMALLEST function with the values that are returned by the ORDINAL function.

```
proc cas;
   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;
```

SQRT Function

Returns the square root of a value.

Returned data type: DOUBLE

Syntax

SQRT(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a nonnegative numeric value.

Data type DOUBLE

Example

The following statements illustrate the SQRT function:

```
proc cas;
      a=sqrt(36);
      print a;
      a=sqrt(25);
      print a;
      a=sqrt(4.4);
      print a;
run;
```

The following output is printed to the SAS Log:

```
   6
   5
   2.097617696
```

STD Function

Returns the standard deviation.

Returned data type: DOUBLE

Syntax

STD(*expression-1, expression-2* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

Example

The following statements illustrate the STD function:

```
proc cas;
      a=std(2,6);
      print a;
      a=std(2,6,.);
      print a;
      a=std(2,4,6,3,1);
      print a;
run;
```

The following output is printed to the SAS Log:

```
2.8284271247
2.8284271247
1.9235384062
```

STDERR Function

Returns the standard error of the mean.

Categories: CAS
Descriptive Statistics

Returned data type: DOUBLE

Syntax

STDERR(*expression-1, expression-2* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

Example

The following examples illustrate the STDERR function.

```
proc cas;
      a=stderr(2,6);
      print a;
      a=stderr(2,6,.);
      print a;
      a=stderr(2,4,6,3,1);
      print a;
run;
```

The following output is printed to the SAS Log:

```
2
2
0.8602325267
```

STRIP Function

Returns a character string with all leading and trailing blanks removed.

Returned data type: STRING, VARCHAR

Syntax

STRIP(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The STRIP function returns the argument with all leading and trailing blanks removed. If the argument is blank, STRIP returns a string with a length of zero.

If the value that is trimmed is shorter than the length of the receiving variable, SAS pads the value with new trailing blanks.

Note: The STRIP function is useful for concatenation because the concatenation operator does not remove trailing blanks.

Comparisons

The following list compares the STRIP function with the TRIM function:

- For blank character strings, the STRIP and TRIM functions both return a string with a length of zero.
- For strings that lack leading blanks, the STRIP and TRIM functions return the same value.

Example: Deleting the Leading and Trailing Blanks Using the STRIP Function in CAS

This example uses the DATA step to delete leading and trailing blanks. The first section of code in this example runs in SAS and creates the data set, lengthn, with the CAS engine. The second section of code runs in CAS, accesses the data in lengthn that is in CAS, and then creates the lengthnn data set.

```
cas casauto;
libname casuser cas caslib="casuser";

data casuser.lengthn;
input string $char8.;
datalines;
  abcd
    abcd
abcdefgh
x y z
;
data casuser.lengthnn;
set casuser.lengthn;
original='*' || string || '*';
stripped='*' || strip(string) || '*';
run;

proc print data=casuser.lengthnn;
run;
```

Output 4.8 *Results from the STRIP Function*

Obs	string	original	stripped
1	abcd	* abcd *	*abcd*
2	abcd	* abcd*	*abcd*
3	abcdefgh	*abcdefgh*	*abcdefgh*
4	x y z	*x y z *	*x y z*

SUBSTR (right of =) Function

Returns a substring from an argument.

Returned data type: STRING, VARCHAR

Note: The SUBSTR function (right of =) is supported in CASL. However, SUBSTR (left of =) is not supported in CASL. If you attempt to use SUBSTR on the left side of an equal sign in CASL, you will receive an error message: `ERROR: Cannot resolve variable for Assignment`. There is currently no equivalent to SUBSTR (left of =) in CASL.

Syntax

SUBSTR(*string*, *position* <, *length*>)

Required Arguments

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

Note: If you specify *string* as a character constant, you must place quotation marks around the value. If you specify string as either a character variable or a character expression, the values should not be placed in quotation marks.

See "Example 1: Specifying the String Value in the SUBSTR= Function"

position
specifies the beginning position of the substring being extracted. *position* can be a numeric constant, a numeric variable, or a numeric expression.

See "Example 2: Specifying the Position Value in the SUBSTR= Function"

Optional Argument

length
specifies the length of the substring being extracted. *length* can be a numeric constant, a numeric variable, or a numeric expression.

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 prints a WARNING to the log indicating that the *length* argument is invalid.

Tip If you omit *length*, SAS extracts the remainder of the expression. See the example "Specifying the Position Value in the SUBSTR= Function: Specify position as a numeric constant".

See "Example 3: Specifying the Length Value in the SUBSTR= Function"

Examples

Example 1: Specifying the String Value in the SUBSTR= Function

Specify *string* as a character constant.

```
proc cas;
   zcode=substr("zip28413",4,5);
   print zcode;
run;
```

The following output is printed to the SAS Log:

```
   28413
```

Specify *string* as a character variable.

```
proc cas;
   z="zip28413";
   zcode=substr(z,4);
   print zcode;
run;
```

The following output is printed to the SAS Log:

```
   28413
```

Specify *string* as a character expression.

```
proc cas;
   z="zip";
   code="28413";
   zcode=substr(z||code,4,5);
   print zcode;
run;
```

The following output is printed to the SAS Log:

```
   28413
```

Example 2: Specifying the Position Value in the SUBSTR= Function

Specify *position* as a numeric constant.

```
proc cas;
   name=substr("nameaddress",1,4);
   print name;
run;
```

The following output is printed to the SAS Log:

```
   name
```

Specify *position* as a numeric variable.

```
proc cas;
   position=1;
   name=substr("nameaddress",position,4);
   print name;
run;
```

The following output is printed to the SAS Log:

```
   name
```

Specify *position* as a numeric expression.

```
proc cas;
   position=1+4;
   address=substr("name123Main",position,7);
   print address;
run;
```

The following output is printed to the SAS Log:

```
   123Main
```

Example 3: Specifying the Length Value in the SUBSTR= Function

Specify *length* as a numeric constant.

```
proc cas;
   car="make##model";
   type=substr(car,7,5);
   print type;
run;
```

The following output is printed to the SAS Log:

```
   model
```

Specify *length* as a numeric variable.

```
proc cas;
   car="make##model";
   length=5;
   position=7;
   model=substr(car,position,length);
   print model;
run;
```

The following output is printed to the SAS Log:

```
   model
```

Specify *length* as a numeric expression.

```
proc cas;
   length=1+4;
   model=substr("make##model",7,length);
   print model;
run;
```

The following output is printed to the SAS Log:

```
   model
```

Example 4: Specifying Invalid Length Values

***length* is set to zero.**

```
proc cas;
   car="make##model";
   type=substr(car,7,0);
   print type;
run;
```

The following output is printed to the SAS Log:

```
   WARNING: Argument 3 is an invalid argument.
   model
```

***length* is set to a negative integer.**

```
proc cas;
   car="make##model";
   type=substr(car,7,-5);
   print type;
run;
```

The following output is printed to the SAS Log:

```
   WARNING: Argument 3 is an invalid argument.
   model
```

***length* is longer than the length of the *string* after the *position* value is applied.**

```
proc cas;
   car="make##model";
   type=substr(car,7,9);
   print type;
run;
```

The following output is printed to the SAS Log:

```
   WARNING: Argument 3 is an invalid argument.
   model
```

SUBSTRN Function

Returns a substring, allowing a result with a length of zero.

Returned data type: STRING, VARCHAR

Syntax

SUBSTRN(*character-expression*, *position*<, *length*>)

Required Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string or numeric value.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Note: If *expression* 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.

Restrictions: If *position* is missing a blank space is returned, a warning is written to the SAS log.

If *position* is past the end of the string, an empty string is returned.

Notes: 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 sum of *position* and *length* is greater than the length of the string, the position is returned until the end of the string.

If the sum of *position* and *length* is less than 0, an empty string is returned.

Optional Argument

length
is an integer that specifies the length of the substring.

Restrictions 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.

If *length* is negative, the SUBSTRN function returns an empty string.

Note 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.

Comparisons

The following table lists comparisons between the SUBSTRN and the SUBSTR functions:

Table 4.4 *Comparison: 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 warning to the SAS log and returns a blank 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 warning to the SAS log and returns a blank string
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 warning to the SAS log and returns a blank string

Examples

Example 1: Comparison between the SUBSTR and SUBSTRN Functions

The following example compares the results of using the SUBSTR function and the SUBSTRN function when the first argument is numeric.

Example Code 4.1 *SUBSTR Example*

```
proc cas;
substr_result="*" || substr(1234.5678,2,6) || "*";
print substr_result;
run;
```

Example Code 4.2 *SUBSTRN Example*

```
substrn_result="*" || substrn(1234.5678,2,6)  || "*";
print substrn_result;
run;
```

SAS writes the following output to the log:

Example Code 4.2 *Results of the SUBSTR Example*

```
WARNING: Argument 1 should be a string.
* *
```

Example Code 4.3 *Results of the SUBSTRN Example*

```
*234.56*
```

Example 2: Manipulating Strings with the SUBSTRN Function

This example shows how to manipulate strings with the SUBSTRN function.

```
proc cas;
columns = {"Position", "Length", "Result"};
coltypes={"integer", "integer", "string"};
table = newtable("Test", columns, coltypes);

xyz="abcd";
do position=-1 to 6;
   do length=max(-1,-position) to 7-position;
   substrn_result="*" || substrn(xyz, position, length) || "*";
   addrow(table,{position, length, substrn_result});
   end;
end;
run;
print table;
quit;
```

Output 4.9 *Partial Output from the SUBSTRN Function*

table: Results

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*

See Also

"SUBSTR (right of =) Function" on page 624

SUM Function

Returns the sum of the non-null or nonmissing arguments.

Returned data type: DOUBLE

Syntax

SUM(*expression-1*, *expression-2* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least two arguments are required.

Data type: BIGINT, DECIMAL, DOUBLE

Details

Null and missing values are ignored and not included in the computation. If all of the arguments have missing values, the result is a missing value. If all the arguments have a null value, the result is a null value.

If any argument to this function is non-numeric, the argument is converted to DOUBLE. If any argument is DOUBLE or REAL, all arguments are converted to DOUBLE (if not so already) and the result is DOUBLE. Otherwise, if any argument is DECIMAL, all arguments are converted to DECIMAL (if not so already) and the result is DECIMAL. Otherwise, all arguments are converted to a BIGINT and the result is BIGINT.

Example

The following statements illustrate the SUM function.

```
proc cas;
      a=sum(4,9,3,8);
      print a;
      a=sum(4,9,3,8,.);
      print a;
run;
```

The following output is printed to the SAS Log:

```
   24
   24
```

SUMABS Function

Returns the sum of the absolute values of the nonmissing arguments.

Returned data type: DOUBLE

Syntax

SUMABS(*value-1*<, ...*value-n*>)

Arguments

value
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

Details

If all arguments have null or missing values, then the result is a null or missing value. Otherwise, the result is the sum of the absolute values of the nonmissing values.

Examples

Example 1: Calculating the Sum of Absolute Values

The following example returns the sum of the absolute values of the nonmissing arguments.

```
proc cas;
  x=sumabs(1,.,-2,0,3,.q,-4);
  print "x=" x;
run;
```

SAS writes the following results to the log:

```
x=10
```

Example 2: Calculating the Sum of Absolute Values When You Use a Variable List

The following example uses a variable list and returns the sum of the absolute value of the nonmissing arguments.

```
proc cas;
  x1=1;
  x2=3;
  x3=4;
  x4=3;
  x5=1;
  x = sumabs(x1, x2, x3, x4, x5);
  print "x=" x;
run;
```

SAS writes the following results to the log:

```
x=12
```

TAN Function

Returns the tangent.

Returned data type: DOUBLE

Syntax

TAN(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value in radians.

Restriction *expression* cannot be an odd multiple of π /2

Data type DOUBLE

Example

The following statements illustrate the TAN function:

```
proc cas;
      a=tan(0.5);
      print a;
      a=tan(0);
      print a;
      a=tan(3.14159/3);
      print a;
run;
```

The following output is printed to the SAS Log:

```
0.5463024898
0
1.7320472695
```

TANH Function

Returns the hyperbolic tangent.

Returned data type: DOUBLE

Syntax

TANH(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction *expression* cannot be an odd multiple of π/2

Data type DOUBLE

Details

The TANH function returns the hyperbolic tangent of the argument, which is given by the following equation.

$$\frac{e^{argument} - e^{-argument}}{e^{argument} + e^{-argument}}$$

Example

The following statements illustrate the TANH function.

```
proc cas;
      a=tanh(0.5);
      print a;
      a=tanh(0);
      print a;
      a=tanh(3.14159/3);
      print a;
run;
```

The following output is printed to the SAS Log:

```
0.4621171573
0
0.78071409
```

TIME Function

Returns the current time of day as a numeric SAS time value.

Returned data type: DOUBLE

Syntax

TIME( )

Details

The TIME function does not take any arguments. It produces the current time in the form of a SAS time value.

Example

The following statements illustrate the TIME function:

```
proc cas;
      a=time();
      print a;
      a=put(time(),time.);
      print a;
run;
```

The following output is printed to the SAS Log:

```
54227.102736
15:03:47
```

TIMEPART Function

Extracts a time value from a SAS datetime value.

Returned data type: DOUBLE

Syntax

TIMEPART(*datetime*)

Arguments

datetime
specifies any valid expression that represents a SAS datetime value.

Data type DOUBLE

Example

The following statements illustrate the TIMEPART function:

```
proc cas;
  ddtm=datetime();
  tm=put(timepart(ddtm),time.);
  print "ddtm=" ddtm;
  print "tm=" tm;
run;
```

The following output is printed to the SAS Log:

```
ddtm=2045142595.8
tm=15:09:56
```

TIMEVALUE Function

Returns the equivalent of a reference amount at a base date by using variable interest rates.

Returned data type: DOUBLE

Syntax

TIMEVALUE(*base-date*, *reference-date*, *reference-amount*, *compounding-interval*, *date-1*, *rate-1*<, ...*date-n*, *rate-n*>)

Arguments

base-date
specifies the time value of the *reference-amount* at the *base-date*.

Requirement *Base-date* is a SAS date.

Data type DOUBLE

reference-date
specifies the date of *reference-amount*.

Requirement *Reference-date* is a SAS date.

Data type DOUBLE

reference-amount
specifies the amount at the *reference-date*.

Data type DOUBLE

compounding-interval
specifies the compounding interval.

Requirement *Compounding-interval* is a SAS interval.

Data type CHAR

date
specifies the time at which *rate* takes effect. Each date is paired with a rate.

Requirement *Date* is a SAS date.

Data type DOUBLE

rate
specifies the interest rate as numeric percentage that starts on *date*. Each rate is paired with a date.

Data type DOUBLE

Details

The following details apply to the TIMEVALUE function:

- The values for rates must be between –99 and 120.
- The list of date-rate pairs does not need to be sorted by date.
- When multiple rate changes occur on a single date, the TIMEVALUE function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the *reference-date* and the *base-date*.

Example

- You can express the accumulated value of an investment of $1,000 at a nominal interest rate of 10% compounded monthly for one year as the following:

```
proc cas;
   bd=16437;
   rd=14612;
   d=14612;
   amount_base1=timevalue(bd, rd, 1000, 'month', d, 10);
   print "amount_base1=" amount_base1;
run;
```

- If the interest rate jumps to 20% halfway through the year, the resulting calculation would be as follows:

```
proc cas;
    bd=16437;
    rd=14610;
    d1=14610;
    d2=14976;
    amount_base2 = timevalue(bd, rd, 1000, 'month', d1, 10, d2, 20);
    print "amount_base2=" amount_base2;
run;
```

- 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:

```
proc cas;
    bd=16437;
    rd=14610;
    d1=14610;
    d2=14910;
    amount_base3 = timevalue(bd, rd, 1000, 'month', d1, 10, d2, 20);
    print "amount_base3=" amount_base3;
run;
```

TINV Function

Returns a quantile from the *t* distribution.

Returned data type: DOUBLE

Syntax

TINV(*p*, *df*<, *nc*>)

Arguments

p

specifies any valid expression that evaluates to a numeric probability.

Range $0 < p < 1$

Data type DOUBLE

df

specifies any valid expression that evaluates to a numeric degrees of freedom parameter.

Range $df > 0$

Data type DOUBLE

nc

specifies any valid expression that evaluates to a numeric noncentrality parameter.

Data type DOUBLE

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.

Comparisons

TINV is the inverse of the PROBT function.

Example

The following statements illustrate the TINV function:

```
proc cas;
  x=tinv(.95, 2);
```

```
  print x;
  x=tinv(.95, 2.5, 3);
  print x;
run;
```

The following output is printed to the SAS Log:

```
   2.9199855804
   11.033833625
```

TODAY Function

Returns the current date as a numeric SAS date value.

Returned data type: DOUBLE

Syntax

TODAY( )

Details

The TODAY function does not take any arguments. It produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

Example

The following statement illustrates the TODAY function:

```
proc cas;
   td=today();
   tday=put(td, date.);
   print "td=" td;
   print "tday=" tday;
run;
```

The following output is written to the SAS Log:

```
td=23664
tday=15OCT24
```

TRANSLATE Function

Replaces specific characters in a character expression.

Returned data type: STRING, VARCHAR

Syntax

TRANSLATE(*expression*, *to-characters*, *from-characters*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string. *expression* contains the original character value.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

to-characters
specifies the characters that you want TRANSLATE to use as substitutes.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

from-characters
specifies the characters that you want TRANSLATE to replace.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Values of *to-characters* and *from-characters* correspond on a character-by-character basis; TRANSLATE changes the first character in *from-characters* to the first character in *to-characters*, and so on. If *to-characters* has fewer characters than *from-characters*, TRANSLATE changes the extra *from-characters* characters to blanks. If *to-characters* has more characters than *from-characters*, TRANSLATE ignores the extra *to-characters*.

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this scan, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANWRD function differs from TRANSLATE in that it scans for words (or patterns of characters) and replaces those words with a second word (or pattern of characters).

Example

The following statement illustrates the TRANSLATE function.

```
proc cas;
   a=translate('XYZW', 'AB', 'VW');
   print "a=" a;
run;
```

The following output is written to the SAS Log:

```
a=XYZB
```

```
proc cas;
   string1='AABBAABABB';
   a=translate(string1, '12', 'AB');
   print "a=" a;
run;
```

The following output is written to the SAS Log:

```
a=1122112122
```

TRANSTRN Function

Replaces or removes all occurrences of a substring in a character string.

Returned data type: STRING, VARCHAR

Syntax

TRANSTRN(*source-expression*, *target-expression*, *replacement-expression*)

Arguments

source-expression
specifies any valid expression that evaluates or can be coerced to a character string and whose characters you want to translate.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

target-expression
specifies any valid expression that evaluates or can be coerced to a character string and whose characters are searched for in *source-expression*.

Requirement The length for *target-expression* must be greater than zero.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

replacement-expression
specifies any valid expression that evaluates or can be coerced to a character string and that replaces *target-expression*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The TRANSTRN function replaces or removes all occurrences of a given substring within a character string. The TRANSTRN function does not remove trailing blanks in the *target-expression* string and the *replacement-expression* string. To remove all occurrences of *target*, specify *replacement-expression* as TRIMN(' ').

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. In doing this scan, however, TRANSLATE searches for every occurrence of any of the individual characters within a string. That is, if any letter (or character) in the target string is found in the source string, it is replaced with the corresponding letter (or character) in the replacement string.

The TRANSTRN function differs from TRANSLATE in that TRANSTRN scans for substrings and replaces those substrings with a second substring.

Examples

Example 1: Replacing All Occurrences of a Word

In this example, the TRANSTRN function is used to replace **Mrs.** and **Miss** with **Ms.**

```
proc cas;
    name1='Mrs.  Joan Smith';
    nameA=transtrn(name1, 'Mrs.', 'Ms.');
    print "nameA=" nameA;
    name2='Miss Alice Cooper';
    nameB=transtrn(name2, 'Miss', 'Ms.');
    print "nameB=" nameB;
run;
```

The following lines are written to the SAS log:

```
nameA=Ms.  Joan Smith
nameB=Ms. Alice Cooper
```

Example 2: Removing Blanks from the Search String

In this example, the TRIM function is used with *target* to exclude blanks. If you did not include the TRIM function, the TRANSTRN function would not replace the source string because the target string contains blanks.

```
proc cas;
    salelist='CATFISH';
    target='FISH';
    replacement='NIP';
    salelist2=transtrn(salelist, trim(target), replacement);
    print "salelist2=" salelist2;
run;
```

The following is written to the SAS log:

```
salelist2=CATNIP
```

Example 3: Zero Length in the Third Argument of the TRANSTRN Function

The following example shows the results of the TRANSTRN function when the third argument, *replacement*, has a length of zero. In DS2, a character constant that consists of two quotation marks with a blank in between them represents a single blank, and not a zero-length string. In the following example, the results for *string1* are different from the results for *string2*.

```
proc cas;
      string1='*' || transtrn('abcxabc', 'abc', trimn(' ')) || '*';
      print "string1=" string1;
      string2='*' || transtrn('abcxabc', 'abc', ' ') || '*';
      print "string2=" string2;
run;
```

SAS writes the following output to the log:

```
string1=*x*
string2=* x *
```

TRIGAMMA Function

Returns the value of the trigamma function.

Returned data type: DOUBLE

Syntax

TRIGAMMA(*expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Restriction: Nonpositive integers are invalid.

Data type: DOUBLE

Details

The TRIGAMMA function returns the derivative of the digamma function. For *expression* > 0, the TRIGAMMA function is the second derivative of the lgamma function.

Example

The following statement illustrates the TRIGAMMA function:

```
proc cas;
  a=trigamma(3);
  print a;
run;
```

The following output is printed to the SAS Log:

```
   0.3949340668
```

TRIM Function

Removes trailing blanks from a character expression, and returns a string with a length of zero if the expression is missing.

Alias: TRIMN

Returned data type: STRING, VARCHAR

Syntax

TRIM('*expression*')

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The TRIM function copies a character argument, removes trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIM returns a string with a length of zero. TRIM is useful after concatenating because concatenation does not remove trailing blanks.

Note: The TRIM function removes both blanks and whitespace characters as defined by the Unicode standard. Consequently, the TRIM function also handle DBCS blanks and shift out/shift in (SO/SI) escape codes. For more information about the Unicode whitespace character standard, see Wikipedia: Unicode character property.

Example

The following statements illustrate the TRIM function:

```
proc cas;
  string1='Testscore   ';
  string2='File.xls';
```

```
  result=trim(string1)||(string2);
  print "result=" result;
run;
```

The following output is written to the SAS Log:

```
   result=TestscoreFile.xls
```

TRIMN Function

Removes trailing blanks from character expressions, and returns a string with a length of zero if the expression is missing.

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

```
proc
cas;

   x="A"||
trimn("")||"B";

   z="
";

   y=">"||
trimn(z)||"<";

   print "x=" x;
   print "y="
y;

run;
```

These statements produce this result:

```
x=AB
y=><
```

TRUNC Function

Truncates a numeric value to a specified length.

Returned data type: DOUBLE

Syntax

TRUNC(*expression*, *length-expression*)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

length-expression
specifies any valid expression that evaluates to a numeric value.

Range 3–8

Data type DOUBLE

Details

The TRUNC function truncates a full-length numeric expression (stored as a DOUBLE) to a smaller number of bytes, as specified in *length-expression* and pads the truncated bytes with 0s. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them.

Example

The following statements illustrate the TRUNC function:

```
proc cas;
   x=trunc(3.1,3);
   print x;
   x=trunc(3.1,4);
   print x;
   x=trunc(3.1,5);
   print x;
   x=trunc(3.1,6);
   print x;
   x=trunc(3.1,7);
   print x;
   x=trunc(3.1,8);
   print x;
run;
```

The following output is printed to the SAS Log:

```
   3.099609375
   3.0999984741
   3.099999994
   3.1
   3.1
   3.1
```

UNIFORM Function

Returns a random variate from a uniform distribution.

Categories: Random Number
CAS

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" in *SAS Functions and CALL Routines: Reference* "RAND Function" in *SAS Functions and CALL Routines: Reference* "CALL STREAMINIT Routine" in *SAS Functions and CALL Routines: Reference*

UPCASE Function

Converts all letters in an argument to uppercase.

Alias: UPPER

Returned data type: STRING, VARCHAR

Syntax

UPCASE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR

Details

The UPCASE function copies a character expression, converts all lowercase letters to uppercase letters, and returns the altered value as a result.

Comparisons

The LOWCASE function converts all letters in an argument to lowercase letters. The UPCASE function converts all letters in an argument to uppercase letters.

Example

The following statement illustrates the UPCASE function:

```
proc cas;
   name=upcase('John B. Smith');
   print name;
run;
```

The following output is written to the SAS Log:

```
   JOHN B. SMITH
```

USS Function

Returns the uncorrected sum of squares.

Returned data type: DOUBLE

Syntax

USS(*expression* <, ...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement: At least one non-null or nonmissing argument is required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

Example

The following statements illustrate the USS function:

```
proc cas;
   a=uss(4,2,3.5,6);
   print a;
   a=uss(4,2,3.5,6,.);
   print a;
run;
```

The following output is printed to the SAS Log:

```
   68.25
```

```
68.25
```

UUIDGEN Function

Returns the short form of a Universally Unique Identifier (UUID).

Returned data type: STRING, VARCHAR

Syntax

UUIDGEN( )

Without Arguments

The UUIDGEN function has no arguments.

Details

The UUIDGEN function returns a UUID (a unique value) for each call. The default result is 36 characters long and it looks like this:

```
5ab6fa40-426b-4375-bb22-2d0291f43319
```

Example

The following example returns a UUID. Note that a variable declaration of 36 characters is required.

```
proc cas;
   x=uuidgen();
   print x;
run;
```

The following value is written to the SAS log. Each UUID is unique.

```
25C752D5-AFA1-4932-BEE6-39E4006C2AAB
```

VAR Function

Returns the variance.

Returned data type: DOUBLE

Syntax

VAR(*expression-1*, *expression-2* < ,...*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value. The argument list can consist of a variable list.

Requirement: At least two non-null or nonmissing arguments are required. Otherwise, the function returns a null or missing value.

Data type: DOUBLE

Example

The following example uses the VAR function and returns the variance.

```
proc cas;
  val1=4;
  val2=2;
  val3=3.5;
  val4=6;
  x1=var(val1, val2, val3);
  x2=var(val1, val4);
  x3=var(x1, x2);
  print "x1=" x1;
  print "x2=" x2;
  print "x3=" x3;
run;
```

The following is printed to the SAS log:

```
x1=1.0833333333
x2=2
x3=0.4201388889
```

VERIFY Function

Returns the position of the first character that is unique to an expression.

Returned data type: DOUBLE

Syntax

VERIFY(*target-expression*, *search–expression*)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string that is to be searched.

Requirement: Literal character strings must be enclosed in single quotation marks.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

search-expression
specifies any valid expression that evaluates or can be coerced to a character string.

Requirement: Literal character strings must be enclosed in single quotation marks.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The VERIFY function returns the position of the first character in *target-expression* that is not present in *search-expression*. If there are no characters in *target-expression* that are unique from those in *search-expression*, VERIFY returns a 0.

Comparisons

The INDEX function returns the position of the first occurrence of *search-expression* that is present in *target-expression* where the VERIFY function returns the position of the first character in *target-expression* that does not contain *search-expression*.

Example

The following statements illustrate the VERIFY function:

```
proc cas;
  x='abc';
  y='abcdef';
  z=verify(y,x);
  print "z=" z;
```

```
run;
```

The following output is printed to the SAS Log:

```
   z=4
```

```
proc cas;
  x='abcdef';
  y='abcdef';
  z=verify(y,x);
  print "z=" z;
run;
```

The following output is printed to the SAS Log:

```
   z=0
```

WEEK Function

Returns the week-number value.

Returned data type: DOUBLE

Syntax

WEEK(<*sas-date*>, <'*descriptor*'>)

Arguments

sas-date
specifies the SAS date value. If the *sas-date* argument is not specified, the WEEK function returns the week-number value of the current date.

Data type DOUBLE

descriptor
specifies the value of the descriptor. The following descriptors can be specified in uppercase or lowercase characters.

U
specifies the number-of-the-week within the year. Sunday is considered the first day of the week. The number-of-the-week value is represented as a decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'u')` is 53. U is the default value.

Tip The U and W descriptors are similar, except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

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.

W

specifies the number-of-the-week within the year. Monday is considered the first day of the week. The number-of-the-week value is represented as a decimal number in the range 0–53. Week 53 has no special meaning. The value of `week('31dec2013'd, 'w')` is 53.

Tip The U and W descriptors are similar except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

Default U

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The Basics

The WEEK function reads a SAS date value and returns the week number. The WEEK function is not dependent on locale, and uses only the Gregorian calendar in its computations.

The U Descriptor

The WEEK function with the U descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year is represented as 05.

Sunday is considered the first day of the week. For example, the value of `week('01jan2013'd, 'u')` is 0.

The V Descriptor

The WEEK function with the V descriptor reads a SAS date value and returns the week number. The number-of-the-week is represented as a decimal number in the range 01–53. The decimal number has a leading zero and a maximum value of 53. Weeks begin on a Monday, and week 1 of the year is the week that includes both January 4 and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. In the following example, 01jan2014 and 31dec2013 occur in the same week. The first day (Monday) of that week is 30dec2013. Therefore,

`week('01jan2014'd, 'v')` and `week('30dec2013'd, 'v')` both return a value of 53. This means that both dates occur in week 53 of the year 2013.

The W Descriptor

The WEEK function with the W descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year would be represented as 05.

Monday is considered the first day of the week. Therefore, the value of `week('30dec2013'd, 'w')` is 1.

Comparisons of Descriptors

U is the default descriptor. Its range is 0-53, and the first day of the week is Sunday. The V descriptor has a range of 1-53 and the first day of the week is Monday. The W descriptor has a range of 0-53 and the first day of the week is Monday.

The following list describes the descriptors and an associated week:

- Week 0:

  U indicates the days in the current Gregorian year before week 1.

  V does not apply.

  W indicates the days in the current Gregorian year before week 1.

- Week 1:

  U begins on the first Sunday in a Gregorian year.

  V begins on the Monday between December 29 of the previous Gregorian year and January 4 of the current Gregorian year. The first ISO week can span the previous and current Gregorian years.

  W begins on the first Monday in a Gregorian year.

- End of Year Weeks:

  U specifies that the last week (52 or 53) in the year can contain less than 7 days. A Sunday to Saturday period that spans 2 consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

  V specifies that the last week (52 or 53) of the ISO year contains 7 days. However, the last week of the ISO year can span the current Gregorian and next Gregorian year.

  W specifies that the last week (52 or 53) in the year can contain less than 7 days. A Monday to Sunday period that spans two consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

Example

The following example shows the values of the U, V, and W descriptors for the date May 12, 2013.

```
proc cas;
  sasdate=(19490);
  x=week(sasdate, 'u');
  y=week(sasdate, 'v');
  z=week(sasdate, 'w');
  print "x=" x;
  print "y=" y;
  print "z=" z;
run;
```

The following lines are written to the SAS log.

```
 x=19
 y=19
 z=18
```

WEEKDAY Function

From a SAS date value, returns a whole number that corresponds to the day of the week.

Returned data type: DOUBLE

Syntax

WEEKDAY(*expression*)

Arguments

expression
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

Details

The WEEKDAY function produces a whole number that represents the day of the week, where 1 = Sunday, 2 = Monday, ..., 7 = Saturday.

Example

The following statement illustrates the WEEKDAY function when the current day is Sunday:

```
proc cas;
  x=weekday(today());
  print x;
run;
```

The following output is printed to the SAS Log:

```
   2
```

WHICHC Function

Returns the first position of a character string from a list of character strings.

Returned data type: DOUBLE

Syntax

WHICHC(*search-expression*, *expression-list-item-1*, *expression-list-item-2* <, ...*expression-list-item-n*>)

Arguments

search-expression
specifies any valid expression that evaluates or can be coerced to a character string that is compared with a list of character string expressions.

Requirement: Literal character strings must be enclosed in single quotation marks.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

expression-list-item
specifies any valid expression that evaluates or can be coerced to a character string and that is a member of a list of character string expressions.

Requirements: Literal character strings must be enclosed in single quotation marks.

At least two expressions are required in the list.

Data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The WHICHC function searches the character expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHC returns its position in the expression list. If none of the expressions match the search expression, WHICHC returns a value of 0.

Example

In the following example, 'Spain' appears twice in the list. The WHICHC function return the first position of 'Spain' in the list:

```
proc cas;
  x=whichc('Spain', 'Denmark', 'Germany', 'Austria',
           'Spain', 'China', 'Egypt', 'Spain', 'France');
  print x;
run;
```

The following output is printed to the SAS Log:

```
   4
```

WHICHN Function

Returns the first position of a number from a list of numbers.

Returned data type: DOUBLE

Syntax

WHICHN(*search-expression*, *expression-list-item-1*, *expression-list-item-2* <, ...*expression-list-item-n*>)

Arguments

search-expression
specifies any valid expression that evaluates to a number and that is compared with a list of numeric expressions.

Data type DOUBLE

expression-list-item
specifies any valid expression that evaluates to a number and is part of a list.

Requirement At least two expressions are required in the list.

Data type DOUBLE

Details

The WHICHN function searches the numeric expression list, from left to right, for the first expression that matches the search expression. If a match is found, WHICHN returns its position in the expression list. If none of the expressions match the search expression, WHICHN returns a value of 0. Arguments for the WHICHN functions can be any numeric data type.

Example

In the following example, 4.5 appears two times in the list. The WHICHN function return the first position of 4.5 in the list.

```
proc cas;
  x=whichn(4.5,7.3, 8.6, 4.5, 4.5, 2.1, 6.4);
  print x;
run;
```

The following output is printed to the SAS Log:

```
   3
```

YEAR Function

Returns the year from a SAS date value.

Returned data type: DOUBLE

Syntax

YEAR(*date*)

Arguments

date
specifies any valid expression that represents a SAS date value.

Data type DOUBLE

Details

The YEAR function produces a four-digit numeric value that represents the year.

Example

```
proc cas;
   theDate=inputn('16JUL1969','DATE9.');
   theDay=day(theDate);
   theMonth=Month(theDate);
   theYear=year(theDate);
   print(NOTE) "Date=" putn(theDate,'date9.');
   print(NOTE) "Day=" theDay ' Month=' theMonth 'Year=' theYear;
```

The program writes the following output to the log:

```
NOTE: Date=16JUL1969
NOTE: Day=16  Month=7 Year=1969
```

YIELDP Function

Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.

Returned data type: DOUBLE

Syntax

YIELDP(*A*, *c*, *n*, *K*, k_0, *p*)

Arguments

A

specifies the face value.

Ranges $A > 0$

$A > 0$

Data type DOUBLE

c

specifies the nominal annual coupon rate, expressed as a fraction.

Ranges $0 \leq c < 1$

$0 \le c < 1$

Data type DOUBLE

n

specifies the number of coupons per year.

Ranges $n > 0$

$n > 0$

Data type DOUBLE

K

specifies the number of remaining coupons from settlement date to maturity.

Ranges $K > 0$

$K > 0$

Data type DOUBLE

k_o

specifies the time from settlement date to the next coupon as a fraction of the annual basis.

Ranges $0 < k_0 \le \frac{1}{n}$

$0 < k_0 \le 1/n$

Data type DOUBLE

p

specifies the price with accrued interest.

Ranges $p > 0$

$p > 0$

Data type DOUBLE

Details

The YIELDP function is based on the following relationship:

$$P = \sum_{k=1}^{K} c(k) \frac{1}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$

- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ldots, K-1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

The YIELDP function solves for *y*.

Example

In the following 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.The value returned is 0.0775031248.

```
proc cas;
  y=yieldp(1000, 0.01, 4, 14, .165, 800);
  print "y=" y;
run;
```

SAS writes the following output to the log:

```
y=0.0775031253
```

YRDIF Function

Returns the difference in years between two dates according to specified day count conventions; returns a person's age.

Returned data type: DOUBLE

Syntax

YRDIF(*start-date*, *end-date*<, *basis*>)

Arguments

start-date
specifies a SAS date value that identifies the starting date.

Data type DOUBLE

end-date
specifies a SAS date value that identifies the ending date.

Data type DOUBLE

basis

identifies a character constant or variable that describes how SAS calculates a date difference or a person's age. The following character strings are valid: '30/360' (a 30-day month and a 360-day year), 'ACT/ACT', 'ACT/360', 'ACT/365', and 'AGE'. For example, '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. For additional information, see SAS DS2 Language Reference.

identifies a character constant or variable that describes how SAS calculates a date difference or a person's age. The following character strings are valid:

'30/360'

specifies a 30-day month and a 360-day year in calculating the number of years. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year.

Alias '360'

Tip If either date falls at the end of a month, it is treated as if it were the last day of a 30-day month.

'ACT/ACT'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366.

Alias 'Actual'

'ACT/360'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

'ACT/365'

uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 365, regardless of the actual number of days in each year.

'AGE'

specifies that a person's age will be computed.

If you do not specify a third argument, AGE becomes the default value for *basis*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

Using YRDIF in Financial Applications

The Basics

The YRDIF function can be used in calculating interest for fixed income securities when the third argument, *basis*, is present. YRDIF returns the difference between two dates according to specified day count conventions.

Calculations That Use ACT/ACT Basis

In YRDIF calculations that use the *ACT/ACT* basis, both a 365-day year and 366-day year are taken into account. For example, if n365 equals the number of days between the start and end dates in a 365-day year, and n366 equals the number of days between the start and end dates in a 366-day year, the YRDIF calculation is computed as YRDIF=n365/365.0 + n366/366.0. This calculation corresponds to the commonly understood *ACT/ACT* day count basis that is documented in the financial literature. The values for *basis* also includes *30/360*, *ACT/360*, and *ACT/365*. Each has well-defined meanings that must be conformed to in calculating interest payments for specific financial instruments.

Computing a Person's Age

The YRDIF function can compute a person's age. The first two arguments, *start-date* and *end-date*, are required. If the value of *basis* is AGE, then YRDIF computes the age. The age computation takes into account leap years. No other values for *basis* are valid when computing a person's age.

Examples

Example 1: Calculating a Difference in Years Based on Basis

In the following example, YRDIF returns the difference in years between two dates based on each of the options for *basis*.

```
proc cas;
  sdate=(19490); /*12MAY2013*/
  edate=(19990); /*24SEP2014*/
  y30360=yrdif(sdate, edate, '30/360');
  yactact=yrdif(sdate, edate, 'ACT/ACT');
  yact360=yrdif(sdate, edate, 'ACT/360');
  yact365=yrdif(sdate, edate, 'ACT/365');
  print "y30360=" y30360;
  print "yactact=" yactact;
  print "yact360=" yact360;
  print "yact365=" yact365;
run;
```

SAS writes the following results to the log:

```
y30360=1.3666666667
yactact=1.3698630137
yact360=1.3888888889
yact365=1.3698630137
```

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:

```
proc cas;
   sdate=(7572); /*24SEP1980*/
   edate=(21451); /*24SEP2018*/
   age=yrdif(sdate, edate, 'AGE');
   print "Age=" age 'years';
run;
```

SAS writes the following results to the log:

```
Age=38 years
```

References

"Day count convention." *Wikipedia*, 2010. Available Day count convention. Accessed on May 1, 2013..

ISDA International Swaps and Derivatives Association, Inc "EMU and Market Conventions: Recent Developments." 1998. ISDA. Available EMU and Market Conventions: Recent Developments.

Mayle, Jan. 1994. *Standard Securities Calculation Methods – Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, New York: Securities Industry Association.

YYQ Function

Returns a SAS date value from year and quarter year values.

Returned data type: DOUBLE

Syntax

YYQ(*year*, *quarter*)

Arguments

year

specifies any valid expression that evaluates to a two-digit or four-digit whole number that represents the year.

Interaction The YEARCUTOFF= system option defines the year value for two-digit dates.

Data type DOUBLE

quarter

specifies the quarter of the year (1, 2, 3, or 4).

Data type DOUBLE

Details

The YYQ function returns a SAS date value that corresponds to the first day of the specified quarter. If either *year* or *quarter* is null or missing, or if the quarter value is not valid, the result is a null or missing value.

Example

The following statements illustrate the YYQ function:

```
proc cas;
  DateValue=yyq(2016,3);
  Date7Value=put(DateValue, date7.);
  Date9Value=put(DateValue, date9.);
  print "DateValue=" DateValue;
  print "Date7Value=" Date7Value;
  print "Date9Value=" Date9Value;
run;
```

The following output is printed to the SAS Log:

```
   DateValue=20636
   Date7Value=01JUL16
   Date9Value=01JUL2016
```

```
proc cas;
   StartOfQuarter=yyq(2016,4);
   StartofQuarter9=put(StartOfQuarter, date9.);
   print "StartofQuarter=" StartofQuarter;
   print "StartofQuarter9=" StartofQuarter9;
run;
```

The following output is printed to the SAS Log:

```
   StartofQuarter=20728
   StartofQuarter9=01OCT2016
```