SAS® Cloud Analytic Services 3.4: CASL Reference

SAS® Documentation
November 4, 2018

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2018. *SAS® Cloud Analytic Services 3.4: CASL Reference*. Cary, NC: SAS Institute Inc.

SAS® Cloud Analytic Services 3.4: CASL Reference

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

November 2018

3.4-P3:proccas

Contents

What's New in SAS Viya 3.4 CASL Reference

Enhancements to CASL Reference

New CASL Statements

The CASL statements shown in the following table are new.

New CASL Statements

Statement	Description
DEFAULT	Enables the user to configure how CASL will process certain statements.
DELETE	Deletes the specified value.
EXIT	Exits the current code stream and returns the value to the application.
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.
GLOBAL	Declares a variable that has global scope.
Labels	Identifies a statement that is referred to the GOTO statement.
Null	Creates an empty statement.
PRINTLST	Prints to the list 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.

Statement	Description
RUN	Executes the previously entered CASL statements.

New CASL Functions

The CASL functions shown in the following table are new.

New CASL Functions

Function	Description
ACTIONSTATUS	Returns a Boolean value if the session is active with an action.
ADDUNIQUE	Adds a value to a list if the value does not already exist within the list.
CANCELACTION	Cancels the action running on the given session.
CANCELACTIONS	Cancels multiple actions on the list of sessions (used for parallel sessions).
CLEAR	Clears the given value.
CODETOSTATUS	Converts the action status to a dictionary.
DICTIONARY	Searches for a value in the dictionary.
EXISTS	Searches a dictionary to see whether a key exists.
EXIT	Exits the current code stream with the given status.
FMTVAR	Creates a variable to store formats.
GETENV	Returns the value for an Operating System environment variable.
INPUT_FORMAT	Converts a string to a value.
is	Operates on the CASL variable to determine whether the variable matches the type of the specified function.
MEMORYSTATS	Displays CPU and memory statistics.
LIST PARALLEL SESSION	Lists a parallel session.

Function	Description
PUT	Formats a value with the given format and returns it as a string.
READFILE	Reads and returns the contents of the input file specified in the filename.
SYMDEL	Deletes a macro variable
SYMGET Function	Returns the value of a SAS macro variable.
SYMPUT	Stores a value in a SAS macro variable.
SYMPUTX	Assigns a value to a macro variable, and removes both leading and trailing blanks.
TIMESTAMP	Returns the current date and time.
UNIQUE	Searches for a value in a list.
UUID	Returns the universally unique identifier (UUID) for the given session.

Documentation Enhancements

- The common functions section is new. This section provides details about functions that are generic to SAS, and are not specific to CASL. For more information, see Common Functions on page 103.
- The functions overview has moved to the CASL built-in functions section in the documentation. For more information, see Overview of CASL Functions on page 64.
- The functions exception handling information has moved to the CASL functions section in the documentation. For more information, see Function Exception Handling on page 65.

Chapter 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.
- 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 (;).
- 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 *CASL Programmer's Guide*.

Chapter 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 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 results table or group of results tables can be sent as a response to an action, or a results table can be created from scratch from within CASL. A results 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.

Syntax: CAS Procedure

PROC CAS;

...CASL statements...

RUN;

QUIT;

Statement	Task
PROC CAS	Enables you to interact with the CAS server from SAS using the CAS language and CAS actions.

PROC CAS Statement

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

Syntax

PROC CAS;
RUN;
QUIT;

Without Arguments

PROC CAS does not have any required or optional arguments. Once PROC CAS is executed, it will continue 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 will have to execute PROC CAS again using the syntax above.

Details

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

The CAS result table is a table that is created as the result of an action. CASL also enables you to create your own result tables. Any operation on a result table might create a new result table. In addition to rows and columns, the table also contains labels and data types. The table is the primary means to return information to CASL.

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

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

```
%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;
```

Log 2.1 *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 2.1 *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: Define a New 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-THEN/ELSE 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.

Log 2.2 *Output Log: 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: Use Run-time Math Functions

Program

In this example, we not only create a new function but also use 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.

```
proc cas;                                          /* #1 */
  function SharedFeature (feature,number)          /* #2 */
  p = exp(lgamma (feature+1)- lgamma(feature-number+1)-   number*log(feature));
  return (1-p);                                    /* #4 */
 end;                                              /* #5 */
run;
   do n over {3 10 22 23 50 75};                   /* #6 */
   p = SharedFeature(365,n)                        /* #7 */
 print "Chance at least 2 out of " put(n,best3.)
  " share same birthday = " put(p,best8.4);        /* #8 */
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 2 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 END statement ends the function processing.

5 The DO statement iterates over a list.

6 The variable *p* is defined here. Use the function **SharedFeature** to calculate the probability that at least two people have the same birthday, given the number of people in the room.

7 The PRINT statement prints the results. Put acts as a DATA step.

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

Log 2.3 *Output: Log: Probability of Two People Sharing the Same Birthday in the Same Room*

```
Chance at least 2 out of    3 share same birthday = 0.0082
Chance at least 2 out of  10 share same birthday = 0.1169
Chance at least 2 out of  22 share same birthday = 0.4757
Chance at least 2 out of  23 share same birthday = 0.5073
Chance at least 2 out of  50 share same birthday = 0.9704
Chance at least 2 out of  75 share same birthday = 0.9997
```

Example 5: Train 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 v9 '/u/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 2.2 *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 2.3 *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 2.4 *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_
1	sunny	75	85	true	2	Don't Play	
2	overcast	83	78	false	1	Play	0.77777
3	sunny	68	80	false	2	Don't Play	

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 2.5 *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 2.6 *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

Chapter 3

CASL Statements

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

CASL Statements Syntax

TIP Use the CAS procedure before submitting your first CASL statement, otherwise it will generate an error in the SAS log.

PROC CAS;

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

<ASSIGNMENT> *target* = *expression*;

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

CONTINUE;

DEFAULT <CLEAR> <PRINT> <RESULT= *variable-name*> <STATUS= *variable-name*> <TOGGLE>;

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;
ENDSOURCE;
EXIT expression;
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
LEAVE;
LOADACTIONSET actionsetname;
NULL;
ON conditional-keyword response-keyword;
OUTPUT <ODS | LOG>;
PRINT value-1...value-N;
PRINTLIST expression;
RAISE condition;
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-name;
SOURCE variable; text; endsource;
UNSET <DISP> <LOGS>;
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;

Table 3.1 *Table of CASL Statements Tasks*

Statement	Task
ACTION	Runs SAS Cloud Analytic Services actions.
Assignment	Assigns a value to a variable.
CALL	Calls function with the specified arguments.
CONTINUE	Enables the next iteration of the loop to process without skipping any code in between.
DEFAULT	Enables you to configure how CASL will process certain statements.
DELETE	Deletes the specified value from memory.
DESCRIBE	Displays the data type of a variable.
DO	Starts a block of code that executes as one statement.
DO Statement, Iterative	Executes statements in a DO loop repetitively until a condition is false.
DO OVER	Iterates over an array, dictionary, or table.
DO UNTIL	Executes statements in a DO loop repetitively until a condition is false.
DO WHILE	Executes statements in a DO loop repetitively until a condition is false.
END	Ends a DO group or SELECT group processing.
ENDSOURCE	Marks the end of the SOURCE statement.
EXIT	Exits the current code stream and returns the value of the application.
FILE	Specifies different locations of output.
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.

Statement	Task
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-THEN/ELSE	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 a SAS Cloud Analytic Services action set.
Null	Creates an empty statement.
ON	Enables you to specify error handling during execution.
OUTPUT	Sets the active output location.
PRINT	Writes the values of constants, variables, and expressions to the current output location.
PRINTLST	Prints to the list file defined by the application.
RAISE	Sends a signal to the calling process to be raised on a condition.
RETURN	Returns a value from the current function.
RUN	Executes the previously entered CASL 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.
SOURCE	Enables you to embed text in the program and assign it to a given variable.
UNSET	Controls which messages are written to the SAS log by actions.

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

Dictionary

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: System Programming Guide*, and *SAS Visual Analytics: Programming Guide*.

Example: "Example 2: Run an Action" on page 7

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 and Creating CASL Variables" in *SAS Viya: 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 and Reason Codes" in *SAS Viya: 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*;

Actions

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 *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

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

```
Name = 'Jane Smith';
```

This assignment statement defines the target variable with the value of “88”.

```
xx.y = 88;
```

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

```
z = min(y, 70);
```

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;
run;
```

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

```
describe x
```

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.

```
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.

See Also

FUNCTION Statement on page 40

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. 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. The DO loop iterates and prints the incremented value of **test**. 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.

```
proc cas;
  function evaltest( x, y, t);
   if (x = y) then put "GOOD " t;
    else             put "ERROR " t;

   return( t+1);
  end;

test = 1;

  do i = 1 to 10 by 1;
```

```
    continue;
    put i;

   end;

test = evaltest( i, 11,test);

    put i;

run;
```

The following lines are written to the SAS log.

```
GOOD 1
11
```

See Also

- DO Statement on page 31
- DO Statement, Iterative Statement on page 32
- DO OVER Statement on page 33
- DO UNTIL Statement on page 35
- DO WHILE Statement on page 36
- LEAVE Statement on page 45

DEFAULT Statement

Enables you to configure how CASL will process certain statements.

Syntax

DEFAULT <CLEAR> <PRINT> <RESULT= *variable-name*> <STATUS= *variable-name*> <TOGGLE>;

Optional Arguments

CLEAR
clears the default variable for RESULT= or STATUS=.

PRINT
prints the STATUS= of the disp and logs state.

RESULT=
sets the default variable where results from an action can be saved to.

Note: If you use the RESULT= parameter to set the default result variable, then the ACTION statement does not need to specify a result variable.

TIP The results are overwritten each time an action runs and generates a result table.

STATUS=
sets the default variable where status and severity codes from an action can be saved to.

TOGGLE
displays the disposition or logs for an action.

Default OFF

variable-name
assigns a name to the variable that you want to set as a default.

Examples

Example 1: Clear the Default for Result or Status

You can set a default for variables saved in the ACTION statement. You can clear the default for result or status.

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

Example 2: Toggle the Display of Logs for an Action

You can toggle the displaying of the disposition and or logs for an action. The default is OFF.

```
default toggle disp logs;
```

Example 3: Print the Results of the Default Variable

You can print the results of the default variable.

```
default result=res;
```

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

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*.

```
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**.

```
List ( 3 entries, 3 used);
[About] List( 6 entries, 6 used);
[CAS] String;
[Version] String;
[Built] String;
[Copyright] String;
[System] List ( 6 entries, 6 used);
[Hostname] String;
[OS Name] String;
[OS Family] String;
[OS Version] String;
[OS Release] String;
[Model Number] String;
[Documentation] String;
[server] Table( [1] Rows [2] columns
Column Names:
[1] Node Count (Double)
[2] Total Actions (Double)
[nodestatus] Table( [143] Rows [5] columns
Column Names:
[1] Node Name (String)
[2] Role (String)
[3] Uptime (Sec) (Double)
[4] Running (Double)
[5] Stalled (Double)
```

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.

```
print r.server[1];
run;
```

The output gives you information about the CAS server.

```
{nodes=143 , actions=17}
```

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.

Log 3.1 *SAS Log*

```
double;
```

See Also

- CASL Data Types
- CASL Expressions
- CASL Result Tables
- CASL Variables
- Using the DESCRIBE Statement

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: Define a New Function" on page 8

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-THEN/ELSE Statement" on page 43. 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.

```
if year > 7 then
  do;
   months = years*12;
   put years= months=;
  end;
else;
  do;
   yrsleft=7-years;
  end;
```

See Also

- CONTINUE Statement on page 26

- END Statement on page 36
- LEAVE Statement on page 45

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

There are four other forms of the DO statement:

- 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 DO statement, see DO Statement on page 31. For more information about the IF-THEN/ELSE statement, see IF-THEN/ELSE Statement on page 43.
- The DO OVER statement iterates over an array, dictionary, or table. For more information, see DO OVER Statement on page 33.
- 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. For more information, see DO UNTIL Statement on page 35.
- 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. For more information, see DO WHILE Statement on page 36.

Example: Using the Do Loop Statement

This statement identifies the loop variable as **i**.

```
do i=1 to 10;
```

This statement executes the loop variable in increments of 3.

```
do i=1 to 10 by 3;
do i=1 to 10 while( i < 5);
do i=1 to 10 by 1 until ( i>=5);
```

See Also

- CONTINUE Statement on page 26
- END Statement on page 36
- LEAVE Statement on page 45

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: Use Run-time Math Functions" on page 9

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.

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: System Programming Guide*.

```
action simple.summary result=CPSSum                        /*1*/
  table={caslib="casdata", name="phoneSubs"};
run;
  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 on page 78.

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 on page 24.

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 on page 33.

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 on page 43.

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 on page 50.

Log 3.2 *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 on page 26
- END Statement on page 36
- LEAVE Statement on page 45

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.

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 ten iterations in all (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

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

See Also

- CONTINUE Statement on page 26
- END Statement on page 36
- LEAVE Statement on page 45

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 32

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.

Example: Using a DO WHILE Statement

These statements repeat the loop while **x** is greater than 5. 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).

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

See Also

- CONTINUE Statement on page 26
- END Statement on page 36
- LEAVE Statement on page 45

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 on page 31
- DO Statement, Iterative on page 32
- DO OVER Statement on page 33
- DO UNTIL Statement on page 35
- DO WHILE Statement on page 36
- FUNCTION Statement on page 40
- SELECT Statement on page 56

ENDSOURCE Statement

Marks the end of the SOURCE statement.

Syntax

ENDSOURCE;

Without Arguments

There are no arguments or parameters for the SOURCE statement.

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.

FILE Statement

Enables you to specify a different output destination.

Default: Defined by the application.

Example: "Example 5: Train a Decision Tree Model and Score Data" on page 10

Syntax

FILE "*fileref*" ;

Required Argument

fileref
Fileref can be any of the following:

- **`ods`** uses the default print location with ODS output when appropriate.
- **`log`** uses the default print location, but does not use ODS. It uses the internal format routines and outputs to the SAS Log when appropriate.
- *<name>* uses the named fileref location for output.
- *<path>* uses the given path as the location for output.

You can use the FILE statement to change the active output location to the specified file. Each *fileref* has a name. The name can be used to change the active file to another *fileref* at any time.

Example: Use the FILE statement to change output locations

```
file tablefile "/u/user/table.lst"
print table;
file ods;                          /* #1 */
file nodes "/u/user/nodes";        /* #2 */
action listnodes result=r;
print r;                           /* #3 */
file log;                          /* #4 */
print r;                           /* #5 */
file ods;
```

1 Sets the active output location to ODS.

2 Sets the active output location to **`/u/user/nodes`**.

3 The result of the listnodes action is printed to **`/u/user/nodes`**.

4 Sets the active output location to the SAS log.

5 The same results are printed to the SAS log.

See Also

OUTPUT Statement on page 49

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;
```

Log 3.3 *SAS Log*

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

See Also

- CASL Functions on page 63
- Common Functions on page 103
- FUNCTION Statement on page 40
- FUNCTIONLIST Statement on page 41

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 date12. 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 statement must follow a FUNCTION statement.

Example: "Example 3: Define a New Function" on page 8

Syntax

FUNCTION *function- name* <*parameter–1*, *parameter–2*>;
statements;
END;

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.

See Also

- CALL Statement on page 25
- END Statement on page 36
- FUNCTIONLIST Statement on page 41

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

GLOBAL Statement

Declares a variable that has global scope.

Syntax

GLOBAL *variable*;

Required Argument

variable
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.

Comparisons

Within CASL, all statements are considered *SAS global statements*. A *SAS global statement* is a string of SAS keywords, SAS names, special characters, and operators that instructs SAS to perform an operation or that gives information to SAS. Each SAS global statement ends with a semicolon and can be used anywhere in a SAS program. In CASL, any CASL statement can be used anywhere a statement is allowed within the program.

The GLOBAL statement declares a variable to be global scope. A global scope variable can be accessed and used anywhere within the CASL program.

Example: Using the GLOBAL Statement

```
proc cas;
  km=13587;
  function distance(km);
  global rental;
  rental.a  = km;
  rental.b  = km*2;
  return(0);
end;
  end_distance = distance(7);
put end_distance rental.a rental.b;
  run;
```

See Also

Variable Scope

GOTO Statement

Transfers execution immediately to a labeled statement.

Syntax

GOTO *label*;

Required Argument

label
specifies a statement label that identifies the GOTO destination.

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 Labels statement.

Comparisons

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

Example: Using the GOTO Statement

In this example, when x = 2, program execution transfers to the DONE label.

```
method run();
   x = 1;
   do;
      if x=2 then goto done;
      put x;
      x+1;
   end;
   done:
   put x;
end;
```

See Also

- DO Statement on page 31
- DO Statement, Iterative on page 32
- DO OVER Statement on page 33
- DO UNTIL Statement on page 35
- DO WHILE Statement on page 36
- IF-THEN/ELSE Statement on page 43
- LEAVE Statement on page 45

IF-THEN/ELSE Statement

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

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

Example: "Example 3: Define a New Function" on page 8

Syntax

IF *condition* **THEN** *statement*;

<ELSE *statement*;>

Required Arguments

condition
Used to determine which statement executes next.

statement
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 then THEN keyword is executed. If the expression results in a false value and there is an ELSE keyword, the statement immediately after the ELSE is executed. If there is no ELSE keyword, 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 on page 56.

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;
```

Log 3.4 *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 on page 31
- DO Statement, Iterative on page 32

- DO OVER Statement on page 33
- DO UNTIL Statement on page 35
- DO WHILE Statement on page 36
- SELECT Statement on page 56

Label Statement

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

Supports: GOTO Statement

Syntax

labelstatement:

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.

See Also

- GOTO Statement on page 42
- LEAVE Statement on page 45

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 a DO loop prematurely. You can use the LEAVE statement either on its own or use it based on a condition (for example, in conjunction with an IF statement).

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:

Log 3.5 *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: Train a Decision Tree Model and Score Data" on page 10

Syntax

LOADACTIONSET "*action-set-name*";

Required Argument

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

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.

ON Statement

Enables you to specify error handling during execution.

Syntax

ON *condition-keyword*
<response-keyword>;

Required Arguments

condition-keyword
specifies a type of an error

The types of supported condition keywords are as follows:

abort
An abort has been raised.

access
An access violation has occurred.

fpe
A floating point exception has occurred.

function
The function name has not been defined.

illop
This condition is raised after performing an invalid operation.

interrupt
A user has raised an interrupt to stop the processing of the current operation.

mixed
A mixed array or dictionary has been detected.

reduce
An operation has failed during the expression evaluation.

unknown
This condition is raised after accessing a variable with an unknown value.

user-*1*, user-*2*, user-*n*
A condition can be raised by a user for their own purpose.

Optional Argument

response-keyword
Determines the type of response necessary for a condition.

The types of supported response keywords are as follows:

cancel
cancels the execution.

ignore
ignores the conditional error, and continues if possible. Otherwise the error is responds by canceling the execution.

propagate
propagates the condition to previous frame.

replace
replaces the operation with the value of the specified expression.

return
if an error occurs within a function, then the function returns the expression value.

Examples

Example 1: Using unknown as the Conditional Response

The following example demonstrates using the unknown the conditional response. The unknown conditional response keyword is raised only when a variable contains a missing value. When evaluating the expression $z = x+y$ the unknown conditional response is raised because x does not have a value.

```
proc cas;
  x=.;
   on unknown replace;
  y=5;
  z=x+y;
  if z>6 then print "The value is greater than 6";
   else print "Value: " z;
run;
```

Log 3.6 *SAS Log*

```
Value:.
```

Example 2: Using illop as the Conditional Response

The following example demonstrates using the unknown the conditional response. The unknown conditional response keyword is raised only when an invalid operation is performed. When evaluating the expression $z = x+y$ the illop conditional response is raised because you cannot add a numeric and a character value together. SAS converts the character value $x=$ *"The"* to numeric and sets it to missing, and attempts to evaluate the expression $z = x+y$.

```
proc cas;
  on illop;
```

```
  x="The";
  y=5;
  z=x+y;
  if z>6 then print "The value is greater than 6";
   else print "Value: " z;
run;
```

Log 3.7 *SAS Log*

```
WARNING: Convserion overflow, value set to missing.
NOTE: Character value "The" has been converted to numeric

Value:.
```

See Also

RAISE Statement on page 52

OUTPUT Statement

Sets the active output location.

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.

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

Log 3.8 *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.

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

Output 3.1 *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 on page 50

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: Use Run-time Math Functions” on page 9

Syntax

PRINT *value-1*<*value-2*>...<*value-N*>;

Optional Argument

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;
```

Log 3.9 *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.

Log 3.10 *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 following example uses the PRINT statement to print the variables, *c*, *d*, and *e*. Variable *c* contains a space after the last value in the string. However, variables *d* and *e* do not contain any spaces. Add spaces as needed at the end of each variable to separate each variable.

```
proc cas;
  c="8ltg25! ";
  d="knj08kuc$";
  e="xnd8";
```

```
  print "random password = "c d e;
run;
```

The SAS log displays the output with the spacing provided in the code. The PRINT statement does not automatically add spaces between the variables *d* and *e*.

Log 3.11 *SAS Log*

```
random password generator =8ltg25! knj08kuc$xnd8
```

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.

RAISE Statement

Sends a signal to the calling process.

Tip: The following statement returns a zero on success and a nonzero on failure: **`raise ( );`**.

Syntax

RAISE *condition*;

Required Argument

condition
specifies one or more numeric or character expressions that result in a value on which some decision depends.

RETURN Statement

Returns a value from the current function.

Example: "Example 5: Train a Decision Tree Model and Score Data" on page 10

Syntax

RETURN <*expression* | *value*>;

Optional 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. The first function determines whether or not the class is on track or not. In the example, the class average is 81 and the class is on track because the class average is greater than 67. The RETURN statement determines the value of the student's grade. The current student has a grade of 67, all students automatically get an additional point added to their grade, therefore the returned value would be 68. The second and third function, determines the student's final grade and whether or not the student is on track to successfully complete the course. The second RETURN statement uses the value of 67 to determine whether or not the student is on track or not. Since the value 67 is not equal to or greater than 81 therefore the student is not on track. In order to find out if another class or student are on track, simply replace the value of 67 from the first IF statement, and the second RETURN statement for an alternate output.

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

Log 3.12 *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.

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

Example: "Example 5: Train a Decision Tree Model and Score Data" on page 10

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.

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.

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

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;
  se
ssion 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 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.

Log 3.13 *SAS Log*

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

See Also

ACTION Statement on page 22

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 on page 31
- END Statement on page 36
- IF-THEN/ELSE Statement on page 43

SESSION Statement

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

Example: "Example 5: Train a Decision Tree Model and Score Data" on page 10

Syntax

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

Required Argument

session-name
specifies the name of an existing session.

Examples

Example 1: Print Session ID

This example prints your session ID 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;
```

```
{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;
```

SOURCE Statement

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

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.

Example: 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;
```

Log 3.14 *SAS Log*

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

UNSET Statement

Controls which messages are written to the SAS log by actions.

Tip: You can specify the following statement to suppress both types of messages: **`unset disp logs;`**

Syntax

UNSET <DISP | LOGS>;

Optional Arguments

<DISP>
specifies to suppress printing the disposition in the SAS log.

<LOGS>
specifies to suppress printing messages from CAS in the SAS log.

Examples

Example 1: Using the UNSET Statement

The UNSET statement turns off options. The **`unset disp`** is used to suppress the disposition from the SAS log.

```
proc cas;
  session casauto;
  unset disp;
  setsessopt / caslib="myCasLib";
run;
```

Log 3.15 *SAS Log*

```
NOTE: Active Session is now casauto.
NOTE: 'MYCASLIB' is now the active caslib.
```

Example 2: Using the SET Statement

You can use the SET statement to turn on options. The **`set disp`** is used to display the disposition in the SAS log.

```
proc cas;
```

```
  session casauto;
  set disp;
  setsessopt / caslib="myCasLib";
run;
```

Log 3.16 *SAS Log*

```
NOTE: Active Session is now casauto.
NOTE: 'MYCASLIB' is now the active caslib.
NOTE: Disposition: Severity Normal
```

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.

Note: 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.

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-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

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

Specifies the file format and options.For more information, see FILETYPE=.

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.

```
libname mycas cas;
proc cas;
  session casauto;
  upload path="your-file-path\titanic3.csv" importoptions="CSV" casout={name="Titanic01"
run;
table.tableInfo;                                                    /* #2 */
run;
proc cas;
  simple.Summary result=Titanic02 / table={name="Titanic01"};      /* #3 */
  print Titanic02;
  Titanic03=findTable(Titanic02);                                   /* #4 */
  saveresult Titanic03 dataset=mycas.Titanic03;                     /* #5 */
run;
```

1. Use the UPLOAD statement to transfer Titanic3.csv from the SAS client to the server. The IMPORTOPTIONS= “CSV” option to specify the file type. 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 on page 78.
5. Use the SAVERESULT statement to save *Titanic03* as a SAS data set to your Mycas caslib. For more information, see SAVERESULT Statement on page 54.

Chapter 4

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 sever-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 on page 40.
- CASL also supports common SAS functions.

For a list of available functions that are specific to CASL, see CASL Built-In Functions on page 63. For a list of supported common SAS functions, see Common Functions on page 103.

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 will display MISSING every time there is an error. Otherwise, it will continue 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:

Array
: functions that operate on a named aggregate collection of homogenous data

Control
: functions that specify how the program maintains program flow

File IO
: functions that are used to input and output data or results from the client or server side

Formatting
: functions that modify the data type of a variable

Function Information
: functions that return strings, integers, or other function information

Macros
: functions that enable you to use macro variables within CASL

Server-Side
: functions that can be used on the server side or during asynchronous sessions.

Status
: functions that operate on the execution state or process flow of a program at any time

Variable Information
: functions that operate on variables and return names, types, lengths, informats, labels, and other variable information

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. 71)	Adds a value to an array, if the value does not already exist within the array.
	DICTIONARY Function (p. 75)	Searches for a value in the given dictionary.
	DIM Function (p. 76)	Retrieves the dimensions of an array or dictionary.
	EXISTS Function (p. 77)	Searches a dictionary to see if a key exists or not.
	SORT Function (p. 88)	Returns a list sorted in ascending order.
	SORT REV Function (p. 90)	Returns a list sorted in descending order.
	UNIQUE Function (p. 99)	Searches an array to determine if a value exists in the array.
Control	EXIT Function (p. 77)	Exits the block of CASL code with the given status.
	GETENV Function (p. 80)	Returns the environment variable for the given string
	SLEEP Function (p. 88)	Sleep in the operating system (OS) for the number of seconds specified.
	TIMESTAMP Function (p. 98)	Returns the current date and time.
	UUID Function (p. 99)	Returns the universally unique identifier (UUID) for the given session.
File IO	READFILE Function (p. 86)	Reads and returns the contents of the input file specified in the file name.
	READPATH Function (p. 86)	Reads the contents of the file given into the variable as a string.
Formatting	FMTVAR Function (p. 79)	Creates a format variable.
	INPUT_FORMAT Function (p. 80)	Converts a string to a value.
	PUT Function (p. 85)	Formats a value with the given format and returns it as a string.
Function Information	CLEAR Function (p. 72)	Clears all values if no arguments are supplied.
Macros	SYMDEL Function (p. 91)	Deletes the specified macro variable.
	SYMGET Function (p. 92)	Returns the value of a macro variable.
	SYMPUT Function (p. 92)	Stores a value in a SAS macro variable.
	SYMPUTX Function (p. 93)	Assigns a value to a macro variable, and removes both leading and trailing blanks.

Category	Language Elements	Description
Result Tables	ADD_TABLE_ATTR Function (p. 69)	Adds attributes to a table.
	ADDBYGROUP Function (p. 69)	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. 71)	Adds a row to a table.
	COMBINE_TABLES Function (p. 74)	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.
	FINDTABLE Function (p. 78)	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.
	LOC Function (p. 82)	Returns the row number in which the given value is found in the given column.
	NEWTABLE Function (p. 84)	Creates a new table.
	RESULT_BY_COL Function (p. 87)	Creates a new table with the given columns.
	RESULT_BY_TYPE Function (p. 87)	Creates a new table with columns that match the type specifications.
	TABCOLUMNS Function (p. 95)	Retrieves the columns from a result table.
	TABTYPES Function (p. 96)	Extracts the data types from a result table into a dictionary.
Server-side	CANCELACTION Function (p. 72)	Cancels the action running on the given session.
	CANCELACTIONS Function (p. 72)	Cancels the action running on the given list of sessions.
	CREATE_PARALLEL_SESSI ON Function (p. 74)	Starts a session with the same identity as the calling action. You can call this function to create to create multiple sessions.
	LIST_PARALLEL_SESSION Function (p. 82)	Lists a parallel session.
	SEND_RESPONSE Function (p. 88)	Sends the specified result back to the client.
	TERM PARALLEL SESSION Function (p. 97)	Terminates a parallel session.
	WAIT_FOR_NEXT_ACTION Function (p. 100)	Wait for a completed action.

Category	Language Elements	Description
Status	ACTIONSTATUS Function (p. 68)	Returns a Boolean value if the session is active with an action.
	CODETOSTATUS Function (p. 73)	Converts the action status into a dictionary.
	GETENV Function (p. 80)	Returns the environment variable for the given string
	MEMORYSTATS Function (p. 83)	Displays CPU and memory statistics.
	SLEEP Function (p. 88)	Sleep in the operating system (OS) for the number of seconds specified.
	TIMESTAMP Function (p. 98)	Returns the current date and time.
	UUID Function (p. 99)	Returns the universally unique identifier (UUID) for the given session.
Variable Information	CLEAR Function (p. 72)	Clears all values if no arguments are supplied.
	is Function (p. 81)	Operates on the CASL variable to determine whether the variable matches the type of the specified function.

Dictionary

ACTIONSTATUS Function

Returns a Boolean value if the session is active with an action.

Category: Status

Returned data type: TRUE | FALSE

Syntax

ACTIONSTATUS (*session-name*);

Required Argument

session-name
Specifies the session name to retrieve status from.

Example: Retrieving ACTIONSTATUS

```
proc cas;
  session casauto;
```

```
  table.loadtable result=r/
   path="iris.sashdat"
   casOut={name="iris", replace="true"};
  table.columnInfo / table="iris";
run;
  stat=actionstatus("casauto");
  print stat;
run;
```

The following log displays the Boolean value of the ACTIONSTATUS function.

Log 4.1 *SAS Log*

```
FALSE
```

ADD_TABLE_ATTR Function

Adds attributes to a table.

Category: Result Tables

Syntax

ADD_TABLE_ATTR (*table-name*, "*key*", "*value*");

Required Arguments

table-name
The name of the table which the attributes are added to.

key
Uniquely identifies a specific record and its order among other records in a database or table.

value
The identified record in a key.

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.

```
proc cas;
  session casauto;
    table.loadTable result=r/
      caslib="sasdemo"
      path="iris.sashdat"
      casOut={name="iris" replace=true};
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 4.1 *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 4.2 *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 a row to a table.

Category: Result Tables

Returned data type: If the function was not successful in adding a row to the table, then the returned value is —1.

Syntax

ADDROW (*table-name*, *row*);

Required Arguments

table-name
Specifies the name of the table where the rows are added to.

row
The row can be a number or a character value.

Example: Add Three Rows to the Results 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});
```

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 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.

CANCELACTION Function

Cancels the action running on the given session.

Category: Server-side

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-name*");

Required Argument

session-name
Specifies the session name on which the action is running.

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-names*});

Required Argument

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

CLEAR Function

Clears all values if no arguments are supplied.

Categories: Function Information

Variable Information

Returned data type: If the value TRUE is returned, then the values were cleared. If the value FALSE is returned, then the values were not cleared.

Syntax

CLEAR ();

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:

Log 4.2 *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

Returned data type: If the function is successful, the returned data type is the CAS result table. If the function is not successful, the returned data type of 0 is too many arguments. If the function is not successful, the returned data type of 3 is unable to combine tables.

Syntax

COMBINE_TABLES (*list-of-tables*, <*"key-pattern"*>);

Required Argument

list-of-tables
The list of tables to combine.

Optional Argument

key-pattern
If the key pattern is given, keep only the tables whose key matches the pattern within the table's key.

CREATE_PARALLEL_SESSION Function

Starts a session with the same identity as the calling action. You can call this function to create 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

```
session[i] = create_parallel_session();
```

Example 2: Create Parallel Sessions with n-Workers

```
session[i] = create_parallel_session(37);
```

DICTIONARY Function

Searches for a value in the given dictionary.

Category: Array

Returned data type: Returns the Boolean value of FALSE for when the string is not a string or there is a parameter error. Returns the Boolean value of TRUE when the string is in the table along with the value of the string.

Syntax

DICTIONARY (*value*, *string*);

Required Arguments

value
Specifies a character or numeric constant, variable, or expression in the array.

string
Specifies the string to search for in the array.

Example: Using the DICTIONARY Function

The following example demonstrates how to use the DICTIONARY function to look up a value in an array. The example searches for the value *Keefer* in the *studentId* array.

```
proc cas;
  studentID={Woods=13445, Shafer=36772, McKenna=37854, Keefer=68101, Johnson=34690,
  print 'StudentID= ' dictionary(studentID, 'Keefer');
```

```
run;
```

Log 4.3 *SAS Log*

```
StudentId=68101
```

See Also

- CASL Arrays
- CASL Dictionaries

DIM Function

Retrieves the dimensions of an array or dictionary.

Category: Array

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,
  print 'Dimensions are=' dim(studentId);
run;
```

Log 4.4 *SAS Log*

```
Dimensions are=8
```

See Also

- CASL Arrays
- CASL Dictionaries

EXISTS Function

Searches a dictionary to see if a key exists or not.

Category: Array

Returned data type: TRUE | FALSE

Syntax

EXISTS (*dictionary*, <*string*>);

Required Argument

dictionary
a type of memory variable that consists of an array that contains a list of numbers or text strings that can be identified by a key.

Optional Argument

string
one or more consecutive alphanumeric characters, other keyboard characters, or both.

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 *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 *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 14

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.

```
proc cas;
  table.fetch result=r / table="cars" to 428;
  findcar=sort_rev(findtable(r), "Invoice)[1:25, {"Make", "Model",
  print findcar;
run;
```

Output 4.3 *Findcar Results*

findcar: Results

Make	Model	Type	DriveTrain	MSRP	Invoice
Porsche	911 GT2 2dr	Sports	Rear	192465	173560
Mercedes-Benz	CL600 2dr	Sedan	Rear	128420	119600
Mercedes-Benz	SL600 convertible 2dr	Sports	Rear	126670	117854
Mercedes-Benz	SL55 AMG 2dr	Sports	Rear	121770	113388
Mercedes-Benz	CL500 2dr	Sedan	Rear	94820	88324
Mercedes-Benz	SL500 convertible 2dr	Sports	Rear	90520	84325
Mercedes-Benz	S500 4dr	Sedan	All	86970	80939
Acura	NSX coupe 2dr manual S	Sports	Rear	89765	79978
Jaguar	XKR convertible 2dr	Sports	Rear	86995	79226
Audi	RS 6 4dr	Sports	Front	84600	76417
Jaguar	XKR coupe 2dr	Sports	Rear	81995	74676
Dodge	Viper SRT-10 convertible 2dr	Sports	Rear	81795	74451
Porsche	911 Carrera 4S coupe 2dr (convert)	Sports	All	84165	72206
Mercedes-Benz	G500	SUV	All	76870	71540
Cadillac	XLR convertible 2dr	Sports	Rear	76200	70546
Porsche	911 Carrera convertible 2dr (coupe)	Sports	Rear	79165	69229
Mercedes-Benz	S430 4dr	Sedan	Rear	74320	69168
Volkswagen	Phaeton W12 4dr	Sedan	Front	75000	69130
Jaguar	XJR 4dr	Sedan	Rear	74995	68306
Jaguar	XK8 convertible 2dr	Sports	Rear	74995	68306
Porsche	911 Targa coupe 2dr	Sports	Rear	76765	67128
BMW	745Li 4dr	Sedan	Rear	73195	66830
Land Rover	Range Rover HSE	SUV	All	72250	65807
Audi	A8 L Quattro 4dr	Sedan	All	69190	64740
Jaguar	XK8 coupe 2dr	Sports	Rear	69995	63756

FMTVAR Function

Creates a format variable.

Category: Formatting

Syntax

FMTVAR(*format-name*, <*format-width*>, <*d*>);

Required Argument

format-name
specifies the name of the format. The format is a SAS format or a user-defined format that was previously defined.

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

If you use an incompatible format, such as a numeric format to write character values, SAS first attempts to use an analogous format of the other type. If this attempt fails, an error message that describes the problem is displayed in the log.

GETENV Function

Returns the environment variable for the given string

Categories: Control
Status

Returned data type: If the function was successful, then the returned data type is a string with 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.

INPUT_FORMAT Function

Converts a string to a value.

Category: Formatting

Returned data type: For a list of possible returned data types, see the Details section below.

Syntax

INPUT_FORMAT (<*string*>);

Optional Argument

string
one or more consecutive alphanumeric characters, other keyboard characters, or both.

Details

Returned Data Type

The following is a list of possible returned data types for INPUT_FORMAT function:

0
wrong args

–1
invalid results

—2
overflow

—3
blank

—4
bad conversion

—5
value is not a string

is Function

Operates on the CASL variable to determine whether the variable matches the type of the specified function.

Category: Variable Information

Returned data type: If the function is successful, then the Boolean value of TRUE is returned. Otherwise, FALSE is returned.

Syntax

is (*Type* , *variable*);

Required Arguments

Type
Specifies the function type.

The following list specifies the supported function types

- isArray
- isBlob

- isDouble
- isDictionary
- isInteger
- isList
- isString
- isTable
- isVarbinary

variable-name
Specifies the name of the variable that does or does not contain the data type.

Example: isDouble Function: 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;
```

Log 4.5 *SAS Log*

```
7 is a double.
```

LIST_PARALLEL_SESSION Function

Lists a parallel session.

Category: Server-side

Syntax

LIST_PARALLEL_SESSION (<*session-name*>);

Optional Argument

session-name
Specifies the session name to list.

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 cas;
  table.fetch result=r/ table="cars" to=428
   findcar=sort_rev(findtable(r), "Invoice")[1:25,     {"Make", "Model", "Type", "DriveT
  y=loc(findcar, "MSRP", 75000);
   print "The row in which the value appears is= " y;
run;
```

Log 4.6 *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;
```

Log 4.7 *SAS Log*

```
{user_cpu_time=9.68,system_cpu_time=2.64,timestamp=1840474683,current_memory=5201
5104,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 types*, *row-1*, <*row-2*, *row-3, ...*>);

Required Arguments

table-name
The new table name.

column-name
Creates a list of variable (column) names.

data types
Creates a list of types for each variable. For information on supported data types, see Data Types Supported in CASL.

row
Creates observations for the new table.

Example

This example creates a new table titled Average Class Grades with three observations and three variables.

```
proc cas;
  title "Average Class Grades";
  columns={'Class A Average', '8th Grade Average', 'Class B Average'};   /* #1 */
  coltypes={'int64' 'int64','int64'};                                    /* #2 */
  row1={71 74 84 65 70 };                                                /* #3 */
  row2={65 74 65 70 75 };
  row3={80 80 90 95 100};
mytable=newtable('mytable',columns, coltypes, row1, row2, row3);         /* #4 */
print  mytable;
describe mytable;
run;

quit;
```

1 The first argument creates a list of variable (column) names.

2 The second argument creates a list of data types for each variable.

3 The third argument creates observations for the new table.

4 Use the NEWTABLE function to create the new table which includes list of variables, list of types for each variables, and observations.

Output 4.4 *NEWTABLE Function Output: Average Class Grades Output*

Average Class Grades

mytable: Results

Class A Average	8th Grade Average	Class B Average
71	74	84
65	74	65
80	80	90

The DESCRIBE statement lists your new table with each column and its data types as you set them.

Log 4.8 *SAS Log: DESCRIBE Statement*

```
[mytable] Table ( [3] Rows [3] columns
 Column Names:
 [1] Class A Average  [Class A Average ] (int64)
 [2] 8th Grade Average [8th Grade Average] (int64)
 [3] Class B Average  [Class B Average ] (int64)
```

See Also

- "CASL Data Types" in *CASL Programmer's Guide*
- DESCRIBE Statement on page 29

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
a type of SAS language element that is used to write or display data values according to the data type: numeric, character, date, time, or timestamp.

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;
```

Log 4.9 *SAS Log*

```
0000000A
```

See Also

PRINT Statement

READFILE Function

Reads and returns the contents of the input file specified in the file name.

Category: File IO

Syntax

READFILE *filename*;

Required Argument

filename
The name of the file whose contents you are attempting to input.

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 the contents of the file given into the variable as a string.

Valid in: Client-side only

Category: File IO

Returned data type: String

Syntax

READPATH (*file-name*);

Required Argument

file-name
The name of the file that you want to read.

Example: Read the Contents of a File Into a Variable

```
store = readpath (" /u/sasdemo/ds/samplecode.sas");
```

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.

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 on supported data types, see Data Types Supported in CASL.

See Also

"What Are Data Types?" in *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

```
send_response(r, {test="test"});
```

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.

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 on page 50.

The sorted list is printed in the SAS log

Log 4.10 *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;
  session casauto;
  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 on page 50.

The sorted keyed list is printed in the SAS log.

Log 4.11 *SAS Log*

```
{Economics_Avg=82.5,Math_Avg=88.6,English_Avg=89.5,Science_Avg=90.3}
```

See Also

SORT REV Function on page 90

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 on page 50.

The sorted list is printed in the SAS log

Log 4.12 *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;
  session casauto;
  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 on page 50.

The sorted keyed list is printed in the SAS log.

Log 4.13 *SAS Log*

```
{Science_Avg=90.3,English_Avg=89.5,Math_Avg=88.6,Economics_Avg=82.5}
```

See Also

SORT Function on page 88

SYMDEL Function

Deletes the specified macro variable.

Category: Macros

Returned data type: For a list of possible returned data types, see Details section below.

Syntax

SYMDEL "*macro-variable*";

Required Argument

macro-variable
The name of the macro variable that you want to delete.

Details

Returned Data Types

The following are the possible returned data types from the SYMDEL function:

!0
failure

0
success

1
arg number string

2
wrong args

7
null string provided

8
null string after removing leading and trailing blanks

See Also

- SYMGET Function
- SYMPUT Function
- SYMPUTX Function

SYMGET Function

Returns the value of a macro variable.

Category: Macros

Returned data type: If the function is not successful and a value is not found, then a missing value is returned. Otherwise, the value of the macro variable is returned.

Syntax

SYMGET "*macro-variable*";

Required Argument

macro-variable
The name of the macro-variable from which the value is returned from.

See Also

- SYMDEL Function
- SYMPUT Function
- SYMPUTX Function

SYMPUT Function

Stores a value in a SAS macro variable.

Category: Macros

Returned data type: Server-side: For a list of possible returned data types, see the Details section below.
Client-side: 1– success, any other returned data type is failure.

Note: Using the SYMPUT function does not modify the values of the macro variable in the SAS client.

Syntax

SYMPUT (*name*, *value*);

Required Arguments

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

Returned Data Types

The following are the possible returned data types for the SYMPUT function:

!=
failure

0
success

1
specified name is not a string

3
wrong arg count

7
specified name is a blank string

8
nothing left of string after leading and trailing blanks were removed

9
cannot format the value as a string

13
out of memory

See Also

- SYMGET Function
- SYMDEL Function
- SYMPUTX Function

SYMPUTX Function

Assigns a value to a macro variable, and removes both leading and trailing blanks.

Category: Macros

Returned data type: For a list of possible returned data types, see Details section below.

Note: Using the SYMPUTX function does not modify the values of the macro variable in the SAS client.

Syntax

SYMPUTX (*macro-variable-name*, *expression*, *value*, <*flag*>);

Required Arguments

macro-variable-name
Can be one of the following forms:

- A character string that is a SAS name, enclosed in quotation marks.
- The name of a character variable whose values are SAS names. A character expression that produces a macro variable name. This form is useful for creating a series of macro variables. A character constant, variable, or expression.

Leading and trailing blanks are removed from the value of name, and the result is then used as the name of the macro variable.

expression
Specifies a macro variable string.

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. Leading and trailing blanks are removed, and the resulting character string is assigned to the macro variable.

Optional Argument

<flag>
Indicates the scope of the variable.

Details

Returned Data Types

The following are the possible return data types for the SYMPUTX function:

!=
failure

0
success

1
specified name is not a string

3
wrong arg count

7
specified name is a blank string

8
nothing left of string after leading and trailing blanks were removed

9
cannot format the value as a string

13
out of memory

See Also

- SYMGET Function
- SYMDEL Function

- SYMPUT Function

TABCOLUMNS Function

Retrieves the columns from a result table.

Category: Result Tables

Note: The TABCOLUMNS function only operates 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 using the TABCOLUMN function to retrieve the columns 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 TABCOLUMNS function retrieves the columns from the *findcar* result table and places the columns into a new variable named *carscol*. Use the PRINT statement to list your results from the TABCOLUMN function.

```
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.

Log 4.14 *SAS Log*

```
{_Index_=_Index_,Make=Make,Model=Model,Type=Type,Origin=Origin,DriveTrain=DriveTr
ain,MSRP=MSRP,Invoice=Invoice,EngineSize=

EngineSize,Cylinders=Cylinders,Horsepower=Horsepower,MPG_City=MPG_City,MPG_Highwa
y=MPG_Highway,Weight=Weight,Wheelbase=Wheelbase,Len
 gth=
 Length}
```

See Also

- FINDTABLE Function on page 78
- SORT_REV Function on page 90

• PRINT Statement on page 50

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 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.

Log 4.15 *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
- DESCRIBE Statement on page 29
- FINDTABLE Function on page 78
- SORT_REV Function on page 90

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-name-1*>, <*session-name-N*>);

Optional Argument

session-name

Specifies the session name to cancel.

TIP Multiple session names can be listed.

Example: Terminate a Parallel Session

```
term_parallel_session(casauto);
```

TIMESTAMP Function

Returns the current date and time.

Categories: Control
Status

Default: `string`

Returned data type: Date and Time

Syntax

TIMESTAMP (“<double>” “<int64>” “<string>”);

Optional Arguments

double
stores a signed, approximate, 64-bit double-precision, floating-point number. The DOUBLE data type stores numbers of large magnitude and permits computations that require many digits of precision to the right of the decimal point.

int64
a 64-bit signed, exact whole number, with a precision of 19 digits.

string
one or more consecutive alphanumeric characters, other keyboard characters, or both.

Example: Using the TIMESTAMP Function

You can use the TIMESTAMP function with no arguments to get the current date and time.

```
print timestamp();
```

The following is printed to the SAS log:

```
Wed May 23 23:58:54 2018
```

You can use the TIMESTAMP function with the **`double`** argument to get the current date and time as the **`double`** data type.

```
print timestamp("double");
```

The following is printed to the SAS log:

```
1842739185.1
```

You can use the TIMESTAMP function with the **`int64`** argument to get the current date and time as the **`int64`** data type.

```
print timestamp("int64");
```

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 the **string** data type.

```
print timestamp("string");
```

The following is printed to the SAS log:

```
Thu May 24 00:01:06 2018
```

See Also

References

- Frequently Used Data Types

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.

UUID Function

Returns the universally unique identifier (UUID) for the given session.

Categories: Control
Status

Returned data type: 36 character UUID

Syntax

UUID (*session-name*);

Required Argument

session-name
Specifies the session name to retrieve the UUID from.

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

```
 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;
```

Example 2: wait_for_next_action With Job Name

```
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;
```

Chapter 5

Common Functions

Dictionary

ABS Function

Returns the absolute value.

Syntax

ABS(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

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

Example

```
proc cas;
  x=abs(2.4);
  y=abs(-3);
 print "x=" x;
 print "y=" y;
run;
```

The following SAS statements produce these results:

SAS Statement	Result
`x=abs(2.4);`	`x=2.4`
`y=abs(-3);`	`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

```
proc cas;
  x=airy(2.0);
  y=airy(-2.0);
 print "x=" x;
 print "y=" y;
run;
```

The following SAS statements produce these results:

SAS Statements	Results
`x=airy(2.0);`	`0.0349241304`
`y=airy(-2.0);`	`0.2274074282`

ANYALNUM Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYALNUM function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYALNUM function searches a string for the first occurrence of any character that is a digit or an uppercase or lowercase letter. If such a character is found,

ANYALNUM returns the position in the string of that character. If no such character is found, ANYALNUM returns a value of 0.

If you use only one argument, 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: Scanning a String from Right to Left

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

```
proc cas;
   string='Next = Last + 1;';
   j=999999;
   do until(j=0);
     j=anyalnum(string, 1-j);
     if j=0 then print 'The end';
     else do;
       c=substr(string, j, 1);
     print "j=" j "c=" c;
         end;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
j=15 c=1
j=11 c=t
j=10 c=s
j=9 c=a
j=8 c=L
j=4 c=t
j=3 c=x
j=2 c=e
j=1 c=N
The end
```

ANYALPHA Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYALPHA function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYALPHA function searches a string for the first occurrence of any character that is an uppercase or lowercase letter. If such a character is found, ANYALPHA returns the position in the string of that character. If no such character is found, ANYALPHA returns a value of 0.

If you use only one argument, 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 nonalphabetic character.

Example: Searching a String for Alphabetic Characters

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

```
proc cas;
   string='Next = _n_ + 12E3;';
   j=1;
   do until(j=0);
      j=anyalpha(string, j+1);
      if j=0 then print "That's all";
      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=e
j=4 c=t
j=9 c=n
j=16 c=E
That's all
```

ANYCNTRL Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYCNTRL function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYCNTRL function searches a string for the first occurrence of a control character. If such a character is found, ANYCNTRL returns the position in the string of that character. If no such character is found, ANYCNTRL returns a value of 0.

If you use only one argument, 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.

ANYDIGIT Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

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

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.

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

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

ANYGRAPH Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYGRAPH function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYGRAPH function searches a string for the first occurrence of a graphical character. A graphical character is 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
```

ANYLOWER Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYLOWER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference 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
```

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.

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

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

ANYPRINT Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYPRINT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYPRINT function searches a string for the first occurrence of a printable character. If such a character is found, ANYPRINT returns the position in the string of that character. If no such character is found, ANYPRINT returns a value of 0.

If you use only one argument, 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 nonprintable 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
```

ANYPUNCT Function

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

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYPUNCT function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYPUNCT function searches a string for the first occurrence of a punctuation character. If such a character is found, ANYPUNCT returns the position in the string of that character. If no such character is found, ANYPUNCT returns a value of 0.

If you use only one argument, 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 or if it is missing, then

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

Comparisons

The ANYPUNCT function searches a character string for a punctuation character. The NOTPUNCT function searches a character string for a character that is not a punctuation character.

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
```

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.

Note: This function supports the VARCHAR type.

Syntax

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

Required Argument

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

Optional Argument

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

Details

The results of the ANYSPACE function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYSPACE function searches a string for the first occurrence of any character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, ANYSPACE returns the position in the string of that character. If no such character is found, ANYSPACE returns a value of 0.

If you use only one argument, 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
```

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

Details

The results of the ANYUPPER function depend directly on the translation table that is in effect (see "TRANTAB= System Option" in *SAS National Language Support (NLS): Reference Guide*) and indirectly on the ENCODING and LOCALE system options.

The ANYUPPER function searches a string for the first occurrence of an uppercase letter. If such a character is found, ANYUPPER returns the position in the string of that character. If no such character is found, ANYUPPER returns a value of 0.

If you use only one argument, 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
is an optional integer that specifies the position at which the search should start and the direction in which to search.

Details

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

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

SAS Statement	Result
`x=arcos(1);`	`0`
`x=arcos(0);`	`1.5707963268`
`x=arcos(-0.5);`	`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

SAS Statement	Result
`x=arsin(0);`	`0`
`x=arsin(1);`	`1.5707963268`
`x=arsin(-0.5);`	`-0.523598776`

ARSINH Function

Returns the inverse hyperbolic sine.

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.

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.

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:

SAS Statement	Result

`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.

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 $-\pi$ 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:

SAS statement	Result
`a=atan2(-1, 0.5);`	`-1.107148718`
`b=atan2(6, 8);`	`0.6435011088`
`c=atan2(5, -3);`	`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:

Statements	Results
`x=band(9,11);`	`9`
`x=band(15,5);`	`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:

Statements	Results
`x=beta(5,3);`	`0.0095238095238`
`x=beta(15,45);`	`1.6710294365008E-15`

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:

Statements	Results
	`----+----1----+---2--`
`a=blackclprc(50, .25, 48, .05, .25);`	`1.55130142723117`
`b=blackclprc(9, 1/12, 10, .05, .2);`	`1`

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:

Statements	Results
	`----+----1----+---2--`
`a=blackptprc(298, .25, 350, .06, .25);`	`1.85980563934969`
`b=blackptprc(145, .5, 170, .05, .2);`	`1.41234979911583`

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:

Statements	Results
	`----+----1----+---2--`
`a=blkshclprc(50, .25, 48, .05, .25);`	`1.79894201954462`
`b=blkshclprc(9, 1/12, 10, .05, .2);`	`1`

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:

Statements	Results
	`----+----1----+---2--`
`a=blkshptprc(230,.5,290,.04,.25);`	`1.56597442946066`
`b=blkshptprc(350,.3,400,.05,.2);`	`1.64091943067592`

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:

Statements	Results
`x=blshift(7,2);`	`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:

Statements	Results
`a=bnot(16);`	`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:

Statements	Results
`a=bor(4,8);`	`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:

Statements	Results
`x=brshift(64,2);`	`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:

Statements	Results
`x=bxor(128,64);`	`192`

BYTE Function

Returns one character in the ASCII or the EBCDIC collating sequence.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

BYTE(*n*)

Arguments

n

specifies a whole number that represents a specific ASCII or EBCDIC character.

Range	0–255
Data type	DOUBLE

Details

For EBCDIC collating sequences, *n* is between 0 and 255. For ASCII collating sequences, the characters that correspond to values between 0 and 127 represent the standard character set. Other ASCII characters that correspond to values between 128 and 255 are available on certain ASCII operating environments, but the information those characters represent varies with the operating environment.

Example

The following statement illustrates the BYTE function:

Statements	Results	
	ASCII	EBCDIC
	`----+----1----+----2`	`----+----1----+----2`
`x=byte(80);`	`P`	`&`

CAT Function

Does not remove leading or trailing blanks, and returns a concatenated character string.

Category: Character

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CAT(*item-1*<, …*item-n*>)

Arguments

item

specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CAT function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CAT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CAT function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CAT is called from the macro processor

If CAT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CAT finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank line in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CAT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators.

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: CHAR, NCHAR, NVARCHAR, 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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CATS(*item*<, …*item*>)

Arguments

item

specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CATS function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CATS function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATS function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses
- up to 65534 characters when CATS is called from the macro processor

If CATS returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATS finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CATS function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators.

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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CATT(*item*<, …*item*>)

Arguments

item

specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, leading blanks are removed and SAS does not write a note to the log.

Details

Length of Returned Variable

If the CATT function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the | | or the .. concatenation operator returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values that are being concatenated.

Length of Returned Variable: Special Cases

The CATT function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATT function has the following length:

- up to 200 characters in WHERE clauses and in PROC SQL
- up to 32767 characters in PROC DS2, except in WHERE clauses

- up to 65534 characters when CATT is called from the macro processor

If CATT returns a value in a temporary buffer, the length of the buffer depends on the calling environment, and the value in the buffer can be truncated after CATT finishes processing. In this case, SAS does not write a message about the truncation to the log.

If the length of the variable or the buffer is not large enough to contain the result of the concatenation, SAS does the following:

- changes the result to a blank value in PROC DS2 and in PROC SQL
- writes a warning message to the log stating that the result was either truncated or set to a blank value, depending on the calling environment
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation
- sets _ERROR_ to 1

The CATT function removes leading and trailing blanks from numeric arguments after it formats the numeric value with the BEST*w*. format.

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operators | | and .. , and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operators.

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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

CATX(*delimiter*, *item-1*<, …*item-n*>)

Arguments

delimiter
specifies a character string that is used as a delimiter between concatenated items.

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, SAS does not write a note to the log.

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: DECIMAL, DOUBLE, NUMERIC

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:

Statements	Results
`a=ceil(-2.4);`	`-2`
`b=ceil(1+1.e-11);`	`2`
`c=ceil(-1+1.e-11);`	`0`

Statements	Results
`d=ceil(1+1.e-13);`	`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:

Statements	Results
`a=ceilz(2.1);`	`3`
`b=ceilz(3);`	`3`
`c=ceilz(1+1.e-11);`	`2`
`d=ceilz(223.456);`	`224`
`e=ceilz(-223.456);`	`-223`

CHOOSEC Function

Returns a character value that represents the results of choosing from a list of arguments.

Returned data type: VARCHAR, NVARCHAR

Syntax

CHOOSEC(*index-expression*, *selection-1*<, …*selection-n*>)

Arguments

index-expression
specifies any valid expression that evaluates to a numeric value.

Data type DOUBLE

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.

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 statement illustrates the CMISS function:

Statements	Results
`x = CMISS(1,0, ' ' , 2,5,' ');`	1
`x = CMISS(1, ' ');`	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:

Statements	Results
`x = COALESCE(., .A, 33, 22, 44, .);`	33

COALESCEC Function

Returns the first non-null or nonmissing value from a list of character arguments.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COALESCEC(*expression*<, …*expression*>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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:

Statements	Results
`a=coalescec('', 'Hello');`	Hello
`a=coalescec('', 'Goodbye', 'Hello');`	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:

Statement	Result
`x=comb(27, 2);`	`351`

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: CHAR, NCHAR, NVARCHAR, 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 5.1 *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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

COMPBL(*character-expression*)

Arguments

character-expression
specifies any valid expression that evaluates or can be coerced to a character string and that specifies the character string to compress.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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.

Statements	Results
	`----+----1----+---2--`
`a=compbl('January  Status');`	`January Status`
`string='125  E. Main St.';` `street=compbl(string);`	`125 E. Main St.`

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 $a \geq 0$

Data type DOUBLE

f

specifies the future amount (at the end of *n* periods).

Range $f \geq 0$

Data type DOUBLE

r

specifies the periodic interest rate expressed as a fraction.

Range $r \geq 0$

Data type DOUBLE

n

specifies the number of compounding periods.

Range $n \geq 0$

Data type DOUBLE

Details

The COMPOUND function returns the missing argument in the list of four arguments from a compound interest calculation. The arguments are related by the following equation:

$$f = a(1 + r)^n$$

One missing argument must be provided. A compound interest parameter is then calculated from the remaining three values. No adjustment is made to convert the results to round numbers.

If n=0, then

$$f = a$$

and

$$(1 + r)^n$$

is equal to 1.

Note: If you choose r as your missing value, then COMPOUND returns an error.

Example

The accumulated value of an investment of $2000 at a nominal annual interest rate of 9%, compounded monthly after 30 months, can be expressed as follows:

```
future=compound(2000, ., 0.09/12, 30);
```

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: CHAR, NCHAR, NVARCHAR, 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

These examples show how to remove blanks from a character string.

Statements	Results
	`----+----1`
`a='AB C D ';` `b=compress(a);`	`ABCD`
`a='AB C D';` `b=compress(a,'A ');`	`BCD`

Example 2: Compressing Vowels

These examples show how to remove characters from a string.

Statements	Results
	`----+----1----+----2----+----3`
`x='123-4567-8901 e 234-5678-9012 i';` `y=compress(x,'aeiou');`	`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
`'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

Constant	Description
`'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.

Range	$2 \le nbytes \le 8$
Default	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 $0 < k_0 \leq \frac{1}{n}$

$0 < k_0 \leq 1/n$

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:

Statements	Results
`x=cos(0.5);`	`0.87758256189037`
`x=cos(0);`	`1`
`x=cos(3.14159/3);`	`0.50000076602519`

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:

Statements	Results
`x=cosh(0);`	`1`
`x=cosh(-5.0);`	`74.2099485247878`
`x=cosh(4.37);`	`39.5281414700662`
`x=cosh(0.5);`	`1.12762596520638`

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:

Statements	Results
`x=cot(0.5);`	`1.83048772171245`
`x=cot(1);`	`0.64209261593433`
`x=cot(3.14159/3);`	`0.57735144856346`

Note: If you use **`x=cot(0);`**, then the COT function returns a missing value, and a note is written to the log that indicates you entered an invalid argument to the function. This is the correct behavior.

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:

Statements	Results

`xyz='This is a thistle? Yes, this is a thistle.';`
`howmanythis=count(xyz,'this');`
`put howmanythis;`
	3

`xyz='This is a thistle? Yes, this is a thistle.';`
`howmanyis=count(xyz,'is');`
`put howmanyis;`
	6

`howmanythis_i=count('This is a thistle? Yes, this is a thistle.'`
`  ,'this','i');`
`put howmanythis_i;`
	4

`variable1='This is a thistle? Yes, this is a thistle.';`
`variable2='is ';`
`variable3='i';`
`howmanyis_i=count(variable1,variable2,variable3);`
`put howmanyis_i;`
	4

`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;`
	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 5.2 *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 5.3 *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:

Statements	Results
`x=csc(0.5);`	`2.08582964293348`
`x=csc(1);`	`1.18839510577812`
`x=csc(3.14159/3);`	`1.1547011280666`

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:

Statements	Results
`a=css(5,9,3,6);`	`18.75`
`b=css(5,8,9,6,.);`	`10`
`c=css(8,9,6,.);`	`4.66666666666666`

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:

Statements	Results
`x1=cv(5,9,3,6);`	`43.4782608695652`
`x2=cv(5,8,9,6,.);`	`26.082026547865`
`x3=cv(8,9,6,.);`	`19.9242421519819`

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:

Statements	Results
`x=dairy(2.0);`	`-0.05309038443365`
`x=dairy(-2.0);`	`0.61825902074169`

DATDIF Function

Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.

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:

Statements	Results
`x=date();`	`18773`

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:

Statements	Results
`x=datejul(11365);`	`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

The following statement illustrates the DATEPART function where the variable **`dtvalue`**, a SAS datetime value, has a value of 1652165417:

Statements	Results
`dp=put(datepart(dtvalue),date9.);`	`09MAY2012`

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

The following statement illustrates the DATETIME function:

Statements	Results
`dt=datetime();`	`1622021468`

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

The following statement illustrates the DAY function where **`dayvalue`**, the SAS date value, has a value of 17531, which is December 31, 2007:

Statements	Results
`dt=day(dayvalue);`	`dt=31`

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 CHAR, NCHAR, NVARCHAR, 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. 206)	`'BERNOULLI' | 'BERN'`
Binomial (p. 207)	`'BINOMIAL' | 'BINO'`
Gamma (p. 207)	`'GAMMA'`
Inverse Gauss (Wald) (p. 208)	`'IGAUSS' | 'WALD'`
Normal (p. 208)	`'NORMAL' | 'GAUSSIAN'`
Poisson (p. 209)	`'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 $variable \geq \varepsilon$

μ
is a numeric mean parameter.

Range $\mu \geq \varepsilon$

ε
is an optional positive numeric value that is used to bound *variable* and μ. Any value of *variable* in the interval $0 \leq variable \leq \varepsilon$ is replaced by ε. Any value of μ in the interval $0 \leq \mu \leq \varepsilon$ is replaced by ε.

The DEVIANCE function returns the deviance from a gamma distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the gamma distribution, where x is the random variable:

$$\text{DEVIANCE}(\text{'}GAMMA\text{'}, x, \mu) = \begin{cases} . & x < 0 \\ 2\left(-\log\left(\frac{x}{\mu}\right) + \frac{x-\mu}{\mu}\right) & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

The Inverse Gauss (Wald) Distribution

DEVIANCE('IGAUSS' | 'WALD', *variable*, *μ*<, *ε*>)

Arguments

variable
is a numeric random variable.

Range $variable \geq \varepsilon$

μ
is a numeric mean parameter.

Range $\mu \geq \varepsilon$

ε
is an optional positive numeric value that is used to bound *variable* and μ. Any value of *variable* in the interval $0 \leq variable \leq \varepsilon$ is replaced by ε. Any value of μ in the interval $0 \leq \mu \leq \varepsilon$ 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:

$$\text{DEVIANCE}(\text{'}IGAUSS\text{'}, x, \mu) = \begin{cases} . & x < 0 \\ \frac{(x-\mu)^2}{\mu^2 x} & x \geq \varepsilon, \mu \geq \varepsilon \end{cases}$$

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 $variable \geq 0$

μ
is a numeric mean parameter.

Range $\mu \geq \varepsilon$

ε
is an optional positive numeric value that is used to bound μ. Any value of μ in the interval $0 \leq \mu \leq \varepsilon$ is replaced by ε.

The DEVIANCE function returns the deviance from a Poisson distribution with a mean parameter μ. The following equation describes the DEVIANCE function for the Poisson distribution, where x is the random variable:

$$\text{DEVIANCE}('POISSON', x, \mu) = \begin{cases} . & x < 0 \\ 2\left(x log\left(\frac{x}{\mu}\right) - (x - \mu)\right) & x \geq 0, \mu \geq \varepsilon \end{cases}$$

DHMS Function

Returns a SAS datetime value from date, hour, minute, and second values.

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:

Statements	Results
`x=digamma(1.0);`	`-0.57721566490153`

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 5.1 *Values That Are Returned by the DIVIDE Function*

		x						
		positive	zero	negative	.I	.M	.__	other
	positive	x/y or .I	0	x/y or .M	.I	.M	.__	x
	zero	.I	.	.M	.I	.M	.__	x
	negative	x/y or .M	0	x/y or .I	.M	.I	.__	x
y	.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 $0 \leq c < 1$

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

$\boldsymbol{k_0}$

specifies the time from the present date to the first coupon date, expressed in terms of the number of periods.

Range $0 < k_0 \leq 1/n$

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

  ```
  effective-rate1 = EFFRATE('MONTH', 10);
  ```
- If a nominal rate is 10%, then the corresponding effective rate when interest is compounded quarterly can be expressed as

  ```
  effective-rate2 = EFFRATE('QUARTER', 10);
  ```

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

The following statements illustrate the ERF function:

Statement	Result
`y=erf(1.0);`	`0.842701`
`y=erf(-1.0);`	`-0.8427`

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:

Statement	Result
`x=erfc(1.0);`	`0.157299`

Statement	Result
`x=erfc(-1.0);`	`1.842701`

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:

Statements	Results
`a=exp(1.0);`	`2.71828182845904`
`a=exp(0);`	`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:

Statement	Result
`x=fact(5);`	`120`

FIND Function

Searches for a specific substring of characters within a character string.

Returned data type: CHAR

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:

Statements	Results
`whereisshe=find('She sells seashells? Yes, she does.','she ');` `print whereisshe;`	27

`variable1='She sells seashells? Yes, she does.';`
`variable2='she ';`
`variable3='i';`
`whereisshe_i=find(variable1,variable2,variable3);`
`print whereisshe_i;`
	1

`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;`
	14

`xyz='She sells seashells? Yes, she does.';`
`startposvar=22;`
`whereisshe_22=find(xyz,'she',startposvar);`
`print whereisshe_22;`
	27

`xyz='She sells seashells? Yes, she does.';`
`startposexp=1-23;`
`whereisShe_ineg22=find(xyz,'She','i',startposexp);`
`print whereisShe_ineg22;`
	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: DECIMAL, DOUBLE, NUMERIC

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:

Statements	Results
`floor(1.95);`	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.

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:

Statements	Results
`var1=2.1;` `a=floorz(var1);`	`2`
`b=floorz(-2.4);`	`-3`
`c=floorz(-1.6);`	`-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, $I(\ldots)$ is the probability from the beta distribution that is given by the following equation:

$$I_x(a,b) = \frac{}{\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}} \int_0^x t^{a-1}(1-t)^{b-1} dt$$

If the algorithm fails to converge to a fixed point, a missing value is returned.

Example

```
proc 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;
```

Figure 5.2 *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:

Statements	Results
`var1=5.9999999999999;` `x=print(fuzz(var1),16.14);`	`6`
`x=print(fuzz(5.99999999), 16.14);`	`5.99999999`

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 $0 \le p < 1$

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 pth quantile from the gamma distribution, with shape parameter a. The probability that a row from a gamma distribution is less than or equal to the returned quantile is p.

Note: GAMINV is the inverse of the PROBGAM function.

Example

The following statements illustrate the GAMINV function:

Statement	Result
`q1=gaminv(0.5, 9);`	`8.668951`
`q2=gaminv(0.1, 2.1);`	`0.584193`

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.

$$\text{GAMMA}(x) = \int_0^\infty t^{x-1} e^{-t}\, dt.$$

For positive integers, GAMMA(x) is (x − 1)!. This function is commonly denoted by $\Gamma(x)$.

Example

The following statement illustrates the GAMMA function:

Statements	Results
`x=gamma(6);`	`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

$\boldsymbol{R}_{\mathrm{d}}$

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

$\boldsymbol{R}_{\mathrm{f}}$

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:

$$\mathrm{CALL} = SN(d_1)\left(\varepsilon^{-R_f t}\right) - EN(d_2)\left(\varepsilon^{-R_d t}\right)$$

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:

Statements	Results
	`----+----1----+---2--`
`a=garkhclprc(40, .5, 38, .06, .04, .2);`	`1.44942510595479`
`c=garkhclprc(19, .25, 20, .05, .03, .09);`	`1.1304209447635`

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:

Statements	Results
	`----+----1----+---2--`
`a=garkhptprc(50, .7, 55, .05, .04, .2);`	`1.4050880944848`
`b=garkhptprc(32, .3, 33, .05, .03, .3);`	`1.56473205137371`

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:

Statements	Results
`x=gcd(5,15)`	`5`
`x=gcd(36,45)`	`9`

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* <,*option(s)*>)

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

option(s)
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 nth root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when floating-point arithmetic is used. Therefore, GEOMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the GEOMEANZ function.

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEAN function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEANZ, GEOMEAN fuzzes the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEAN function:

Statements	Results
`x1=geomean(1,2,2,4);`	2
`x2=geomean(.,2,4,8);`	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 nth root of the product of the values:

$$\sqrt[n]{(x_1 * x_2 * \ldots * x_n)}$$

Equivalently, the geometric mean is shown in this equation.

$$\exp\left(\frac{(\log(x_1) + \log(x_2) + \ldots + \log(x_n))}{n}\right)$$

Comparisons

The MEAN function returns the arithmetic mean (average), and the HARMEAN function returns the harmonic mean, whereas the GEOMEANZ function returns the geometric mean of the non-null or nonmissing values. Unlike GEOMEAN, GEOMEANZ does not fuzz the values of the arguments that are approximately zero.

Example

The following statements illustrate the GEOMEANZ function:

Statements	Results
`x1=geomeanz(1,2,2,4);`	2
`x2=geomeanz(.,2,4,8);`	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:

Statements	Results
`x1=harmean(1,2,4,4);`	2
`x2=harmean(.,4,12,24);`	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:

Statements	Results
`x1=harmeanz(1,2,4,4);`	2
`x2=harmeanz(.,4,12,24);`	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:

Statements	Results
`a=hms(12,45,10);` `b=put(a, time.);`	`45910` `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 5.1 *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
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

Holiday Value	Description	Date Celebrated
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:

Statements	Results

`proc cas;`
`thanks = holiday('thanksgiving', 2018);`
`format thanks date9.;`
`print thanks;`
	`22NOV2018`

Statements	Results

`proc cas;`
`boxing = holiday('boxing', 2018);`
`format boxing date9.;`
`print boxing;`
	26DEC2018

`proc cas;`
`easter = holiday('easter', 2018);`
`format easter date9.;`
`print easter;`
	01APR2018

`proc cas;`
`canada = holiday('canada', 2018);`
`format canada date9.;`
`print canada;`
	01JUL2018

`proc cas;`
`fathers = holiday('fathers', 2018);`
`format fathers date9.;`
`print fathers;`
	17JUN2018

`proc cas;`
`valentines = holiday('valentines', 2018);`
`format valentines date9.;`
`print valentines;`
	14FEB2018

`proc cas;`
`victoria = holiday('victoria', 2018);`
`format victoria date9.;`
`print victoria;`
	21MAY2018

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:

Statements	Results
`a=hour(time());`	`10`

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 $nu \geq 0$

Data type DOUBLE

x

specifies a numeric constant, variable, or expression.

Range $x \geq 0$

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:

Statements	Results
`x=ibessel(2, 2, 0);`	`0.68894844769873`
`x=ibessel(2, 2, 1);`	`0.09323903330473`

IFC Function

Returns a character value based on whether an expression is true, false, or missing.

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.

IFN Function

Returns a numeric value based on whether an expression is true, false, or missing.

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.

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: INTEGER

Syntax

INDEX(*target-expression*, *search-expression*)

Arguments

target-expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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:

Statements	Results

`a='ABC.DEF (X=Y)';`
`b='X=Y';`
`c=index(a,b);`
	`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:

Statements	Results

`a='ABC.DEP (X2=Y1)';`
`b='()';`
`c='=.';`
`b=indexc(a,'0123',';()=.');`
`b=indexc(a,b);`
`b=indexc(a,b,c);`
	4
	8
	4

`c='have a good day';` `d=indexc(c,'pleasant','very');`	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.

Statements	Results

`a='The power to know.';`
`word='power';`
`c=indexw(a,word);`
	5

`a='The power to know.';` `b=indexw(a,'know');`	0
`a='The power to know.';` `b=indexw(a,'know','.');`	14
`a='abc,def@ xyz';` `b=indexw(a,',', '@');`	0
`a="abc,def@ xyz";` `b=indexw(a,'def', '@,');`	5

Statements	Results
`x="abc,def@ xyz";` `xyz=indexw(x, ' xyz', '@');`	`10`

INPUTC Function

Enables you to specify a character informat at run time.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INPUTC(*source*, *informat*<, *w*>)

Arguments

source
specifies a character constant, variable, or expression to which you want to apply the informat.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

informat
is a character constant, variable, or expression that contains the character informat that you want to apply to *source*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

w
specifies any valid expression that evaluates to a numeric width to apply to the informat.

Interaction If you specify a width here, it overrides any width specification in the informat.

Data type DOUBLE

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:

Statements	Results
`var1=2.1;` `a=int(var1);`	2
`a=int(-2.4);`	-2
`a=int(1+1.e-11);`	1
`a=int(-1.6);`	-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:

Statements	Results

`proc cas;`
`  cycle_index1=intcindex('day', 45910);`
`  print "cycle_index1=" cycle_index1;`
`run;`
	`cycle_index1=37`

`proc cas;`
`  cycle_index1=intcindex('dtqtr', 45910);`
`  print "cycle_index1=" cycle_index1;`
`run;`
	`cycle_index1=1`

Statements	Results

`proc cas;`
`  cycle_index1=intcindex('tenday', 45910);`
`  print "cycle_index1=" cycle_index1;`
`run;`
	`cycle_index1=1`

`proc cas;`
`  cycle_index1=intcindex('tenday', 45910);`
`  print "cycle_index1=" cycle_index1;`
`run;`
	`cycle_index1=13`

`proc cas;`
`  cycle_index1=intcindex('semimonth', 45910);`
`  print "cycle_index1=" cycle_index1;`
`run;`
	`cycle_index1=1`

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 5.2 *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-day work week with a Saturday-Sunday weekend). *days* identifies the weekend days by number (1=Sunday ... 7=Saturday). By default, *days*=17.			WEEKDAY35W	Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)
	TENDAY	Ten-day intervals (a U.S. automobile industry convention)	First, 11th, and 21st of each month	Ten-day periods	TENDAY4.2	Four ten-day periods starting at the second TENDAY period
	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

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
	MONTH	Monthly intervals	First of each month	Months	MONTH2.2	February-March, April-May, June-July, August-September, October-November, and December-January of the following year
	QTR	Quarterly (three-month) intervals	January 1	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:

Statements	Results

`proc cas;`
`  qtr=intck('qtr', 21185, 21366);`
`  print "qtr=" qtr;`
`run;`
	`qtr=2`

`proc cas;`
`  year=intck('year', 21185, 21550);`
`  print "year=" year;`
`run;`
	`year=1`

`proc cas;`
`  year=intck('year', 21185, 21489);`
`  print "year=" year;`
`run;`
	`year=0`

`proc cas;`
`  semiyear=intck('semiyear', 21185, 21915);`
`  print "semiyear=" semiyear;`
`run;`
	`semiyear=4`

`proc cas;`
`  weekvar=intck('week2.2', 21185, 21550);`
`  print "weekvar=" weekvar;`
`run;`
	`weekvar=26`

`proc cas;`
`  wdvar=intck('weekday7w', 21185, 21216);`
`  print "wdvar=" wdvar;`
`run;`
	`wdvar=27`

`proc cas;`
`  newyears=intck('year', 17900, 21550);`
`  print "newyears=" newyears;`
`run;`
	`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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTCYCLE({*interval*<*multiple*><*.shift-index*>}<, *seasonality*>)

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Note The possible values of *interval*

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:

Statements	Results

`proc cas;`
`  cycle_year=intcycle('year');`
`  print "cycle_year=" cycle_year;`
`run;`
	`cycle_year=YEAR`

`proc cas;`
`  cycle_quarter=intcycle('qtr');`
`  print "cycle_quarter=" cycle_quarter;`
`run;`
	`cycle_quarter=YEAR`

`proc cas;`
`  cycle_month=intcycle('month', 3);`
`  print "cycle_month=" cycle_month;`
`run;`
	`cycle_month=QTR`

`proc cas;`
`  cycle_month=intcycle('month');`
`  print "cycle_month=" cycle_month;`
`run;`
	`cycle_month=YEAR`

`proc cas;`
`  cycle_weekday=intcycle('weekday');`
`  print "cycle_weekday=" cycle_weekday;`
`run;`
	`cycle_weekday=WEEK`

Statements	Results

`proc cas;`
`  cycle_weekday2=intcycle('weekday', 5);`
`  print "cycle_weekday2=" cycle_weekday2;`
`run;`
	`cycle_weekday2=WEEK`

`proc cas;`
`  cycle_day=intcycle('day');`
`  print "cycle_day=" cycle_day;`
`run;`
	`cycle_day=WEEK`

`proc cas;`
`  cycle_day2=intcycle('day', 10);`
`  print "cycle_day2=" cycle_day2;`
`run;`
	`cycle_day2=TENDAY`

`proc cas;`
`  cycle_second=intcycle('second');`
`  print "cycle_second=" cycle_second;`
`run;`
	`cycle_second=DTMINUTE`

INTFIT Function

Returns a time interval that is aligned between two dates.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTFIT(*expresssion-1*, *expression-2*, '*type*')

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string and that represents a SAS date or datetime value.

Data type DOUBLE

'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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTGET(*date-1*, *date-2*, *date-3*)

Argument

date

specifies any valid expression that evaluates to a SAS date or datetime value.

Data type DOUBLE

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:

Statements	Results

`sasdate1=to_double(date'2013-01-01');`
`sasdate2=to_double(date'2014-01-01');`
`sasdate3=to_double(date'2014-05-01');`
`c=intget(sasdate1, sasdate2, sasdate3);`
`put c;`
	MONTH4

`sasdate1=to_double(date'2012-02-29');`
`sasdate2=to_double(date'2014-02-28');`
`sasdate3=to_double(date'2016-02-29');`
`c=intget(sasdate1, sasdate2, sasdate3);`
`put c;`
	YEAR2.2

`sasdate1=to_double(date'2013-02-01');`
`sasdate2=to_double(date'2013-02-16');`
`sasdate3=to_double(date'2013-03-01');`
`c=intget(sasdate1, sasdate2, sasdate3);`
`put c;`
	SEMIMONTH

`sasdate1=to_double(date'2013-01-02');`
`sasdate2=to_double(date'2014-02-02');`
`sasdate3=to_double(date'2015-03-02');`
`c=intget(sasdate1, sasdate2, sasdate3);`
`put c;`
	MONTH13.13

Statements	Results

`sasdate1=to_double(date'2013-02-10');`
`sasdate2=to_double(date'2013-02-19');`
`sasdate3=to_double(date'2013-02-28');`
`c=intget(sasdate1, sasdate2, sasdate3);`
`put c;`
	`DAY9.5`

`sasdate1=to_double(timestamp'2014-04-01 01:03:00.0000');`
`sasdate2=to_double(timestamp'2014-04-01 01:04:00.0000');`
`sasdate3=to_double(timestamp'2014-04-01 01:05:00.0000');`
`c=intget(sasdate1, sasdate2, sasdate3);`
`put c;`
	`MINUTE`

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:

Statements	Results

`sasdate1=to_double(date'2013-08-14');`
`interval1 = intindex('qtr', sasdate1);`
`put interval1;`
	3

`sasts1=to_double(timestamp'2013-12-23 15:09:19');`
`interval2 = intindex('dtqtr', sasts1);`
`put interval2;`
	4

`sastime1=to_double(time'09:05:15');`
`interval3 = intindex('hour', sastime1);`
`put interval3;`
	10

Statements	Results

`sasdate1=to_double(date '2013-02-26');`
`interval4 = intindex('month', sasdate1);`
`put interval4;`
	2

`sasts1=to_double(timestamp'2013-05-28 05:15:00');`
`interval5 = intindex('dtmonth', sasts1);`
`put interval5;`
	5

`sasdate1=to_double(date '2013-09-09');`
`interval6 = intindex('week', sasdate1);`
`put interval6;`
	37

`sasdate1=to_double(date '2013-04-16');`
`interval7 = intindex('tenday', sasdate1);`
`put interval7;`
	11

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.

Examples

Example 1: Using the INTNX Function

The following statements illustrate the INTNX function.

Statements	Results

`sasdate1 = to_double(date'2013-02-05');`
`yr=intnx('year', sasdate1, 3);`
`print yr;`
`print put yr date7.;`
	20454
	01Jan16

`sasdate1 = to_double(date'2013-01-05');`
`x=intnx('month', sasdate1, 0);`
`print x;`
`print put x date7.;`
	19359
	01JAN13

`sasdate1 = to_double(date'2013-01-01');`
`next=intnx('semiyear', sasdate1, 1);`
`print next;`
`print put next date7.;`
	19540
	01JUL13

`sasdate1 = to_double(date'2012-08-01');`
`past=intnx('month2', sasdate1, -1);`
`print past;`
`print put past date7.;`
	19114
	01MAY12

`sasdate1 = to_double(date'2013-04-01');`
`sm=intnx('semimonth2.2', sasdate1, 4);`
`print sm;`
`print put sm date7.;`
	19555
	16JUL13

`x='month';`
`sasdate1 = to_double(date'2013-06-01');`
`nextmon=intnx(x, sasdate1, 1);`
`print nextmon;`
`print put nextmon date7.;`
	19540
	01JUL13

Statements	Results

`m1='month     ';`
`m2=trim(m1);`
`sasdate1 = to_double(date'2012-06-15');`
`x=intnx(m2, sasdate1, 1);`
`print x;`
`print put x date7.;`
	19175
	01JUL12

Example 2: Using the ALIGNMENT Argument

The following examples show the results of advancing a date by using the optional *alignment* argument.

Statements	Results

`sasdate1 = to_double(date'2013-01-01');`
`x=intnx('month', sasdate1, 5, 'beginning');`
`print x;`
`print put x date7.;`
	19510
	01JUN13

`sasdate1 = to_double(date'2013-01-01');`
`x=intnx('month', sasdate1, 5, 'middle');`
`print x;`
`print put x date7.;`
	19524
	15JUN13

`sasdate1 = to_double(date'2013-01-01');`
`x=intnx('month', sasdate1, 5, 'end');`
`print x;`
`print put x date7.;`
	19539
	30JUN13

`sasdate1 = to_double(date'2013-01-01');`
`x=intnx('month', sasdate1, 5, 'sameday');`
`print x;`
`print put x date7.;`
	19510
	01JUN13

`sasdate1 = to_double(date'2013-03-15');`
`x=intnx('month', sasdate1, 5, 'same');`
`print x;`
`print put x date7.;`
	19585
	15AUG13

`interval='month';`
`align='m';`
`sasdate1 = to_double(date'2013-09-01');`
`x=intnx(interval, sasdate1,2, align);`
`print x;`
`print put x date7.;`
	19667
	15NOV13

`m1='month     ';`
`m2=trim(m1);`
`sasdate1 = to_double(date'2012-09-01');`
`x=intnx(m2, sasdate1, 2, 'm');`
`print x;`
`print put x date7.;`
	19312
	15NOV12

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.

$$r = \begin{cases} \frac{1}{x^{freq}} - 1 & freq > 0 \\ -log_{\varepsilon}(x) & freq = 0 \end{cases}$$

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

```
rate=intrr(1,-400,100,200,300);
```

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:

Statements	Results
`cycle_years = intseas('year');` `print cycle_years;`	1
`cycle_smiyears = intseas('semiyear');` `print cycle_smiyears;`	2
`cycle_quarters = intseas('quarter');` `print cycle_quarters;`	4
`cycle_number = intseas('month', 'qtr');` `print cycle_number;`	3
`cycle_months = intseas('month');` `print cycle_months;`	12
`cycle_smimonths = intseas('semimonth');` `print cycle_smimonths;`	24
`cycle_tendays = intseas('tenday');` `print cycle_tendays;`	36
`cycle_weeks = intseas('week');` `print cycle_weeks;`	52

Statements	Results
`cycle_wkdays = intseas('weekday');` `print cycle_wkdays;`	5
`cycle_hours = intseas('hour');` `print cycle_hours;`	24
`cycle_minutes = intseas('minute');` `print cycle_minutes;`	60
`cycle_month2 = intseas('month2.2');` `print cycle_month2;`	6
`cycle_week2 = intseas('week2');` `print cycle_week2;`	26
`cycle_var1 = intseas(month4.3);` `print cycle_var1;`	3
`cycle_day1 = intseas('day1');` `print cycle_day1;`	7

See Also

Other References:

- *SAS/ETS User's Guide*

INTSHIFT Function

Returns the shift interval that corresponds to the base interval.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

INTSHIFT(*interval*<*multiple*><.*shift-index*>)

Arguments

interval

specifies a character constant, a variable, or an expression that evaluates or can be coerced to a character string and that contains an interval name such as WEEK, MONTH, or QTR.

Data type	CHAR, NCHAR, NVARCHAR, VARCHAR
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:

Statements	Results
`shift1=intshift('year');` `print shift1;`	`MONTH`
`shift2=intshift('dtyear');` `print shift2;`	`DTMONTH`
`shift3=intshift('minute');` `print shift3;`	`DTMINUTE`
`shift4 = intshift('weekdays');` `print shift4;`	`WEEKDAY`

Statements	Results
`shift5=intshift('weekday5.4');` `print shift5;`	WEEKDAY
`shift6=intshift('qtr');` `print shift6;`	MONTH
`shift7=intshift('dttenday');` `print shift7;`	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.

Statements	Results
`test1 = inttest('month');` `print test1;`	1
`test2 = inttest('week6.13');` `print test2;`	1
`test3 = inttest('tenday');` `print test3;`	1
`test4 = inttest('twoweeks');` `print test4;`	0
`test5 = inttest('hour2.2');` `print 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:

Statements	Results
`var1=2.1;` `a=intz(var1);`	`2`
`a=intz(-2.4);`	`-2`
`a=intz(1+1.e11);`	`1`
`a=intz(-1.6);`	`-1`

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:

```
InterestPaid1 = ipmt(0.1/12, 1, 36, 8000);
```

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:

- `InterestPaid2 = ipmt(0.1/12, 1, 36, 8000, 0, 1);`

  This computation returns a value of 0.

- `InterestPaid3 = ipmt(0.1, 3, 3, 8000);`

  This computation returns a value of 292.447129909366.

- `InterestPaid4 = ipmt(0.09/12, 359, 360, 125000, 0, 1);`

  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:

Statements	Results
`a=iqr(2,4,1,3,999999);`	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

```
rate=irr(1,-400,100,200,300);
```

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:

Statements	Results
`x=jbessel(2, 2);`	`0.35283402861563`

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:

Statements	Results
`julian=juldate(mdy(12,31,2013));`	`7365`
`julian=print(juldate(mdy(12,31,2013)),z5.);`	`07365`
`julian=juldate(mdy(9,1,1999));`	`99244`
`julian=juldate(mdy(7,1,1886));`	`1886182`

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:

Statements	Results
`julian=juldate7(mdy(12,31,2006));`	`2007365`
`julian=juldate7(mdy(12,31,2016));`	`2016366`

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:

Statements	Results
`a=kurtosis(5,9,3,6);`	`0.92799999999999`
`b=kurtosis(5,8,9,6,.);`	`-3.3`
`c=kurtosis(8,9,6,1);`	`1.5`
`d=kurtosis(8,1,6,1);`	`-4.48337950138504`

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:

Statements	Results
`k=1;` `largest1=largest(k, 456, 789, .Q, 123);`	`789`
`k=2;` `largest2=largest(k, 456, 789, .Q, 123);`	`456`
`k=3;` `largest3=largest(k, 456, 789, .Q, 123);`	`123`
`k=4;` `largest4=largest(k, 456, 789, .Q, 123);`	`.`

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:

Statements	Results
`a=lcm(1,5,3,0);`	`0`
`b=lcm(25,70,85,130);`	`77350`
`c=lcm(33,78);`	`858`

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:

Statements	Results
`x=lcomb(5000, 500);`	`1621.44113611415`
`y=lcomb(100, 10);`	`30.4823233622786`

LEFT Function

Left aligns a character expression.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

LEFT(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

LEFT returns a character string with leading blanks moved to the end of the value.

Example

The following statements illustrate the LEFT function:

Statements	Results
	`----+----1----+---2--`
`a='  END-OF-YEAR';` `b=left(a);`	`END-OF-YEAR`

LENGTH Function

Returns the length of a character string, excluding trailing blanks, in characters.

Alias: LENGTHN

Returned data type: BIGINT, INTEGER

Syntax

LENGTH(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The LENGTH function returns an integer that represents the position of the rightmost non-blank character or number in *expression*. If the value of *expression* is a blank-filled or an empty string, LENGTH returns a value of 0. If *expression* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string.

Comparisons

- The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks. LENGTH always returns a value that is less than or equal to the value returned by LENGTHC.
- The LENGTH function returns the length of a character string in characters, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. LENGTH always returns a value that is less than or equal to the value returned by LENGTHM. However, with an expression argument, LENGHTM always returns a null value, whereas LENGTH returns the length, in characters, of the character value.

Example

The following statements illustrate the LENGTH function:

Statements	Results
`a=length('ABCDEF    ');`	`10`
`b=date();` `a=length(b);`	* `5`
`a=length('  ');`	`0`
`a=length(.);`	** `1`

* The data type for *b* is converted to NCHAR with a value of 20943 whose length is 5.

** The data type for the null or missing value (.) is converted from DOUBLE to NCHAR with a value of . whose length is 1.

LENGTHC Function

Returns the length of a character string, including trailing blanks, in characters.

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::

Statements	Results

`proc cas;`
`a=lengthc('string with trailing blanks    ');`
`print "results=" a;`
	`results=31`

`proc cas;`
`string1='The power to know.  ';`
`a=lengthc(string1);`
`print "results=" a;`
	`results=20`

Statements	Results

`proc cas;`
`  a=lengthc('');`
`  print "results=" a;`
	`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:

Statements	Results

`proc cas;`
`a='ABCDEF    ';`
`b=lengthm(a);`
`print b;`
	`9`

`proc cas;`
`string1='the power to know.  ';`
`a=lengthm(string1);`
`print a;`
	`20`

LENGTHN Function

Returns the length of a character string, excluding trailing blanks, in characters.

Alias: LENGTH

Returned data type: BIGINT, INTEGER

Syntax

LENGTHN(*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 LENGTHN function returns an integer that represents the position of the rightmost non-blank character in *expression*. If the value of *expression* is blank-filled or an empty string, LENGTHN returns a value of 0. If *expression* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string.

Comparisons

- The 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:

Statements	Results
`c=lengthn('  abc  ');` `print c;`	6
`d=lengthn('abc  ');` `print d;`	3
`e=lengthn(18);` `print e;`	2
`f=lengthn(' ');` `print f;`	0

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:

Statements	Results
`x=lfact(5000);`	37591.1435088767
`y=lfact(100);`	363.739375555563

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:

Statements	Results
`a=lgamma(2);`	`0`
`a=lgamma(1.5);`	`-0.12078223763524`

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:

Statements	Results
`a=log(1.0);`	`0`
`a=log(10.0);`	`2.30258509299404`

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:

Statements	Results
`a=log10(1.0);`	`0`
`a=log10(10.0);`	`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:

Statements	Results
`a=log2(8.0);`	3
`a=log2(4);`	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:

Statements	Results
`x = LOGISTIC(.5);`	`0.62245933120185`
`x = LOGISTIC(7.3);`	`0.99932491726936`

LOWCASE Function

Converts all letters in a character expression to lowercase.

Alias: LOWER

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

LOWCASE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

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:

Statements	Results
`a=lowcase('INTRODUCTION');`	`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:

Statements	Results
`c=mad(2, 4, 1, 3, 5, 999999);` `print c;`	`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:

Statements	Results
`a=margrclprc(15, .5, 13, .06, .05, 1);` `print a;`	2
`b=margrclprc(2, .25, 1, .3, .2, 1);` `print b;`	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:

Statements	Results
`a=margrptprc(2, .25, 3, .06, .2, 1);` `print a;`	`1.00000000009729`
`b=margrptprc(3, .25, 4, .05, .3, 1);` `print b;`	`1.00157624907711`

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:

Statements	Results
`x=max(8,3);`	8
`x=max(2, 6, .);`	6
`x=max(2, -3, 1, -1);`	2
`x=max(3, ., -3);`	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:

Note: CASL outputs the birthday date as a SAS date value.

Statements	Results

`proc cas;`
`  mn=8;`
`  dy=27;`
`  yr=12;`
`  birthday=mdy(mn, dy, yr);`
`  print "birthday=" birthday;`
`run;`
	`birthday=19232`

`proc cas;`
`  mn=7;`
`  dy=11;`
`  yr=12;`
`  anniversary=mdy(mn, dy, yr);`
`  print "anniversary=" anniversary;`
`run;`
	`anniversary=19185`

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:

Statements	Results
`a=mean(2,.,.,6);`	`4`

Statements	Results
`a=mean(1,2,3,2);`	`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:

Statements	Results
`x=median(2, 4, 1, 3);`	`2.5`
`y=median(5, 8, 0, 3, 4);`	`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:

Statements	Results
`a=min(2,.,6);`	`2`
`a=min(2,-3,1,-1);`	`-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:

Statement	Result
`a=minute(time());`	`19`

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:

Statements	Results
`a=mod(10,3);`	`1`
`a=mod(.35,-.1);`	`0.05`
`a=mod(17,3);`	`2`
`a=mod(.3,-.9);`	`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:

Statements	Results
`a=mod(10,3);` `b=modz(10,3);`	`1` `1`
`a=mod(.35,-.1);` `b=modz(.35,-.1);`	`0.05` `0.4999999999999`
`a=mod(17,3);` `b=modz(17,3)`	`2` `2`
`a=mod(.3,-.9);` `b=modz(.3,-.9);`	`0.3` `0.3`

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

The following statement illustrates the MONTH function when the month is November:

Statements	Results
`a=month(date());`	`11`

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:

Statements	Results
`a=n(1,0,.,2,5,.);`	4
`a=n(1,2);`	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:

Statements	Results
`a=nmiss(1,0,.,2,5,.);`	2
`a=nmiss(1,0);`	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.

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.

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.

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:

Statement	Result
`a=ordinal(4,1,.,2,3,-4,5,6,7);`	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, $0 \leq percentage \leq 100$.

percentage is numeric where, $0 \leq percentage \leq 100$.

expression

specifies any valid expression that evaluates to a numeric value, whose value is computed in the percentile calculation.

Data type DOUBLE

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:

Statements	Results
`lower_quartile=PCTL(25,2,4,1,3);`	1.5
`percentile_def2=PCTL2(25,2,4,1,3);`	1
`lower_tertile=PCTL(100/3,2,4,1,3);`	2
`percentile_def3=PCTL3(100/3,2,4,1,3);`	2
`median=PCTL(50,2,4,1,3);`	2.5
`upper_tertile=PCTL(200/3,2,4,1,3);`	3
`upper_quartile=PCTL(75,2,4,1,3);`	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:

Statements	Results
`x=perm(5, 1);`	`5`
`x=perm(5);`	`120`
`x=perm(5, 2)`	`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:

  ```
  Payment1 = PMT(0.08/12., 10, 10000, 0, 0);
  ```

  ```
  Payment1 = PMT(0.08/12., 10, 10000);
  ```

  These computations return a value of 1037.03208935915.

- If the same loan has beginning-of-period payments, then payment can be computed as follows:

  ```
  Payment2 = PMT(0.08/12., 10, 10000, 0, 1);
  ```

  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:

  ```
  Payment3 = PMT(.01/12., 5, 5000);
  ```

  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:

  ```
  Payment4 = PMT(0.06/12., 216, 0, 50000, 0);
  ```

  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:

Statements	Results
`x=poisson(1, 2);`	`0.9196986029286`

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:

  ```
  PrincipalPayment = PPMT(0.1/12., 1, 24, 2000);
  ```

  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:

  ```
  PrincipalPayment2 = PPMT(0.1/12., 1, 36, 20000, 0, 1);
  ```

  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:

  ```
  PrincipalPayment3 = PPMT(0.1/12., 1, 36, 20000, 5000, 0);
  ```

  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 $0 \le x \le 1$

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 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:

Statements	Results
`x=probbeta(.2,3,4);`	`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 $0 \le p \le 1$

Data type DOUBLE

n

is the number of independent Bernoulli trials parameter. This argument must be a whole number.

Range $n > 0$

Data type DOUBLE

m

is the number of successes random variable. This argument must be a whole number.

Range $0 \le m \le n$

Data type DOUBLE

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success p, number of trials n, and number of successes m, is less than or equal to m. To compute the probability that an observation is equal to a given value m, compute the difference of two probabilities from the binomial distribution for m and $m-1$ successes.

Example

The following statement illustrates the PROBBNML function:

Statements	Results
`x=probbnml(0.5,10,4);`	`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:

Statements	Results
`x=probchi(11.264,11);`	`0.5785813293173`

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	$x \geq 0$
Data type	DOUBLE

ndf

is a numeric numerator degrees of freedom parameter.

Range	$ndf > 0$
Data type	DOUBLE

ddf

is a numeric denominator degrees of freedom parameter.

Range	$ddf > 0$
Data type	DOUBLE

nc

is an optional numeric noncentrality parameter.

Range	$nc \geq 0$
Data type	DOUBLE

Details

The PROBF function returns the probability that an observation from an *F* distribution, with numerator degrees of freedom *ndf*, denominator degrees of freedom *ddf*, and noncentrality parameter *nc*, is less than or equal to *x*. The PROBF function accepts noninteger degrees of freedom parameters *ndf* and *ddf*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central *F* distribution.

The significance level for an *F* test statistic is given by the following equation.

```
p=1-probf(x,ndf,ddf);
```

Example

The following statement illustrates the PROBF function:

Statements	Results
`x=probf(3.32,2,3);`	`0.82639336022431`

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:

Statements	Results
`x=probgam(1,3);`	`0.08030139707139`

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:

Statements	Results
`x=probhypr(200,50,10,2);`	`0.52367340812167`

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 pth quantile from the standard normal distribution. The probability that an observation from the standard normal distribution is less than or equal to the returned quantile is *p*.

CAUTION:

The result could be truncated to lie between -8.222 and 7.941.

Note: PROBIT is the inverse of the PROBNORM function.

Example

The following statements illustrate the PROBIT function:

Statements	Results
`x=probit(.025);`	`-1.95996398454005`
`x=probit(1.e-7);`	`-5.19933758219281`

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 ith sample being of size n_i. Let $I = \sqrt{-1}$. The distribution function [1, 2, 3, 4, 5] is the CDF for the maximum absolute of a k-dimensional multivariate $\mathbb{T}$ vector, with ν degrees of freedom, and an associated correlation matrix $\rho_{ij} = -\alpha_i\alpha_j$. This equation can be written as 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" on page 372.

Many-One t-Statistics: Dunnett's One-Sided Test

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are λ_1, ..., λ_k, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. 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" on page 375.

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 $\sigma_1, \ldots, \sigma_k$, 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 $\sigma_1, ..., \sigma_k$, 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 $\sigma_1, ..., \sigma_k$, 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 $\frac{1}{2}\nu S^2$ is a χ^2 variable with ν 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.

$$\begin{aligned}\Pr(Y_k > d) &= \Pr(U_1 > d)\\ &+\Pr(U_2 > d, U_1 < d)\\ &+\Pr(U_3 > d, U_2 < d, U_1 < d)\\ &+\ldots\\ &+\Pr(U_k > d, U_{k-1} < d, \ldots, U_1 < d)\\ &= \Pr(Y_{k-1} > d) + \Pr(X_k + (k-1)U_{k-1} > kd)\end{aligned}$$

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 $\nu = \infty$. Due to the complexity of the operations, this lengthy algorithm is replaced by a much faster one when $k \leq 15$ for both finite and infinite degrees of freedom ν. For $k \geq 16$, 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 $k = 6$, $\nu = 42$, and $t = 2.60$ is .9924467341, which shows strong evidence that there is a response to the substance. You can also compute the quantiles for the upper 5% and 1% tails, as shown in the following table.

```
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;
```

Statements	Results
`prob=probmc('williams',2.6,.,42,6);`	`0.99244668715827`

Statements	Results
`quant5=probmc('williams',.,.95,42,6);`	`1.80656253603889`
`quant1=probmc('williams',.,.99,42,6);`	`2.49090827298686`

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 $0 \le p \le 1$

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:

Statements	Results
`x=probnegb(0.5,2,1);`	`0.5`

PRXCHANGE Function

Performs a pattern-matching replacement.

Returned data type: CHAR

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 na
  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 5.4 *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
```

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;
```

```
position=7
```

PRXPOSN Function

Returns a character string that contains the value for a capture buffer.

Returned data type: CHAR

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.

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;
```

```
*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;
```

Log 5.1 *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 $0 \le c < 1$

$0 \le c < 1$

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.

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:.

Statements	Results

`a=17180;`
`b=print(a,date7.);`
`c=qtr(a);`
	`14JAN07`
	`1`

QUOTE Function

Adds double quotation marks to a character value.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

QUOTE(*expression*)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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:

Statements	Results
`a='A"B';` `b=quote(a);`	`"A""B"`
`a='A''B';` `b=quote(a);`	`"A'B"`
`a='Paul''s Catering Service  ';` `b=quote(trim(a));`	`"Paul's Catering Service"`

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:

Statements	Results
`a=range(.,.);`	.
`a=range(-2,6,3);`	8
`a=range(2,6,3,.);`	4
`a=range(1,6,3,1);`	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:

Statement	Result	
	ASCII	EBCDIC
`a=rank('A');`	65	193

REPEAT Function

Repeats a character expression.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

REPEAT(*expression*, *n*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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:

Statements	Results
`a=repeat('ONE',2);`	`ONEONEONE`

REVERSE Function

Reverses a character expression.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

REVERSE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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:

Statements	Results
	`----+----1`
`a=reverse('xyz  ');`	`zyx`

RIGHT Function

Right aligns a character expression.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

RIGHT(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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 statements illustrate the RIGHT function:

Statements	Results
	`----+----1----+`

Statements	Results

`a='Due Date  ';`
`b=print(a, $10.);`
`c=print(right(a),$10.);`
	`Due Date`
	`  Due Date`

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:

Statements	Results
`x1=rms(1,7);`	`5`
`x2=rms(.,1,5,11);`	`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.

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*>)

[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.

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.

Examples

Example 1: Comparing Results from the ROUNDZ and ROUND Functions

The following example compares the results that are returned by the ROUNDZ and the ROUND function.

```
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
```

Example 2: Sample Output from the ROUNDZ Function

The following statements illustrate the ROUNDZ function:

Statement	Results

`a=223.456;`
`b=roundz(a,1);`
`c=(put(b), 9.5);`
	`223.00000`

`var2=223.456;`
`x=roundz(var2,.01);`
`print x 9.5;`
	`223.46000`

`x=roundz(223.456,100);` `print x 9.5;`	`200.00000`
`x=roundz(223.456,.3);` `print x 9.5;`	`223.50000`

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 accumulate at least $12,000, can be expressed as follows:

```
number=saving(12000, 100, .05/12, .);
```

The value that is returned is 97.18 months. The fourth argument is set to missing, which indicates that the number of payments is to be calculated. The 5% nominal annual rate is converted to a monthly rate of 0.05/12. The rate is the fractional (not the percentage) interest rate per compounding period.

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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

SCAN(*expression*, *n* <, *delimiters*<, *modifier*>>)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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 *charlist* and *modifier* arguments do not change. The O modifier applies separately to each instance of the SCAN function in your SAS code, and does *not* cause all instances of the SCAN function to use the same delimiters and modifiers.

p or P adds punctuation marks to the list of characters.

q or Q ignores delimiters that are inside substrings that are enclosed in quotation marks. If the value of the *string* argument contains unmatched quotation marks, then scanning from left to right produces different words than scanning from right to left.

r or R removes leading and trailing blanks from the word that SCAN returns. If you specify the Q and R modifiers, the SCAN function first removes leading and trailing blanks from the word. Then, if the word begins with a quotation mark, SCAN also removes one layer of quotation marks from the word.

s or S adds space characters to the list of characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed).

t or T trims trailing blanks from the *string* and *charlist* arguments. If you want to remove trailing blanks from only one character argument instead of both character arguments, use the TRIM function instead of the SCAN function with the T modifier.

u or U adds uppercase letters to the list of characters.

w or W adds printable (writable) characters to the list of characters.

x or X adds hexadecimal characters to the list of characters.

Restriction This argument is supported only in SAS Viya and on the CAS server.

Tip If the *modifier* argument is a character constant, enclose the argument in quotation marks. Specify multiple modifiers in a single set of quotation marks. A *modifier* argument can also be expressed as a character variable or expression.

Details

Definitions of "Delimiter" and "Word"

A *delimiter* is any of several characters that are used to separate words. You can specify the delimiters in the *delimiter* and *modifier* arguments.

If you specify the Q modifier, delimiters inside substrings that are enclosed in quotation marks are ignored.

In the SCAN function, “word” refers to a substring that has all of these characteristics:

- It is bounded on the left by a delimiter or the beginning of the string.
- It is bounded on the right by a delimiter or the end of the string.
- It contains no delimiters.

A word can have a length of zero if there are delimiters at the beginning or end of the string, or if the string contains two or more consecutive delimiters. However, the SCAN function ignores words that have a length of zero unless you specify the M modifier.

Note: The definition of “word” is the same in the SCAN and COUNTW functions.

Using Default Delimiters in ASCII and EBCDIC Environments

If you use the SCAN function with only two arguments, then the default delimiters depend on whether your computer uses ASCII or EBCDIC characters.

- If your computer uses ASCII characters, the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ^ |

  In ASCII environments that do not contain the ^ character, the SCAN function uses the ~ character instead.

- If your computer uses EBCDIC characters, then the default delimiters are as follows:

  blank ! $ % & ( ) * + , - . / ; < ¬ | ¢

If you use the *modifier* argument without specifying any characters as delimiters, then the only delimiters that are used are delimiters that are defined by the *modifier* argument. In this case, the lists of default delimiters for ASCII and EBCDIC environments are not used. In other words, modifiers add to the list of delimiters that are explicitly specified by the *delimiter* argument. Modifiers do not add to the list of default modifiers.

The Length of the Result

Leading delimiters before the first word in the expression do not affect SCAN. If there are two or more contiguous delimiters, SCAN treats them as one.

In DS2, if the SCAN function returns a value to a variable that has not yet been given a length, and then that variable is given the length of the first argument. If you need the SCAN function to assign to a variable a value that is different from the length of the first argument, use a DECLARE statement for that variable before the statement that uses the SCAN function.

The minimum length of the word that is returned by the SCAN function depends on whether the M modifier is specified.

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:

Statements	Results
`expr='ABC.DEF(X=y)';` `word=scan(expr,3);`	`X=Y`
`scan('ABC.DEF(X=Y)',-3);`	`ABC`

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:

Statement	Result
`x=sec(0.5);`	`1.13949392732454`
`y=sec(0);`	`1`
`z=sec(3.14159/3);`	`1.99999693590391`

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:

Statements	Results
`a=hms(3,19,24);` `b=second(a);`	`11964` `24`
`a=hms(6,25,65);` `s=second(a);`	`23165` `5`
`a=hms(3,19,60);` `b=second(a);`	`12000` `0`

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:

Statements	Results
`a=sign(-5);`	`-1`
`a=sign(5);`	`1`
`a=sign(0);`	`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:

Statements	Results
`a=sin(25.6)`	`0.45044059427538`
`a=sin(5);`	`-0.95892427466313`

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:

Statements	Results
`a=sinh(0);`	`0`
`a=sinh(1);`	`1.1752011936438`
`a=sinh(-1.0);`	`-1.1752011936438`

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:

Statements	Results
`x1=skewness(0,1,1);`	`-1.73205080756887`
`x2=skewness(2,4,6,3,1);`	`0.59012865638436`
`x3=skewness(2,0,0);`	`1.73205080756887`

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.

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:

Statements	Results
`a=sqrt(36);`	`6`
`a=sqrt(25);`	`5`
`a=sqrt(4.4);`	`2.0976176963403`

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:

Statements	Results
`a=std(2,6);`	`2.82842712474619`
`a=std(2,6,.);`	`2.82842714274619`
`a=std(2,4,6,3,1);`	`1.92353840616713`

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.

Statements	Results
`a=stderr(2,6);`	2
`a=stderr(2,6,.);`	2
`a=stderr(2,4,6,3,1);`	0.86023252670426

STRIP Function

Returns a character string with all leading and trailing blanks removed.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

STRIP(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

The STRIP function returns the argument with all leading and trailing blanks removed. If the argument is blank, STRIP returns a string with a length of zero.

If the value that is trimmed is shorter than the length of the receiving variable, SAS pads the value with new trailing blanks.

Note: The STRIP function is useful for concatenation because the concatenation operator does not remove trailing blanks.

Comparisons

The following list compares the STRIP function with the TRIM function:

- For blank character strings, the STRIP and TRIM functions both return a string with a length of zero.
- For strings that lack leading blanks, the STRIP and TRIM functions return the same value.

Example

The following example shows the results of using the STRIP function to delete leading and trailing blanks.

```
proc cas;
  string1="abcd";
  string2=" abcd ";
  string3=" abcd";
  string4="abcdefgh";
  string5="x y z";
  original1='*' || string1 || '*';
  original2='*' || string2 || ' *';
  original3=' *' || string3 || '*';
  original4='*' || string4 || '*';
  original5='*' || string5 || '*';
  stripped1='*' || strip (string1) || '*';
  stripped2='*' || strip (string4) || '*';
  stripped3='*' || strip (string5) || '*';

  columns={'string', 'original', 'stripped'};
  coltypes={'string', 'string', 'string'};
  row1={string1, original1, stripped1};
  row2={string2, original2, stripped1};
  row3={string3, original3, stripped1};
  row4={string4, original4, stripped3};
  row5={string5, original5, stripped3};
  striptab=newtable('striptab', columns, coltypes, row1, row2, row3, row4, row5);
  print striptab;
run;
```

Output 5.5 *Results from the STRIP Function*

striptab: Results

string	original	stripped
abcd	*abcd*	*abcd*
abcd	* abcd *	*abcd*
abcd	* abcd*	*abcd*
abcdefgh	*abcdefgh*	*x y z*
x y z	*x y z*	*x y z*

SUBSTR (right of =) Function

Returns a substring, allowing a result with a length of zero.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

SUBSTR(*character-expression*, *position-expression* <, *length-expression*>)

Arguments

character-expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

position-expression
specifies any valid expression that evaluates to a BIGINT and that specifies the character position of the first character in the substring.

Requirement *position-expression* must be greater than or equal to zero.

Data type BIGINT

length-expression
specifies any valid expression that evaluates to a BIGINT and that specifies the character length of the substring. If you do not specify *length-expression*, the SUBSTR (right of) function returns the substring that extends from the character position that you specify to the end of the string.

Data type BIGINT

Details

The following information applies to the SUBSTR function:

- The SUBSTR (right of) function returns a missing (SAS mode) or null value (ANSI mode) if any of the following are true:
  - *position-expression* is less than 1 or greater than the length of *character-expression*
  - *character-expression* is a missing (SAS mode) or null value (ANSI mode)
  - *position-expression* is a missing (SAS mode) or null value (ANSI mode)

  Note: With the last condition, a warning is written to the log stating that the second argument is invalid.
- The SUBSTR (right of =) function returns the substring from *position-expression* to the end of the *character-expression* if *length-expression* meets any of the following conditions:
  - is not specified
  - is less than zero
  - is greater than the length of the substring from *position-expression* to the end of the *character-expression*
  - is a missing or null value

  Note: With the last condition, a warning is written to the log stating that the third argument is invalid.

Example

The following statements illustrate the SUBSTR function:

Statements	Results
`a='chsh234960b3';` `substr(a,5);`	`234960b3`
`a='chsh234960b3';` `substr(a,5,6);`	`234960`
`a='chsh234960b3';` `substr(a,5,15);`	`234960b3`
`a='chsh234960b3';` `substr(a,5,-5);`	`234960b3`

SUBSTR (left of =) Function

Replaces as substring of content of a character variable.

Syntax

SUBSTR (*character-variable*, *position-expression* <, *length-expression*>) = *characters-to-replace*

Arguments

character-variable
specifies character variable.

Restriction	*character-variable* must be a variable or an array reference. It cannot be an expression or a literal.
Data type	CHAR, NCHAR, NVARCHAR, VARCHAR

position-expression
specifies any valid expression that evaluates or can be coerced to an integer and that specifies the character position of the first character in the substring in the *character-variable* that is to be replaced.

Requirement	*position-expression* must be greater than or equal to zero.
Data type	BIGINT. Other types are coerced to BIGINIT.

length-expression
specifies any valid expression that evaluates or can be coerced to an integer and that specifies the character length of the substring in *character-variable* that is to be replaced. If you do not specify *length-expression*, *length-expression* defaults to the character length of the substring in *character-variable* that extends from the character position that you specify to the end of the string.

Restriction *length-expression* cannot be larger than the character length of the substring in *character-variable* that extends from *position-expression* to the maximum length of *character-variable*.

Data type BIGINT. Other types are coerced to BIGINIT.

Note *length-expression* can be larger than the character length of the substring in *character-variable* that is to be replaced.

characters-to-replace

specifies any valid expression that evaluates or can be coerced to a character string and represents the string that replaces a substring in *character-variable*.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

Details

When you use the SUBSTR function on the left side of an assignment statement, a substring of the content in the character variable is replaced with the character string value on the right side.

If *length-expression* is less than or equal to the character length of the substring that extends from *position-expression* to the end of the string in *character-variable*, the substring starting at *position-expression* and extending *length-expression* characters is replaced. If *length-expression* is greater than the character length of the substring that extends from *position-expression* to the end of the string in *character-variable*, the substring starting at *position-expression* and extending to the end of the string is replaced.

If the character length of characters-to-replace is longer than *length-expression*, characters-to-replace is truncated to *length-expression* characters before being used as the substring replacement in *character-variable*. If the character length of characters-to-replace is shorter than *length-expression*, characters-to-replace is blank padded to *length-expression* characters before being used as the substring replacement in *character-variable*.

If the character length of the resulting character string after the substring replacement is longer than the maximum character length of *character-variable*, the resulting character string is truncated to the maximum character length of *character-variable*.

The SUBSTR (left of =) function does not modify *character-variable* if any of the following are true:

- *character-variable* is a null value (ANSI mode)

  length-expression is less than 1

  length-expression is larger than the character length of the substring in *character-variable* that extends from *position-expression* to the maximum length of *character-variable*

  Note: In addition, a warning is written to the log stating that the argument is invalid.

Example

The following statements illustrate the SUBSTR function:

Statements	Results

`mystring='kidnap';`
`substr(mystring, 1, 3)='cat';`
`print mystring;`
	`catnap`

`mystring=' ';`
`substr(mystring, 1)='cat';`
`print mystring;`
	`cat`

`mystring='kidnap';`
`substr(mystring,.)='cat';`
`print mystring;`
	`kidnap`

SUBSTRN Function

Returns a substring, allowing a result with a length of zero.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

SUBSTRN(*expression*, *position-expression*<, *length-expression*>)

Arguments

expression

specifies any valid expression that evaluates or can be coerced to a character string or numeric value.

specifies any valid expression that evaluates or can be coerced to a character string or numeric value.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR, DOUBLE

position-expression

specifies any valid expression that evaluates to a BIGINT and that specifies the position of the first character in the substring.

Data type BIGINT

length-expression

specifies any valid expression that evaluates to a BIGINT and that specifies the length of the substring.

Data type BIGINT

Details

The Basics

The following information applies to the SUBSTRN function:

- The SUBSTRN function returns the substring from *position-expression* to the end of the *character-expression* if *length-expression* meets either of the following conditions:
  - is not specified
  - is greater than the length of the substring from *position-expression* to the end of the *expression*
- The SUBSTRN function returns a missing (SAS mode) or null value (ANSI mode) with a length of zero if any of the following conditions are true:
  - *position-expression* is a missing (SAS mode) or null value (ANSI mode)
  - *length-expression* is a missing (SAS mode) or null value (ANSI mode)

  Note: In addition, an error is written to the log stating that the argument is invalid.
- The SUBSTRN function returns a missing value (SAS mode) or an empty string (ANSI mode) with a length of zero if any of the following conditions are true:
  - *expression* is a missing (SAS mode) or null value (ANSI mode)
  - *length-expression* is less than 1
  - *position-expression* is less than 1 or greater than the length of *character-expression*

Comparisons

The following table lists comparisons between the SUBSTRN and the SUBSTR functions:

Condition	Function	Result
the value of *position-expression* is nonpositive	SUBSTRN	returns a missing value (SAS mode) or empty string (ANSI mode) of length 0
the value of *position-expression* is nonpositive	SUBSTR	returns a missing (SAS mode) or null value (ANSI mode) of length 0
the value of *length-expression* is nonpositive	SUBSTRN	returns a missing value (SAS mode) or empty string (ANSI mode) of length 0
the value of *length-expression* is nonpositive	SUBSTR	returns the substring from *position-expression* to the end of the *character-expression*
the value of the *length-expression* is a missing (SAS mode) or null value (ANSI mode)	SUBSTRN	returns a missing or null value with a length of zero and writes an error to the log that the third argument is invalid
the value of the *length-expression* is a missing (SAS mode) or null value (ANSI mode)	SUBSTR	returns substring from *position-expression* to the end of the *expression* and writes a warning to the log that the third argument is invalid

Example: 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.

```
proc cas;
      substr_result='*' || substr('   1234.5678',2,6) || '*';
      print "substr_result=" substr_result;
      substrn_result='*' || substrn('1234.5678',2,6) || '*';
      print "substrn_result=" substrn_result;
run;
```

SAS writes the following output to the log:

```
substr_result=*  1234*
substrn_result=*234.56*
```

SUM Function

Returns the sum of the non-null or nonmissing arguments.

Returned data type: BIGINT, DECIMAL, DOUBLE

Syntax

SUM(*expression-1*, *expression-2* <, …*expression-n*>)

Arguments

expression
specifies any valid expression that evaluates to a numeric value.

Requirement	At least two arguments are required.
Data type	BIGINT, DECIMAL, DOUBLE

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.

Statements	Results
`a=sum(4,9,3,8);`	24
`a=sum(4,9,3,8,.);`	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:

Statements	Results
`a=tan(0.5);`	`0.54630248984379`
`a=tan(0);`	`0`
`a=tan(3.14159/3);`	`1.73204726945457`

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 $\pi/2$

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.

Statements	Results
`a=tanh(0);`	`0`
`a=tanh(0.5);`	`0.46211715726`
`a=tanh(-0.5);`	`-0.46211715726`

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:

Statements	Results
`a=time();`	`56526.0399990081`
`a=put(time(),time.);`	`15:42:06`

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:

Statements	Results

`proc cas;`
`  ddtm=datetime();`
`  tm=put(timepart(ddtm),time.);`
`  print "ddtm=" ddtm;`
`  print "tm=" tm;`
`run;`
	`ddtm=1841668215.1`
	`tm=14:30:15`

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 pth quantile from the Student's *t* distribution with degrees of freedom *df* and a noncentrality parameter *nc*. The probability that an observation from a *t* distribution is less than or equal to the returned quantile is *p*.

TINV accepts a noninteger degree of freedom parameter *df*. If the optional parameter *nc* is not specified or is 0, the quantile from the central *t* distribution is returned.

CAUTION:

For large values of nc, the algorithm can fail. In that case, a missing value is returned.

Comparisons

TINV is the inverse of the PROBT function.

Example

The following statements illustrate the TINV function:

Statement	Result
`x=tinv(.95, 2);`	`2.91998558035372`
`x=tinv(.95, 2.5, 3);`	`11.0338336251942`

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:

Statements	Results

`proc cas;`
`  td=today();`
`  tday=put(td, date.);`
`  print "td=" td;`
`  print "tday=" tday;`
`run;`
	`td=21315`
	`tday=11MAY18`

TRANSLATE Function

Replaces specific characters in a character expression.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRANSLATE(*expression*, *to-characters*, *from-characters*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string. *expression* contains the original character value.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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.

Statements	Results

`proc cas;`
`  a=translate('XYZW', 'AB', 'VW');`
`  print "a=" a;`
`run;`
	`a=XYZB`

`proc cas;`
`  string1='AABBAABABB';`
`  a=translate(string1, '12', 'AB');`
`  print "a=" a;`
`run;`
	`a=1122112122`

TRANSTRN Function

Replaces or removes all occurrences of a substring in a character string.

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRANSTRN(*source-expression*, *target-expression*, *replacement-expression*)

Arguments

source-expression
specifies any valid expression that evaluates or can be coerced to a character string and whose characters you want to translate.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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:

Statements	Results
`a=trigamma(3);`	`0.39493406684822`

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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

TRIM('*expression*')

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR, NVARCHAR, VARCHAR

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:

Statements	Results

`proc cas;`
`  string1='Testscore  ';`
`  string2='File.xls';`
`  result=trim(string1)||(string2);`
`  print "result=" result;`
`run;`
	`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

```
data new;
   x="A"||trimn("")||"B";
   z=" ";
   y=">"||trimn(z)||"<";
   put x= 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:

Statements	Results
`x=trunc(3.1,3);`	`3.099609375`
`x=trunc(3.1,4);`	`3.09999847412109`
`x=trunc(3.1,5);`	`3.09999999403953`
`x=trunc(3.1,6);`	`3.09999999997671`
`x=trunc(3.1,7);`	`3.0999999999999`
`x=trunc(3.1,8);`	`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: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

UPCASE(*expression*)

Arguments

expression
specifies any valid expression that evaluates or can be coerced to a character string.

Data type CHAR, NCHAR

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:

Statements	Results
`name=upcase('John B. Smith');`	`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:

Statements	Results
`a=uss(4,2,3.5,6);`	`68.25`
`a=uss(4,2,3.5,6,.);`	`68.25`

UUIDGEN Function

Returns the short form of a Universally Unique Identifier (UUID).

Returned data type: CHAR, NCHAR, NVARCHAR, VARCHAR

Syntax

UUIDGEN( )

Without Arguments

The UUIDGEN function has no arguments.

Details

The UUIDGEN function returns a UUID (a unique value) for each call. The default result is 36 characters long and it looks like this:

```
5ab6fa40-426b-4375-bb22-2d0291f43319
```

Example

The following 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:

Statements	Results

`proc cas;`
`  x='abc';`
`  y='abcdef';`
`  z=verify(y,x);`
`  print "z=" z;`
`run;`
	`z=4`

`proc cas;`
`  x='abcdef';`
`  y='abcdef';`
`  z=verify(y,x);`
`  print "z=" z;`
`run;`
	`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:

Statements	Results
`x=weekday(today());`	1

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:

Statements	Results
`x=whichc('Spain', 'Denmark', 'Germany', 'Austria', 'Spain', 'China', 'Egypt', 'Spain', 'France');`	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.

Statements	Results
`x=whichn(4.5,7.3, 8.6, 4.5, 4.5, 2.1, 6.4);`	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

The following statement illustrates the YEAR function when the year is 2007.

Statements	Results
`date=today();` `y=year(date);`	`2007`

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_0

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:

Statements	Results

`proc cas;`
`  DateValue=yyq(2016,3);`
`  Date7Value=put(DateValue, date7.);`
`  Date9Value=put(DateValue, date9.);`
`  print "DateValue=" DateValue;`
`  print "Date7Value=" Date7Value;`
`  print "Date9Value=" Date9Value;`
`run;`
	`DateValue=20636`
	`Date7Value=01JUL16`
	`Date9Value=01JUL2016`

Statements	Results

`proc cas;`
`  StartOfQuarter=yyq(2016,4);`
`  StartofQuarter9=put(StartOfQuarter, date9.);`
`  print "StartofQuarter=" StartofQuarter;`
`  print "StartofQuarter9=" StartofQuarter9;`
`run;`
	`StartofQuarter=20728`
	`StartofQuarter9=01OCT2016`

Recommended Reading

- *Getting Started with CASL Programming*
- *CASL Programmer's Guide*
- *SAS Cloud Analytic Services: User's Guide*
- *SAS Viya: System Programming Guide*
- *SAS Visual Analytics: Programming Guide*
- SAS Visual Statistics: Programming Guide
- SAS Visual Data Mining and Machine Learning: Programming Guide

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books