SAS® 9.3 Functions and CALL Routines

Reference

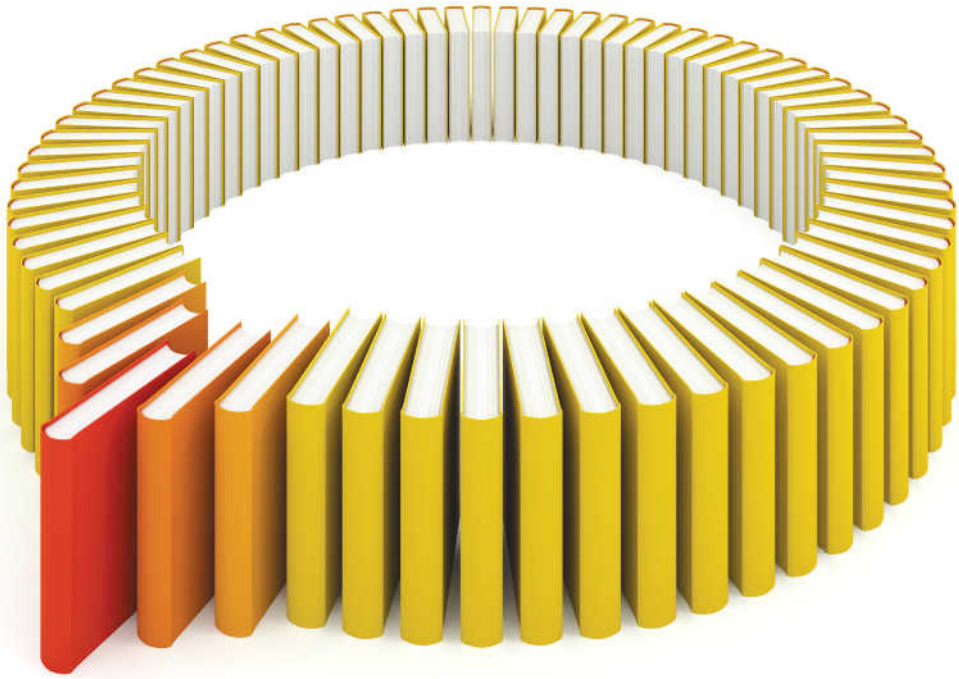
SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2011. *SAS® 9.3 Functions and CALL Routines: Reference*. Cary, NC: SAS Institute Inc.

SAS® 9.3 Functions and CALL Routines: Reference

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

Printing 2, August 2012



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.

Contents

About This Book

Syntax Conventions for the SAS Language

Overview of Syntax Conventions for the SAS Language

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

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

Syntax Components

The components of the syntax for most language elements include a keyword and arguments. For some language elements, only a keyword is necessary. For other language elements, the keyword is followed by an equal sign (=).

keyword
specifies the name of the SAS language element that you use when you write your program. Keyword is a literal that is usually the first word in the syntax. In a CALL routine, the first two words are keywords.

In the following examples of SAS syntax, the keywords are the first words in the syntax:

CHAR (*string, position*)

CALL RANBIN (*seed, n, p, x*);

ALTER (*alter-password*)

BEST *w*.

REMOVE <*data-set-name*>

In the following example, the first two words of the CALL routine are the keywords:

CALL RANBIN(*seed*, *n*, *p*, *x*)

The syntax of some SAS statements consists of a single keyword without arguments:

DO;
... SAS code ...

END;

Some system options require that one of two keyword values be specified:

DUPLEX | **NODUPLEX**

argument
specifies a numeric or character constant, variable, or expression. Arguments follow the keyword or an equal sign after the keyword. The arguments are used by SAS to process the language element. Arguments can be required or optional. In the syntax, optional arguments are enclosed between angle brackets.

In the following example, *string* and *position* follow the keyword CHAR. These arguments are required arguments for the CHAR function:

CHAR (*string, position*)

Each argument has a value. In the following example of SAS code, the argument *string* has a value of 'summer', and the argument *position* has a value of 4:`x=char('summer', 4);`

In the following example, *string* and *substring* are required arguments, while *modifiers* and *startpos* are optional.

FIND(*string, substring* <,*modifiers*> <,*startpos*>

Note: In most cases, example code in SAS documentation is written in lowercase with a monospace font. You can use uppercase, lowercase, or mixed case in the code that you write.

Style Conventions

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

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

ERROR<*message*>;

UPPERCASE
identifies arguments that are literals.

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

CMPMODEL = BOTH | CATALOG | XML

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

- nonliteral arguments In the following example of the LINK statement, the argument *label* is a user-supplied value and is therefore written in italics:

  LINK *label*;

- nonliteral values that are assigned to an argument

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

  FORMAT = *variable-1* <, ..., *variable-n format* ><DEFAULT = *default-format*>;

Items in italics can also be the generic name for a list of arguments from which you can choose (for example, *attribute-list*). If more than one of an item in italics can be used, the items are expressed as *item-1, ..., item-n*.

Special Characters

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

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

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

  MAPS = *location-of-maps*

< >
: angle brackets identify optional arguments. Any argument that is not enclosed in angle brackets is required.

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

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

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

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

  CMPMODEL = BOTH | CATALOG | XML

...
: an ellipsis indicates that the argument or group of arguments following the ellipsis can be repeated. If the ellipsis and the following argument are enclosed in angle brackets, then the argument is optional.

  In the following example of the CAT function, the ellipsis indicates that you can have multiple optional items:

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

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

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

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

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

  In the following example each statement ends with a semicolon: `data namegame; length color name $8; color = 'black'; name = 'jack'; game = trim(color) || name; run;`

References to SAS Libraries and External Files

Many SAS statements and other language elements refer to SAS libraries and external files. You can choose whether to make the reference through a logical name (a libref or fileref) or use the physical filename enclosed in quotation marks. If you use a logical name, you usually have a choice of using a SAS statement (LIBNAME or FILENAME) or the operating environment's control language to make the association. Several methods of referring to SAS libraries and external files are available, and some of these methods depend on your operating environment.

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

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

What's New in SAS 9.3 Functions and CALL Routines

Overview

The SAS functions and CALL routines are now published as a separate document. They are no longer part of the *SAS Language Reference: Dictionary*. For more information, see "Changes to SAS Language Reference: Dictionary" on page xiii.

The ability to call Web services in the DATA step is a new feature. For this feature, six new SOAPxxx functions were added. In addition, several other new functions are new, and enhancements to existing functions were added.

New Functions and CALL Routines

The following functions and CALL routines are new:

CALL RANCOMB (p. 215)
: permutes the values of the arguments, and returns a random combination of *k* out of *n* values.

EFFRATE (p. 391)
: returns the effective annual interest rate.

MVALID (p. 676)
: checks the validity of a character string for use as a SAS member name.

NOMRATE (p. 684)
: returns the nominal annual interest rate.

SAVINGS (p. 846)
: returns the balance of periodic savings by using variable interest rates.

SOAPWEB (p. 865)
: calls a Web service by using basic Web authentication; credentials are provided in the arguments.

SOAPWEBMETA (p. 867)
: calls a Web service by using basic Web authentication; credentials for the authentication domain are retrieved from metadata.

SOAPWIPSERVICE (p. 869)
: calls a SAS registered service by using WS-Security authentication; credentials are provided in the arguments.

SOAPWIPSRS (p. 871)
calls a SAS registered Web service by using WS-Security authentication; credentials are provided in the arguments. The Registry Service is called directly to determine how to locate the Security Token Service.

SOAPWS (p. 873)
calls a Web service by using WS-Security authentication; credentials are provided in the arguments.

SOAPWSMETA (p. 875)
calls a Web service by using WS-Security authentication; credentials for the provided authentication domain are retrieved from metadata.

SQUANTILE (p. 881)
returns the quantile from a distribution when you specify the right probability (SDF).

SYSEXIST (p. 903)
returns an indication of the existence of an operating environment variable.

TIMEVALUE (p. 913)
returns the equivalent of a reference amount at a base date by using variable interest rates.

Enhancements to Existing Functions

The following enhancements were made to existing functions:

- The GENPOISSON and TWEEDIE distributions were added to the following functions:
  - CDF
  - PDF
  - SDF
  - LOGCDF
  - LOGPDF
  - LOGSDF
  - QUANTILE
- A new argument, *seasonality*, was added to the INTCYCLE, INTINDEX, and INTSEAS functions. The *seasonality* argument enables you to have more flexibility in working with dates and time cycles. For more information, see the "INTCYCLE Function" on page 565, "INTINDEX Function" on page 574, and "INTSEAS Function" on page 589.
- A new option that computes age was added to the YRDIF function. For more information, see the "YRDIF Function" on page 986.
- An explanation about SAS session encoding and UTF-8 encoding was added to the URLDECODE and URLENCODE functions. For more information, see the "URLDECODE Function" on page 929, and "URLENCODE Function" on page 930.
- In the GETOPTION function, you can use the following options:

- The DEFAULTVALUE option obtains the default shipped value for a system option. The value can be used to reset a system option to its default.
- The HEXVALUE option returns a system option value as a hexadecimal value.
- The LOGNUMBERFORMAT option returns a system option numeric value. The punctuation that is used is dependent on the language locale.
- The STARTUPVALUE option returns the system option value that was used to start SAS either on the command line or in a configuration file.

Documentation for Existing Functions and Concepts

The documentation for the following five functions was moved from the *SAS/ETS User's Guide* to *SAS Functions and CALL Routines: Reference*:

CUMIPMT (p. 347)
: returns the cumulative interest paid on a loan between the start and end period.

CUMPRINC (p. 348)
: returns the cumulative principal paid on a loan between the start and end period.

IPMT (p. 598)
: returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.

PMT (p. 745)
: returns the periodic payment for a constant payment loan or the periodic savings for a future balance.

PPMT (p. 748)
: returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.

In the second maintenance release for SAS 9.3, the following enhancements were made to the documentation:

- The FINANCE function now includes documentation for ISPMT, which calculates the interest that is paid during a specific period of investment.
- A section about using the DATA step with custom time intervals was added to the documentation. This section includes examples that use the INTNX and INTCK functions and the INTERVALDS system option.

Changes to *SAS Language Reference: Dictionary*

Prior to 9.3, this document was part of *SAS Language Reference: Dictionary*. Starting with 9.3, *SAS Language Reference: Dictionary* has been divided into seven documents:

- *SAS Data Set Options: Reference*
- *SAS Formats and Informats: Reference*
- *SAS Functions and CALL Routines: Reference*

- *SAS Statements: Reference*
- *SAS System Options: Reference*
- *SAS Component Objects: Reference* (contains the documentation for hash, hash iterator, and Java objects)
- *Base SAS Utilities: Reference* (contains the documentation for the SAS DATA step debugger and the SAS Utility macro %DS2CSV)

Recommended Reading

Here is the recommended reading list for this title:

- *Base SAS Glossary*
- *Base SAS Procedures Guide*
- *SAS Companion for UNIX Environments*
- *SAS Companion for Windows*
- *SAS Companion for z/OS*
- *SAS Data Set Options: Reference*
- *SAS Formats and Informats: Reference*
- *SAS Language Reference: Concepts*
- *SAS Metadata LIBNAME Engine: User's Guide*
- *SAS National Language Support (NLS): Reference Guide*
- *SAS Output Delivery System: User's Guide*
- *SAS Scalable Performance Data Engine: Reference*
- *SAS Statements: Reference*
- *SAS System Options: Reference*

The recommended reading list from SAS Press includes the following titles:

- *An Array of Challenges - Test Your SAS Skills*
- *Cody's Data Cleaning Techniques Using SAS*
- *Combining and Modifying SAS Data Sets: Examples*
- *Debugging SAS Programs: A Handbook of Tools and Techniques*
- *SAS Functions by Example*
- *SAS Guide to Report Writing: Examples*
- *Health Care Data and SAS*
- *The Little SAS Book: A Primer*
- *Output Delivery System: The Basics and Beyond*
- *SAS Programming by Example*
- *Quick Results with the Output Delivery System*

- *Step-by-Step Programming with Base SAS Software*
- *Using the SAS Windowing Environment: A Quick Tutorial*
- *The SAS Workbook*
- *SAS XML LIBNAME Engine: User's Guide*

For a complete list of SAS books, go to support.sas.com/bookstore. If you have questions about which titles you need, please contact a SAS Book Sales Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-3228
Fax: 1-919-677-8166
E-mail: sasbook@sas.com
Web address: support.sas.com/bookstore

Chapter 1

SAS Functions and CALL Routines

Definitions of Functions and CALL Routines

Definition of Functions

A SAS function is a component of the SAS programming language that can accept arguments, perform a computation or other operation, and return a value. Functions can return either numeric or character results. The value that is returned can be used in an assignment statement or elsewhere in expressions. Many functions are included with SAS, and you can write your own functions as well.

In Base SAS software, you can use SAS functions in DATA step programming statements, in a WHERE expression, in macro language statements, in PROC REPORT, and in Structured Query Language (SQL).

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

Definition of CALL Routines

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

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

Syntax

Syntax of Functions

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

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

function-name (OF *variable-list*)

function-name (<*argument* | OF *variable-list* | OF *array-name*[*]>
<…, <*argument* | OF *variable-list* | OF *array-name*[*]>>)

function-name
names the function.

argument
can be a variable name, constant, or any SAS expression, including another function. The number and type of arguments that SAS allows are described with individual functions. Multiple arguments are separated by a comma.

Note: If the value of an argument is invalid (for example, missing or outside the prescribed range), SAS writes a note to the log indicating that the argument is invalid, sets _ERROR_ to 1, and sets the result to a missing value. The following are examples:

- `x=max(cash,credit);`
- `x=sqrt(1500);`
- `NewCity=left(upcase(City));`
- `x=min(YearTemperature-July,YearTemperature-Dec);`
- `s=repeat('-+.16);`
- `x=min((enroll-drop),(enroll-fail));`
- `dollars=int(cash);`
- `if sum(cash,credit)>1000 then    put 'Goal reached';`

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

- `a=sum(of x  y  z);`
- `z=sum(of y1-y10);`
- `z=msplint(x0,5,of x1-x5,of y1-y5,-2,2);`

Example The following two examples are equivalent.

```
a=sum(of x1-x10 y1-y10 z1-z10);
a=sum(of x1-x10, of y1-y10, of z1-z10);
```

***array-name*{*}**
names a currently defined array. Specifying an array with an asterisk as a subscript causes SAS to treat each element of the array as a separate argument.

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

See For a list of these restrictions, see "Using the OF Operator with Temporary Arrays" on page 5.

Syntax of CALL Routines

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

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

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

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

routine-name
names a SAS CALL routine.

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

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

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

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

Using Functions and CALL Routines

Restrictions Affecting Function Arguments

If the value of an argument is invalid, SAS writes a note or error message to the log and sets the result to a missing value. Here are some common restrictions for function arguments:

- Some functions require that their arguments be restricted within a certain range. For example, the argument of the LOG function must be greater than 0.
- When a numeric argument has a missing value, many functions write a note to the SAS log and return a missing value. Exceptions include some of the descriptive statistics functions and financial functions.

- For some functions, the allowed range of the arguments is platform-dependent, such as with the EXP function.

Using the OF Operator with Temporary Arrays

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

There are some limitations in using temporary arrays. These limitations are listed after the example.

The following example shows how you can use temporary arrays:

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

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

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

```
x=55
x=10
```

The following limitations affect temporary array OF lists:

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

Characteristics of Target Variables

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

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

Functions	
COMPBL	RIGHT
COMPRESS	STRIP
DEQUOTE	SUBSTR
INPUTC	SUBSTRN
LEFT	TRANSLATE
LOWCASE	TRIM
PUTC	TRIMN
REVERSE	UPCASE

The following list of functions shows the length of the target variable if the target variable has not been assigned a length:

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

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

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

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

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

Notes about Descriptive Statistic Functions

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

Notes about Financial Functions

Types of Financial Functions

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

Table 1.2 *Types of Financial Functions*

Function Type	Function	Description
Cashflow	CONVX, CONVXP	calculates convexities for cashflows
	DUR, DURP	calculates modifies duration for cashflows.
	PVP, YIELDP	calculates present value and yield-to-maturity for a periodic cashflow
Parameter calculations	COMPOUND	calculates compound interest parameters
	MORT	calculates amortization parameters
Internal rate of return	INTRR, IRR	calculates the internal rate of return
Net present and future value	NETPV, NPV	calculates net present and future values
	SAVING	calculates the future value of periodic saving
Depreciation	DACCxx	calculates the accumulated depreciation up to the specified period
	DEPxxx	calculates depreciation for a single period
Pricing	BLKSHCLPRC, BLKSHPTPRC	calculated call prices and put prices for European options on stocks, based on the Black-Scholes model
	BLACKPLPRC, BLACKPTPRC	calculates call prices and put prices for European options on futures, based on the Black model
	GARKHCLPRC, GARKHPTPRC	calculates call prices and put prices for European options on stocks, based on the Garman-Kohlhagen model
	MARGRCLPRC, MARGRPTPRC	calculates call options and put prices for European options on stocks, based on the Margrabe model

Using Pricing Functions

A pricing model is used to calculate a theoretical market value (price) for a financial instrument. This value is referred to as a mark-to-market (MTM) value. Typically, a pricing function has the following form:

price = *function*(*rf*1, *rf*2, *rf*3, ...)

In the pricing function, *rf1*, *rf2*, and *rf3* are risk factors such as interest rates or foreign exchange rates. The specific values of the risk factors that are used to calculate the MTM value are the base case values. The set of base case values is known as the base case market state.

After determining the MTM value, you can perform the following tasks with the base case values of the risk factors (*rf1*, *rf2*, and *rf3*):

- Set the base case values to specific values to perform scenario analyses.
- Set the base case values to a range of values to perform profit/loss curve analyses and profit/loss surface analyses.
- Automatically set the base case values to different values to calculate sensitivities - that is, to calculate the delta and gamma values of the risk factors.
- Perturb the base case values to create many possible market states so that many possible future prices can be calculated, and simulation analyses can be performed. For Monte Carlo simulation, the values of the risk factors are generated using mathematical models and the copula methodology.

A list of pricing functions and their descriptions are included in "Types of Financial Functions" on page 6.

Using DATA Step Functions within Macro Functions

The macro functions %SYSFUNC and %QSYSFUNC can call most DATA step functions to generate text in the macro facility. %SYSFUNC and %QSYSFUNC have one difference: %QSYSFUNC masks special characters and mnemonics and %SYSFUNC does not. For more information about these functions, see %QSYSFUNC and %SYSFUNC in *SAS Macro Language: Reference*.

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

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

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

```
%sysfunc(compress(%sysfunc(getoption(sasautos)),
   %str(%)%('')));
```

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

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

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

```
%let dsid=%sysfunc(open(&mydata,&mode));
```

Using CALL Routines and the %SYSCALL Macro Statement

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

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

- “CALL COMPCOST Routine” on page 165
- “LEXCOMB Function” on page 622
- “LEXPERK Function” on page 627
- “CALL LEXPERM Routine” on page 187
- “CALL PRXCHANGE Routine” on page 198
- “CALL PRXNEXT Routine” on page 203
- “CALL PRXSUBSTR Routine” on page 208
- “CALL SCAN Routine” on page 237
- “CALL SORTC Routine” on page 249
- “CALL STDIZE Routine” on page 251
- “CALL SYSTEM Routine” on page 259

In comparison, %SYSCALL invokes a CALL routine and returns an unresolved value, which contains delta characters. %SYSFUNC invokes a function and returns a resolved value, which does not contain delta characters. For more information, see “Macro Quoting” in Chapter 7 of *SAS Macro Language: Reference*, “%SYSCALL Statement” in *SAS Macro Language: Reference*, and “%SYSFUNC and %QSYSFUNC Functions” in *SAS Macro Language: Reference*.

Using Functions to Manipulate Files

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

When you use external files, the FOPEN function allocates a buffer called the File Data Buffer (FDB) and opens the external file for reading or updating. The FREAD function reads a record from the external file and copies the data into the FDB. The FGET function then moves the data to the DATA step variables. The function returns a value that you can check with statements or other functions in the DATA step to determine

how to further process your data. After the records are processed, the FWRITE function writes the contents of the FDB to the external file, and the FCLOSE function closes the file.

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

For a complete listing of functions and CALL routines, see "SAS Functions and CALL Routines by Category" on page 65. For complete descriptions and examples, see the dictionary section of this book.

Function Compatibility with SBCS, DBCS, and MBCS Character Sets

Overview

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

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

- "I18N Level 0" on page 10
- "I18N Level 1" on page 10
- "I18N Level 2" on page 10

For more information about function compatibility, see "Internationalization Compatibility for SAS String Functions" in Chapter 10 of *SAS National Language Support (NLS): Reference Guide*.

I18N Level 0

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

I18N Level 1

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

I18N Level 2

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

Using Random-Number Functions and CALL Routines

Types of Random-Number Functions

Two types of random-number functions are available in SAS. The newest random-number function is the RAND function. It uses the Mersenne-Twister pseudo-random number generator (RNG) that was developed by Matsumoto and Nishimura (1998). This RNG has a very long period of $2^{19937} - 1$, and has very good statistical properties. (A period is the number of occurrences before the pseudo-random number sequence repeats.)

The RAND function is started with a single seed. However, the state of the process cannot be captured by a single seed, which means that you cannot stop and restart the generator from its stopping point. Use the STREAMINIT function to produce a sequence of values that begins at the beginning of a stream. For more information, see the Details section of the "RAND Function" on page 806.

The older random-number generators include the UNIFORM, NORMAL, RANUNI, RANNOR, and other functions that begin with RAN. These functions have a period of only $2^{31} - 2$ or less. The pseudo-random number stream is started with a single seed, and the state of the process can be captured in a new seed. This means that you can stop and restart the generator from its stopping point by providing the proper seed to the corresponding CALL routines. You can use the random-number functions to produce a sequence of values that begins in the middle of a stream.

Seed Values

Random-number functions and CALL routines generate streams of pseudo-random numbers from an initial starting point, called a *seed*, that either the user or the computer clock supplies. A seed must be a nonnegative integer with a value less than $2^{31}–1$ (or 2,147,483,647). If you use a positive seed, you can always replicate the stream of random numbers by using the same DATA step. If you use zero as the seed, the computer clock initializes the stream, and the stream of random numbers cannot be replicated.

Understanding How Functions Generate a Random-Number Stream

Using the DATA Step to Generate a Single Stream of Random Numbers

The DATA steps in this section illustrate several properties of the random-number functions. Each of the DATA steps that call a function generates a single stream of pseudo-random numbers based on a seed value of 7, because that is the first seed for the first call for every step. Some of the DATA steps change the seed value in various ways. Some of the steps have single function calls and others have multiple function calls. None of these DATA steps change the seed. The only seed that is relevant to the function calls is the seed that was used with the first execution of the first random-number function. There is no way to create separate streams with functions (CALL routines are used for this purpose), and the only way that you can restart the function random-number stream is to start a new DATA step.

The following example executes multiple DATA steps:

```
  /* This DATA step produces a single stream of random numbers */
   /* based on a seed value of 7.                               */
data a;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
   a = ranuni (7); output;
run;

   /* This DATA step uses a DO statement to produce a single */
   /* stream of random numbers based on a seed value of 7.   */
data b (drop = i);
   do i = 7 to 18;
      b = ranuni (i);
      output;
   end;
run;

   /* This DATA step uses a DO statement to produce a single */
   /* stream of random numbers based on a seed value of 7.   */
data c (drop = i);
   do i = 1 to 12;
      c = ranuni (7);
      output;
   end;
run;

   /* This DATA step calls the RANUNI and the RANNOR functions */
   /* and produces a single stream of random numbers based on  */
   /* a seed value of 7.                                       */
data d;
   d = ranuni (7); f = ' '; output;
   d = ranuni (8); f = ' '; output;
   d = rannor (9); f = 'n'; output;
   d = .;          f = ' '; output;
   d = ranuni (0); f = ' '; output;
   d = ranuni (1); f = ' '; output;
   d = rannor (2); f = 'n'; output;
   d = .;          f = ' '; output;
   d = ranuni (3); f = ' '; output;
   d = ranuni (4); f = ' '; output;
   d = rannor (5); f = 'n'; output;
   d = .;          f = ' '; output;
run;
```

```
   /* This DATA step calls the RANNOR function and produces a     */
   /* single stream of random numbers based on a seed value of 7. */
data e (drop = i);
   do i = 1 to 6;
      e = rannor (7); output;
      e = .;          output;
   end;
run;

   /* This DATA step merges the output data sets that were */
   /* created from the previous five DATA steps.           */
data five;
   merge a b c d e;
run;

   /* This procedure writes the output from the merged data sets. */
proc print label data=five;
   options missing = ' ';
   label f = '00'x;
   title 'Single Random Number Streams';
run;
```

The following output shows the program results.

Display 1.1 *Results from Generating a Single Random-Number Stream*

Single Random Number Streams

Obs	a	b	c	d		e
1	0.29474	0.29474	0.29474	0.29474		0.39464
2	0.79062	0.79062	0.79062	0.79062		
3	0.79877	0.79877	0.79877	0.26928	n	0.26928
4	0.81579	0.81579	0.81579			
5	0.45122	0.45122	0.45122	0.45122		0.27475
6	0.78494	0.78494	0.78494	0.78494		
7	0.80085	0.80085	0.80085	-0.11729	n	-0.11729
8	0.72184	0.72184	0.72184			
9	0.34856	0.34856	0.34856	0.34856		-1.41879
10	0.46597	0.46597	0.46597	0.46597		
11	0.73523	0.73523	0.73523	-0.39033	n	-0.39033
12	0.66709	0.66709	0.66709			

The pseudo-random number streams in output data sets A, B, and C are identical. The stream in output data set D mixes calls to the RANUNI and the RANNOR functions. In

observations 1, 2, 5, 6, 9, and 10, the values that are returned by RANUNI exactly match the values in the previous streams. Observations 3, 7, and 11, which are flagged by "n", contain the values that are returned by the RANNOR function. The mix of the function calls does not affect the generation of the pseudo-random number stream. All of the results are based on a single stream of uniformly distributed values, some of which are transformed and returned from other functions such as RANNOR. The results of the RANNOR function are produced from two internal calls to RANUNI. The DATA step that creates output data set D executes the following steps three times to create 12 observations:

- call to RANUNI
- call to RANUNI
- call to RANNOR (which internally calls RANUNI twice)
- skipped line to compensate for the second internal call to RANUNI

In the DATA step that creates data set E, RANNOR is called six times, each time skipping a line to compensate for the fact that two internal calls to RANUNI are made for each call to RANNOR. Notice that the three values that are returned from RANNOR in the DATA step that creates data set D match the corresponding values in data set E.

Using the %SYSFUNC Macro to Generate a Single Stream of Random Numbers

When the RANUNI function is called through the macro language by using %SYSFUNC, one pseudo-random number stream is created. You cannot change the seed value unless you close SAS and start a new SAS session. The %SYSFUNC macro produces the same pseudo-random number stream as the DATA steps that generated the data sets A, B, and C for the first macro invocation only. Any subsequent macro calls produce a continuation of the single stream.

```
%macro ran;
   %do i = 1 %to 12;
      %let x = %sysfunc (ranuni (7));
      %put &x;
   %end;
   %mend;

%ran;
```

SAS writes the following output to the log:

Log 1.2 *Results of Execution with the %SYSFUNC Macro*

```
10   %macro ran;
11      %do i = 1 %to 12;
12         %let x = %sysfunc (ranuni (7));
13         %put &x;
14      %end;
15   %mend;
16   %ran;
0.29473798875451
0.79062100955779
0.79877014262544
0.81579051763554
0.45121804506109
0.78494144826426
0.80085421204606
0.72184205973606
0.34855818345609
0.46596586120592
0.73522999404707
0.66709365028287
```

Comparison of Seed Values in Random-Number Functions and CALL Routines

Each random-number function and CALL routine generates pseudo-random numbers from a specific statistical distribution. Each random-number function requires a seed value expressed as an integer constant or a variable that contains the integer constant. Each CALL routine calls a variable that contains the seed value. Additionally, every CALL routine requires a variable that contains the generated pseudo-random numbers.

The seed variable must be initialized before the first execution of the function or CALL routine. After each execution of a function, the current seed is updated internally, but the value of the seed argument remains unchanged. However, after each iteration of the CALL routine the seed variable contains the current seed in the stream that generates the next pseudo-random number. With a function, it is not possible to control the seed values, and, therefore, the pseudo-random numbers after the initialization.

Except for the NORMAL and UNIFORM functions, which are equivalent to the RANNOR and RANUNI functions, respectively, SAS provides a CALL routine that has the same name as each random-number function. Using CALL routines gives you greater control over the seed values.

Generating Multiple Streams from Multiple Seeds in Random-Number CALL Routines

Overview of Random-Number CALL Routines and Streams

You can use the random-number CALL routines to generate multiple streams of pseudo-random numbers within a single DATA step. If you supply a different seed value to initialize each of the seed variables, the streams of the generated pseudo-random numbers are computationally independent, but they might not be statistically independent unless you select the seed values carefully.

Note: Although you can create multiple streams with multiple seeds, this practice is not recommended. It is always safer to create a single stream. With multiple streams, as the streams become longer, the chances of the stream overlapping increase.

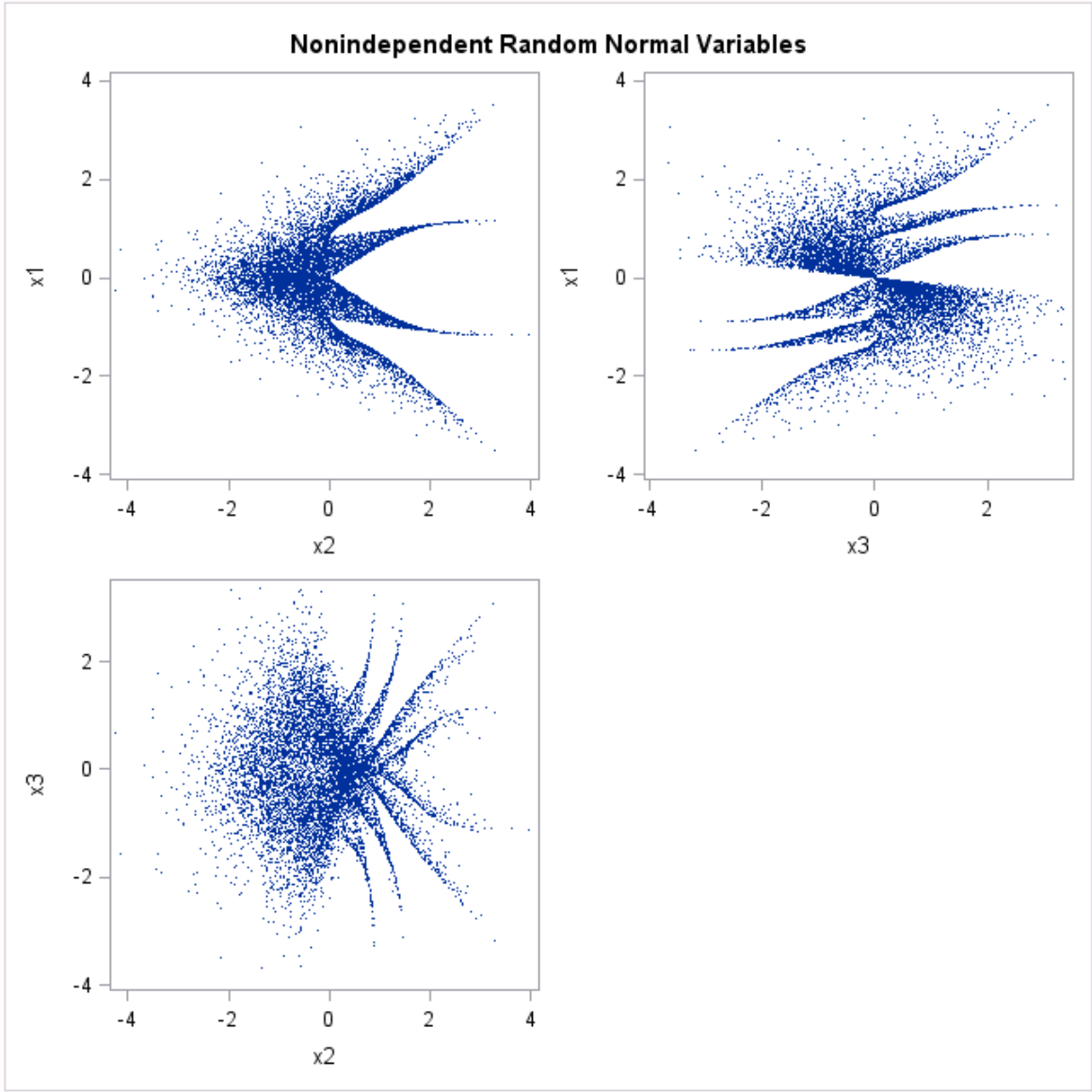
The following two examples deliberately select seeds to illustrate worst-case scenarios. The examples show how to produce multiple streams by using multiple seeds. Although this practice is not recommended, you can use the random-number CALL routines with multiple seeds.

Example 1: Using Multiple Seeds to Generate Multiple Streams

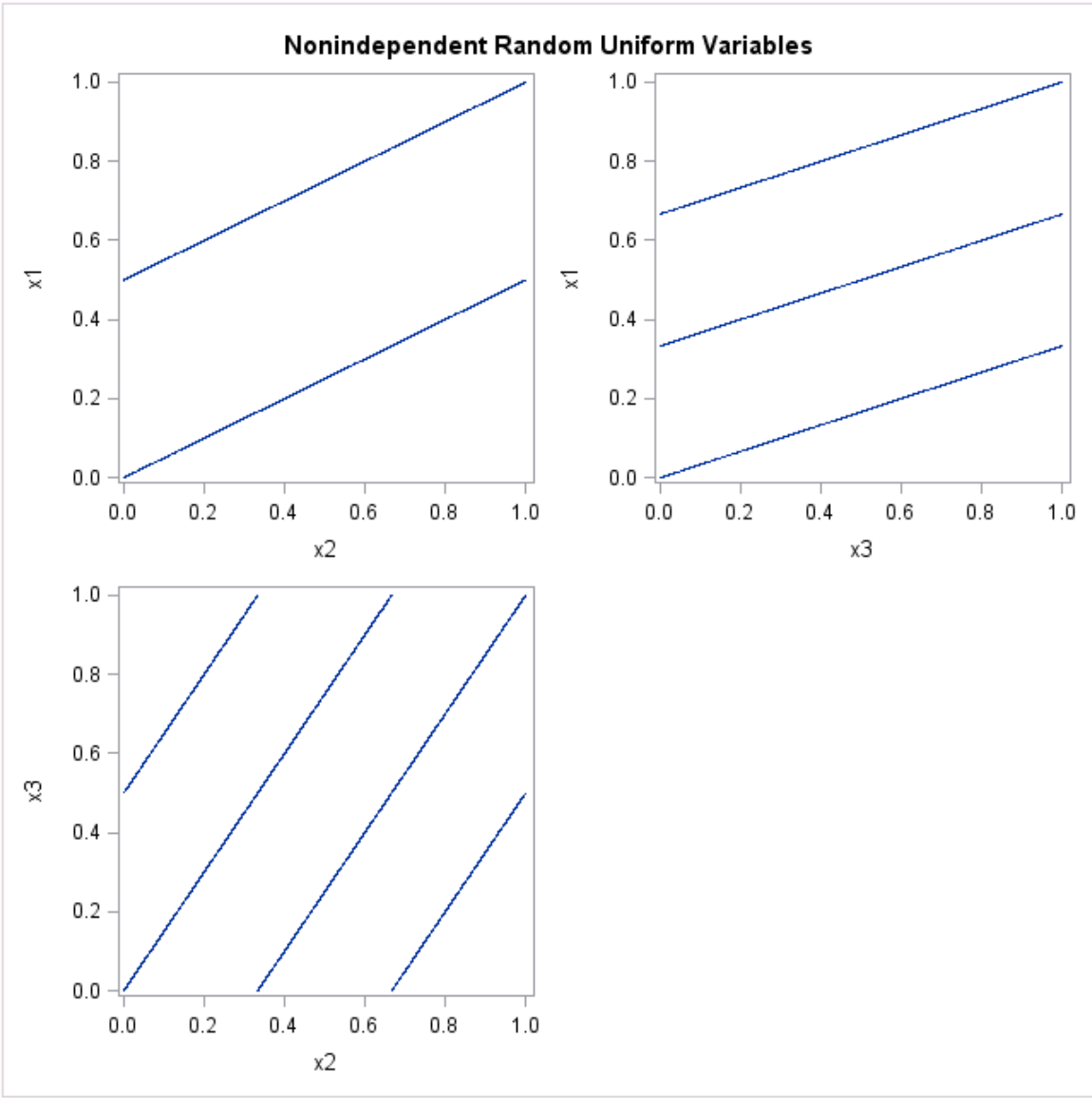
This example shows that you can use multiple seeds to generate multiple streams of pseudo-randomly distributed values by using the random-number CALL routines. The first DATA step creates a data set with three variables that are normally distributed. The second DATA step creates variables that are uniformly distributed. The SGSCATTER procedure (see the *SAS ODS Graphics: Procedures Guide*) is used to show the relationship between each pair of variables for each of the two distributions.

```
data normal;
   seed1 = 11111;
   seed2 = 22222;
   seed3 = 33333;
   do i = 1 to 10000;
      call rannor(seed1, x1);
      call rannor(seed2, x2);
      call rannor(seed3, x3);
      output;
   end;
run;

data uniform;
   seed1 = 11111;
   seed2 = 22222;
   seed3 = 33333;
   do i = 1 to 10000;
      call ranuni(seed1, x1);
      call ranuni(seed2, x2);
      call ranuni(seed3, x3);
      output;
   end;
run;

proc sgscatter data = normal;
   title 'Nonindependent Random Normal Variables';
   plot x1*x2 x1*x3 x3*x2 / markerattrs = (size = 1);
run;

proc sgscatter data = uniform;
   title 'Nonindependent Random Uniform Variables';
   plot x1*x2 x1*x3 x3*x2 / markerattrs = (size = 1);
run;
```

Display 1.2 *Multiple Streams from Multiple Seeds: Nonindependent Random Normal Variables*



Display 1.3 *Multiple Streams from Multiple Seeds: Nonindependent Random Uniform Variables*



The first plot (Display 1.2 on page 17) shows that normal variables appear to be linearly uncorrelated, but they are obviously not independent. The second plot (Display 1.3 on page 18) shows that uniform variables are clearly related. With this class of random-number generators, there is never any guarantee that the streams will be independent.

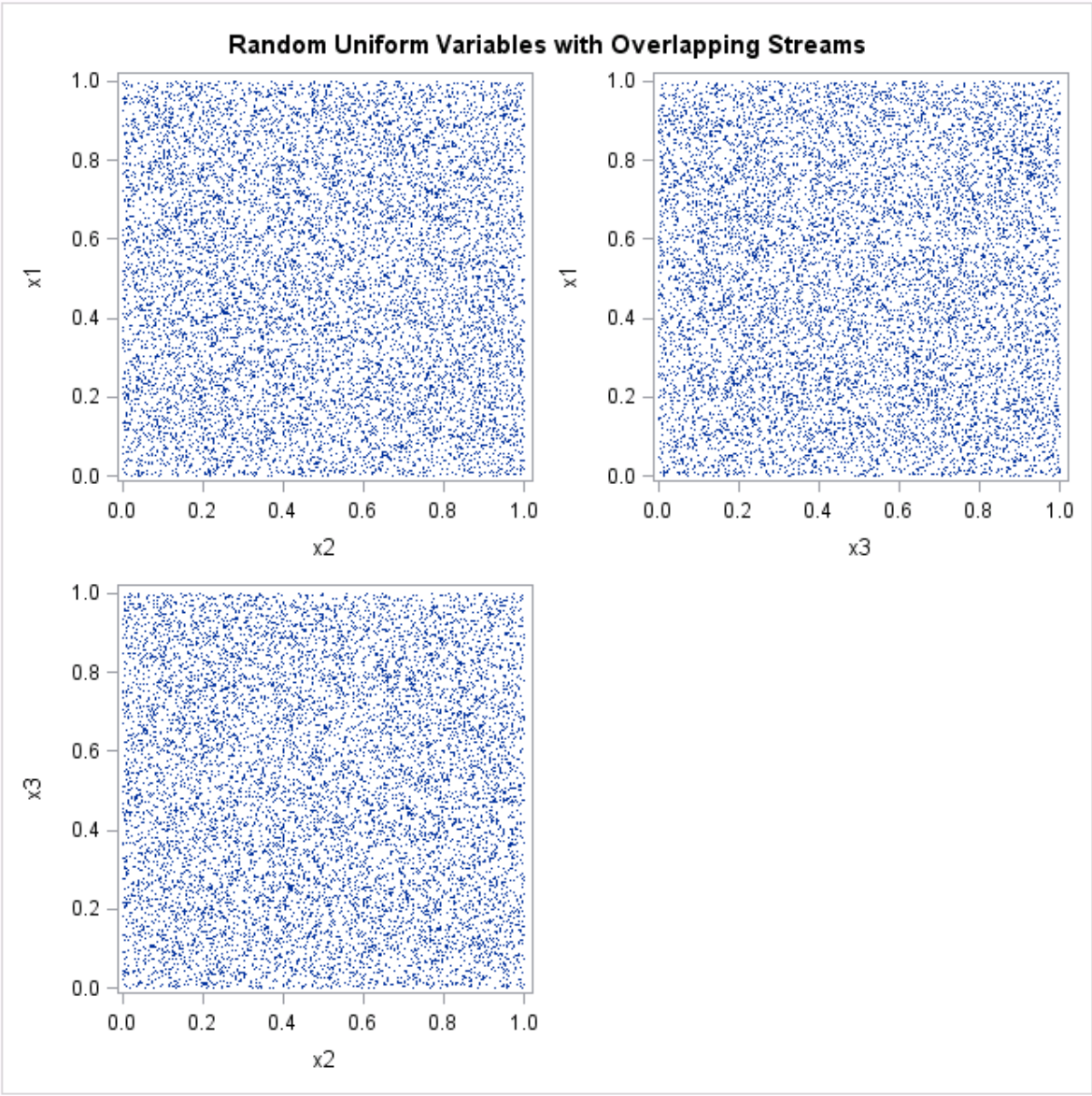
Example 2: Using Different Seeds with the CALL RANUNI Routine

The following example uses three different seeds and the CALL RANUNI routine to produce multiple streams.

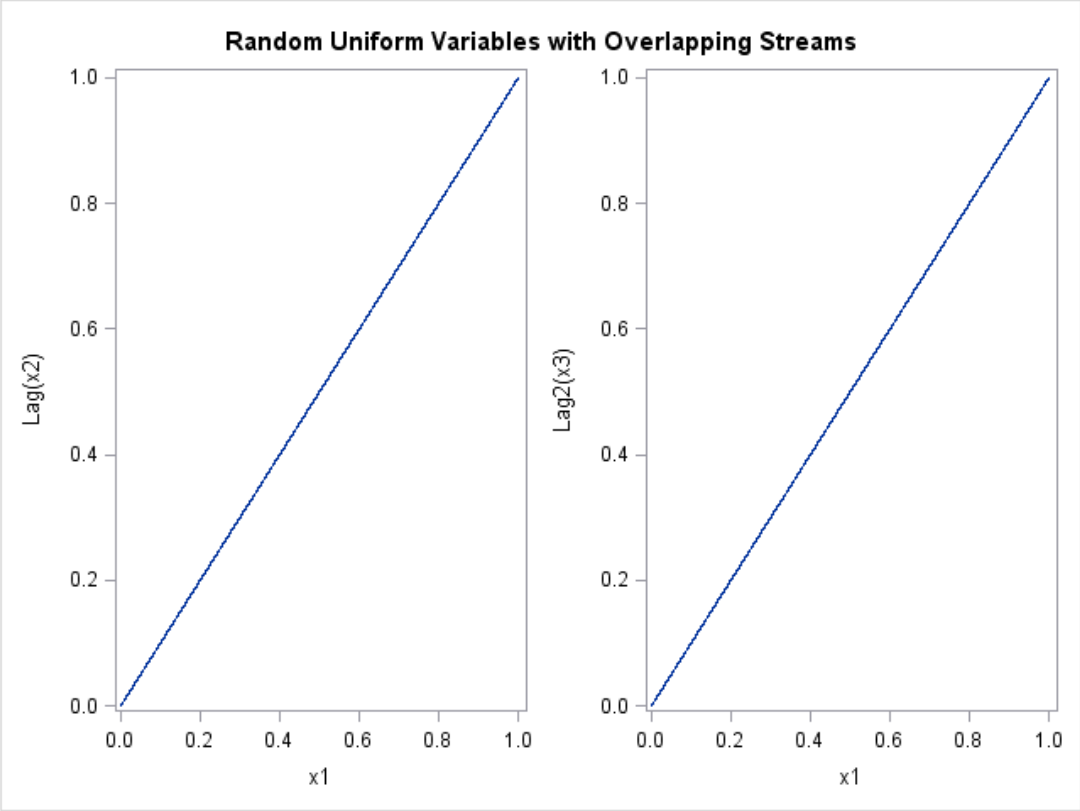
```
data uniform(drop=i);
   seed1 = 255793849;
   seed2 =1408147117;
   seed3 = 961782675;
   do i=1 to 10000;
```

```
      call ranuni(seed1, x1);
      call ranuni(seed2, x2);
      call ranuni(seed3, x3);
      i2 = lag(x2);
      i3 = lag2(x3);
      output;
   end;
label i2='Lag(x2)' i3='Lag2(x3)';
run;

title 'Random Uniform Variables with Overlapping Streams';
proc sgscatter data=uniform;
   plot x1*x2 x1*x3 x3*x2 / markerattrs = (size = 1);
run;

proc sgscatter data=uniform;
   plot i2*x1 i3*x1 / markerattrs = (size = 1);
run;

proc print noobs data=uniform(obs=10);
run;
```

Display 1.4 *Using Different Seeds with CALL RANUNI: Random Uniform Variables with Overlapping Streams, Plot 1*



Display 1.5 *Using Different Seeds with CALL RANUNI: Random Uniform Variables with Overlapping Streams, Plot 2*



Display 1.6 *Random Uniform Variables with Overlapping Streams*

Random Uniform Variables with Overlapping Streams

seed1	seed2	seed3	x1	x2	x3	i2	i3
1408147117	961782675	383001085	0.65572	0.44786	0.17835	.	.
961782675	383001085	1989090982	0.44786	0.17835	0.92624	0.44786	.
383001085	1989090982	1375749095	0.17835	0.92624	0.64063	0.17835	0.17835
1989090982	1375749095	89319994	0.92624	0.64063	0.04159	0.92624	0.92624
1375749095	89319994	1345897251	0.64063	0.04159	0.62673	0.64063	0.64063
89319994	1345897251	561406336	0.04159	0.62673	0.26143	0.04159	0.04159
1345897251	561406336	1333490358	0.62673	0.26143	0.62095	0.62673	0.62673
561406336	1333490358	963442111	0.26143	0.62095	0.44864	0.26143	0.26143
1333490358	963442111	1557707418	0.62095	0.44864	0.72536	0.62095	0.62095
963442111	1557707418	137842443	0.44864	0.72536	0.06419	0.44864	0.44864

The first plot (Display 1.4 on page 20) shows expected results: the variables appear to be statistically independent. However, the second plot (Display 1.5 on page 21) and the listing of the first 10 observations show that there is almost complete overlap between the two streams. The last 9999 values in x1 match the first 9999 values in x2, and the last 9998 values in x1 match the first 9998 values in x3. In other words, there is perfect agreement between the nonmissing parts of x1 and lag(x2) and also x1 and lag2(x3). Even if the streams appear to be independent at first glance as in the first plot, there might be overlap, which might be undesirable depending on how the streams are used.

In practice, if you make multiple small streams with separate and randomly selected seeds, you probably will not encounter the problems that are shown in the first two examples. Display 1.5 on page 21 deliberately selects seeds to illustrate worst-case scenarios.

It is always safer to create a single stream. With multiple streams, as the streams get longer, the chances of the streams overlapping increase.

Generating Multiple Variables from One Seed in Random-Number Functions

Overview of Functions and Streams

If you use functions in your program, you cannot generate more than one stream of pseudo-random numbers by supplying multiple seeds within a DATA step.

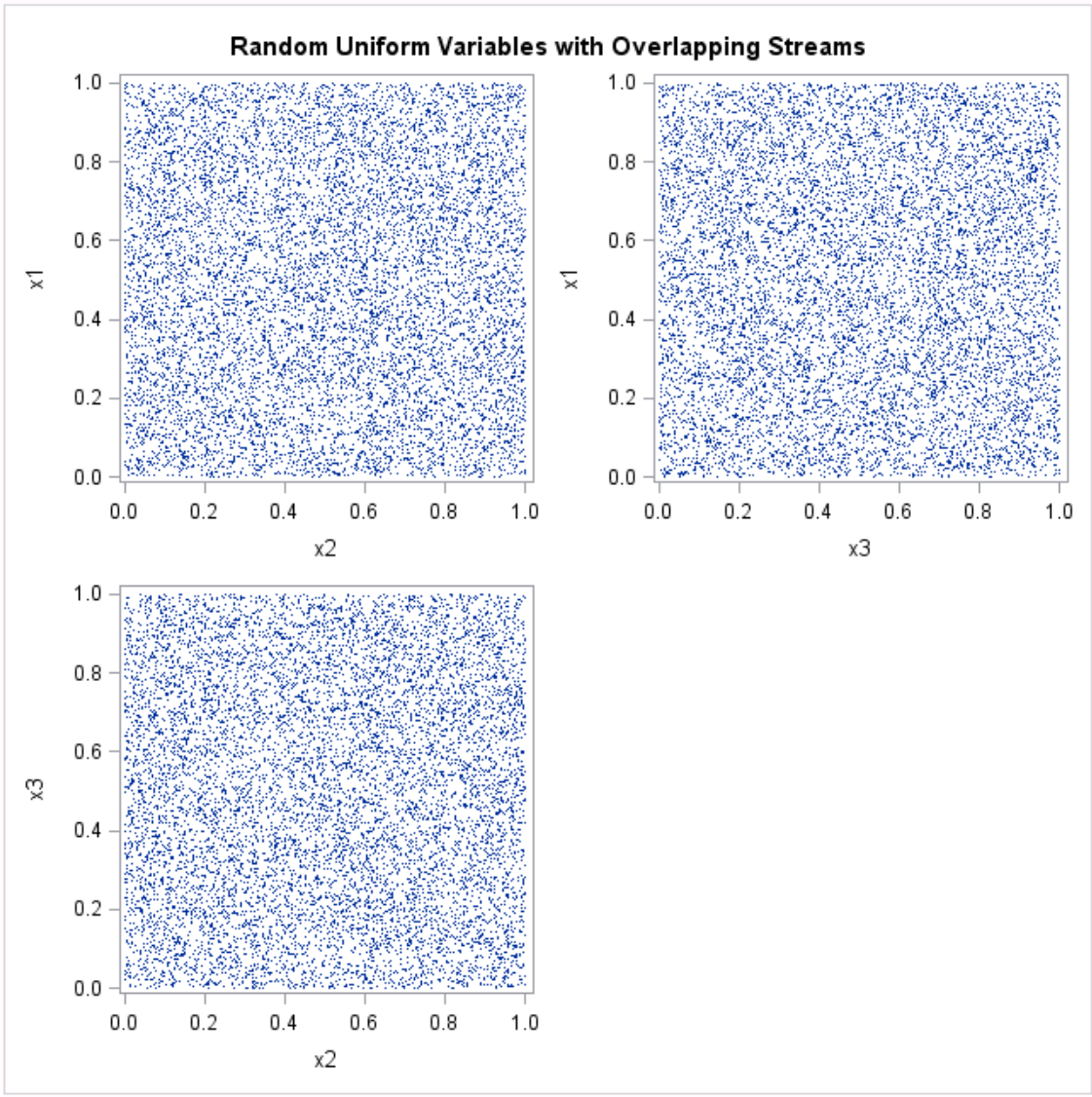
The following example uses the RANUNI function to show the safest way to create multiple variables from the same stream with a single seed.

Example: Generating Random Uniform Variables with Overlapping Streams

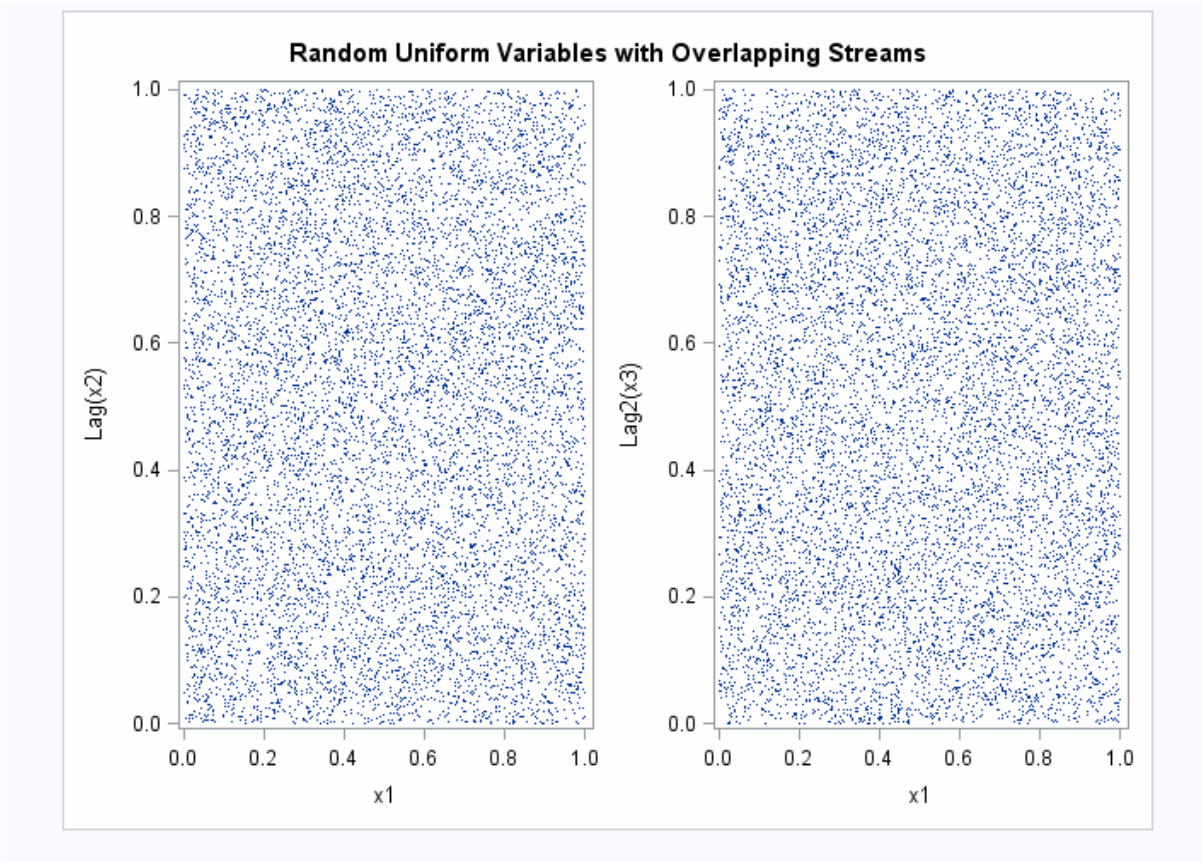
In the following example, the RANUNI function is used to create random uniform variables with overlapping streams. The example shows the safest way to create multiple variables by using the RANUNI function. All variables are created from the same stream with a single seed.

```
data uniform(drop=i);
   do i = 1 to 10000;
      x1 = ranuni(11111);
      x2 = ranuni(11111);
      x3 = ranuni(11111);
      i2 = lag(x2);
      i3 = lag2(x3);
      output;
   end;
label i2 = 'Lag(x2)' i3 = 'Lag2(x3)';
run;

title 'Random Uniform Variables with Overlapping Streams';
proc sgscatter data = uniform;
   plot x1*x2 x1*x3 x3*x2 / markerattrs = (size = 1);
run;

proc sgscatter data = uniform;
   plot i2*x1 i3*x1 / markerattrs = (size = 1);
run;
```

Display 1.7 *Random Uniform Variables with Overlapping Streams: Plot 1*



Display 1.8 *Random Uniform Variables with Overlapping Streams: Plot 2*



In "Example: Generating Random Uniform Variables with Overlapping Streams" on page 23, it appears that the variables are independent. However, even this programming approach might not work well in general. The random-number functions and CALL routines have a period of only 2^{31} - 2 or less (approximately 2.1 billion). When this limit is reached, the stream repeats. Modern computers performing complicated simulations can easily exhaust the entire stream in minutes.

Using the RAND Function as an Alternative

A better approach to generating random uniform variables is to use the RAND function, where multiple streams are not permitted. The RAND function has a period of 2^{19937} - 1. This limit will never be reached, at least with computers of the early 21st century. The number 2^{19937} - 1 is approximately 10^{6000} (1 followed by 6000 zeros). In comparison, the largest value that can be represented in eight bytes on most computers that run SAS is approximately 10^{307}.

The RAND function, which is the latest random-number function that was designed, does not allow multiple streams. The RAND function uses a different algorithm from the random-number CALL routines, which enable you to create multiple streams with multiple seeds. Because the state of the RAND process cannot be captured by a single seed, you cannot stop and restart the generator from its stopping point. Therefore, the RAND function allows only a single stream of numbers, but it can be used to make multiple streams, just as the RANUNI function can.

Effectively Using the Random-Number CALL Routines

Starting, Stopping, and Restarting a Stream

A reasonable use of the random-number CALL routines is starting and stopping a single stream, provided the stream never exhausts the RANUNI stream. For example, you might want SAS to perform iterations, stop, evaluate the results, and then restart the stream at the point that it stopped. The following example illustrates this principle.

Example: Starting, Stopping, and Restarting a Stream

This example generates a stream of five numbers, stops, restarts, generates five more numbers from the same stream, combines the results, and generates the full stream for comparison. In the first DATA step, the state of the random-number seed is stored in a macro variable seed for use as the starting seed in the next step. The separate streams in the example output match the full stream.

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call ranuni(seed, x);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call ranuni(seed, x);
      output;
   end;
run;

data all;
   set u1 u2;
   z = ranuni(104);
run;

proc print label;
   title 'Random Uniform Variables with Overlapping Streams';
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 1.9 *Starting, Stopping, and Restarting a Stream*

Random Uniform Variables with Overlapping Streams

Obs	Separate Streams	Single Stream
1	0.23611	0.23611
2	0.88923	0.88923
3	0.58173	0.58173
4	0.97746	0.97746
5	0.84667	0.84667
6	0.80484	0.80484
7	0.46983	0.46983
8	0.29594	0.29594
9	0.17858	0.17858
10	0.92292	0.92292

Comparison of Changing the Seed in a CALL Routine and in a Function

Example: Changing Seeds in a CALL Routine and in a Function

If you use a CALL routine to change the seed, the results are different from using a function to change the seed. The following example shows the difference.

```
data seeds;
   retain Seed1 Seed2 Seed3 104;
   do i = 1 to 10;
      call ranuni(Seed1,X1);
      call ranuni(Seed2,X2);
      X3 = ranuni(Seed3);
      if i = 5 then do;
         Seed2 = 17;
         Seed3 = 17;
      end;
      output;
   end;
run;

proc print data = seeds;
   title 'Random Uniform Variables with Overlapping Streams';
   id i;
```

```
run;
```

Display 1.10 *Changing Seeds in a CALL Routine and in a Function*

Random Uniform Variables with Overlapping Streams

i	Seed1	Seed2	Seed3	X1	X2	X3
1	507036483	507036483	104	0.23611	0.23611	0.23611
2	1909599212	1909599212	104	0.88923	0.88923	0.88923
3	1249251009	1249251009	104	0.58173	0.58173	0.58173
4	2099077474	2099077474	104	0.97746	0.97746	0.97746
5	1818205895	17	17	0.84667	0.84667	0.84667
6	1728390132	310018657	17	0.80484	0.14436	0.80484
7	1008960848	1055505749	17	0.46983	0.49151	0.46983
8	635524535	1711572821	17	0.29594	0.79701	0.29594
9	383494893	879989345	17	0.17858	0.40978	0.17858
10	1981958542	1432895200	17	0.92292	0.66724	0.92292

Changing Seed2 in the CALL RANUNI statement when i=5, forces the stream for X2 to deviate from the stream for X1. However, changing Seed3 in the RANUNI function has no effect. The X3 stream continues on as if nothing has changed, and the X1 and X3 streams are the same.

Using SYSRANDOM and SYSRANEND Macro Variables to Produce Random Number Streams

Overview of the SYSRANDOM and SYSRANEND Macro Variables

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

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

The SYSRANDOM Macro Variable

The SYSRANDOM macro variable stores the random number seed from the most recent procedure. The macro variable corresponds to the integer that is specified in the SEED= option. Many procedures, such as FREQ, GLM, MI, MCMC, OPTEX, PHREG, and

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

The SYSRANEND Macro Variable

The SYSRANEND macro variable stores a seed that can be used to start the next step in your program. In some cases, this seed captures the state of the random number process when a procedure is completed. Your program can contain multiple steps and can control the random number sequence without specifying an explicit seed for each procedure. You can start the simulation with one seed, and use the SYSRANEND macro variable to provide the seed in all subsequent procedures. In some cases, you can also use the SYSRANEND macro variable to stop and restart the random number generators (RNGs) that are continuing the same stream.

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

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

Example: Reproducing Results

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

```
title 'Bayesian Linear Regression';

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

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

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

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

```
proc mcmc data=sashelp.class seed=&sysrandom outpost=out2;
   parms beta0 0 beta1 0;
   prior beta0 beta1 ~ normal(mean=0, var=1e6);
```

```
   mu=beta0 + beta1*height;
   model weight ~ n(mu, var=137);
run;
```

You can submit the following step to see that the two PROC MCMC executions produce identical samples:

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

Example: Creating a Reproducible Random Number Stream

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

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

%put sysranend=&sysranend;

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

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

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

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

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

```
      run;
      %end;
   ods listing;

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

%design;
```

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

Date and Time Intervals

Definition of a Date and Time Interval

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

Interval Names and SAS Dates

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

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

Incrementing Dates and Times by Using Multipliers and by Shifting Intervals

SAS provides date, time, and datetime intervals for counting different periods of elapsed time. By using multipliers and shift indexes, you can create multiples of intervals and shift their starting point to construct more complex interval specifications.

The general form of an interval name is

name<multiplier><.shift-index>

Both the *multiplier* 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 that begin in January. If you specify other values for *multiplier* and for *shift-index*, you can create multiple intervals that begin in different parts of the year. For example, the interval WEEK6.11 specifies six-week intervals starting on second Wednesdays.

For more information, see "Single-Unit Intervals" in Chapter 7 of *SAS Language Reference: Concepts*, "Multi-Unit Intervals" in Chapter 7 of *SAS Language Reference: Concepts*, and "Shifted Intervals" in Chapter 7 of *SAS Language Reference: Concepts*.

Commonly Used Time Intervals

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

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

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

The following table lists time intervals that are commonly used.

Table 1.3 *Commonly Used Intervals with Optional Multiplier and Shift Indexes*

Interval	Description
DAY3	Three-day intervals
WEEK	Weekly intervals starting on Sundays
WEEK.7	Weekly intervals starting on Saturdays
WEEK6.13	Six-week intervals starting on second Fridays
WEEK2	Biweekly intervals starting on first Sundays
WEEK1.1	Same as WEEK
WEEK.2	Weekly intervals starting on Mondays
WEEK6.3	Six-week intervals starting on first Tuesdays
WEEK6.11	Six-week intervals starting on second Wednesdays
WEEK4	Four-week intervals starting on first Sundays
WEEKDAY	Five-day work week with a Saturday-Sunday weekend

Interval	Description
WEEKDAY1W	Six-day week with Sunday as a weekend day
WEEKDAY35W	Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)
WEEKDAY17W	Same as WEEKDAY
WEEKDAY67W	Five-day week with Friday and Saturday as weekend days
WEEKDAY3.2	Three-weekday intervals with Saturday and Sunday as weekend days (The intervals are aligned with respect to Jan. 1, 1960. For intervals that nest within a year, it is not necessary to go back to Jan. 1, 1960 to determine the alignment.)
TENDAY4.2	Four ten-day periods starting at the second TENDAY period
SEMIMONTH2.2	Intervals from the sixteenth of one month through the fifteenth of the next month
MONTH2.2	February–March, April–May, June–July, August–September, October–November, and December–January of the following year
MONTH2	January–February, March–April, May–June, July–August, September–October, November–December
QTR3.2	Nine-month intervals starting on February 1, 1960, November 1, 1960, August 1, 1961, May 1, 1962, and so on.
SEMIYEAR.3	Six-month intervals, March–August and September–February
YEAR.10	Fiscal years starting in October
YEAR2.7	Biennial intervals starting in July of even years
YEAR2.19	Biennial intervals starting in July of odd years
YEAR4.11	Four-year intervals starting in November of leap years (frequency of U.S. presidential elections)
YEAR4.35	Four-year intervals starting in November of even years between leap years (frequency of U.S. midterm elections)
DTMONTH13	Thirteen-month intervals starting at midnight of January 1, 1960, such as November 1, 1957, December 1, 1958, January 1, 1960, February 1, 1961, and March 1, 1962
HOUR8.7	Eight-hour intervals starting at 6 a.m., 2 p.m., and 10 p.m. (might be used for work shifts)

For a complete list of the valid values for *interval*, see Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts*.

Retail Calendar Intervals: ISO 8601 Compliant

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third months of each period, respectively.

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

For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in Chapter 7 of *SAS Language Reference: Concepts*.

Custom Time Intervals

Reasons for Using Custom Time Intervals

Standard time intervals (for example, QTR, MONTH, WEEK, and so on) do not always fit the data. Additionally, some time series are measured at standard intervals where there are gaps in the data. For example, you might want to use fiscal months that begin on the 10th day of each month. In this case, using the MONTH interval is not appropriate because the MONTH interval begins on the 1st day of each month. You can use a custom interval to model data at a frequency that is familiar to the business and to eliminate gaps in the data by compressing the data. The intervals must be listed in ascending order. There cannot be gaps between intervals, and intervals cannot overlap.

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

Using Custom Time Intervals in a SAS Program

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

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

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

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

   For more information, see "INTERVALDS= System Option" in *SAS System Options: Reference*.

2. Create a data set that describes the custom interval.

   The data set must contain the *begin* variable; it can also contain *end* and *season* variables. In your SAS program, include a FORMAT statement that is associated with the *begin* variable that specifies a SAS date, datetime, or numeric format that matches the *begin* variable data. If an *end* variable is present, include it in the FORMAT statement. A numeric format that is not a SAS date or SAS datetime

format indicates that the values are observation numbers. If the *end* variable is not present, the implied value of *end* at each observation is one less than the value of *begin* at the next observation.

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

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

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

The following DATA step creates the StoreHoursDS data set for a business that is open from 9:00 AM to 6:00 PM Monday through Friday, and Saturday from 9:00 AM to 1:00 PM. The example uses the "INTNX Function" on page 580, which increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value. In this example, StoreHours is the interval, and StoreHoursDS is the data set that contains user-supplied holidays:

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

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

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

Display 1.11 *A Custom Interval for Store Hours*

Store Hours Custom Interval

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

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

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

```
options intervalds=(FiscalMonth=FMDS);
data FMDS(keep=begin season);
   start = '10JAN1999'd;
   stop  = '10JAN2001'd;
   nmonths = intck('month',start,stop);
   do i=0 to nmonths;
      begin = intnx('month',start,i,'S');
      season = month(begin);
      output;
   end;
   format begin date9.;
```

```
run;

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

Display 1.12 *Fiscal Month Data*

Fiscal Month Data

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

The difference between the custom FiscalMonth interval and a standard interval is seen in the following example. The output from the program compares how the data is accumulated. For the FiscalMonth interval, values in the first nine days of the month are accumulated with the interval that begins in the previous month. For the standard MONTH interval, values in the first nine days of the month are accumulated with the calendar month.

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

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

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

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

title 'Standard Monthly Data and Fiscal Month Data';

proc print data=compare;
run;
```

Display 1.13 *Comparison of Standard Monthly Data and Fiscal Month Data*

Standard Monthly Data and Fiscal Month Data

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

Example 3: Using Custom Intervals with the INTCK Function

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

```
options intervalds=(BankingDays=BankDayDS);
data BankDayDS(keep=begin);
   start = '15DEC1998'd;
   stop  = '15JAN2002'd;
   nwkdays = intck('weekday',start,stop);
   do i = 0 to nwkdays;
      begin = intnx('weekday',start,i);
      year = year(begin);
      if begin ne holiday('NEWYEAR',year) and
```

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

   title 'Methods of Counting Days';

   proc print data=CountDays;
   run;
```

Display 1.14 *Bank Days Custom Interval*

Methods of Counting Days

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

Best Practices for Custom Interval Names

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

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

The following paragraphs describe the variables:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Executing **INTSHIFT ('CUSTBASE');** or **INTSHIFT ('CUSTBASE*m.s*');** returns the value of CUSTBASE.

- With INTNX, INTCK, and INTTEST, the intervals CUSTBASE, CUSTBASE*m*, and CUSTBASE*m.s* work as expected.

Pattern Matching Using Perl Regular Expressions (PRX)

Definition of Pattern Matching

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

For example, you can search for multiple occurrences of a string and replace those strings with another string. You can search for a string in your source file and return the position of the match. You can find words in your file that are doubled.

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

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

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

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

Benefits of Using Perl Regular Expressions in the DATA Step

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

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

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

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

Using Perl Regular Expressions in the DATA Step

Syntax of Perl Regular Expressions

The Components of a Perl Regular Expression

Perl regular expressions consist of characters and special characters that are called metacharacters. When performing a match, SAS searches a source string for a substring that matches the Perl regular expression that you specify. Using metacharacters enables SAS to perform special actions. These actions include forcing the match to begin in a particular location, and matching a particular set of characters. Paired forward slashes are the default delimiters. The following two examples show metacharacters and the values that they match:

- If you use the metacharacter \d, SAS matches a digit between 0–9.

- If you use /\dt/, SAS finds the digits in the string “Raleigh, NC 27506”.

You can see lists of PRX metacharacters in “Tables of Perl Regular Expression (PRX) Metacharacters” on page 1003. For a complete list of metacharacters, see the Perl documentation.

Basic Syntax for Finding a Match in a String

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

```
/search-string/source-string/
```

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

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

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

Basic Syntax for Searching and Replacing Text

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

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

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

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

Arguments

s
: specifies the metacharacter for substitution.

world
: specifies the regular expression.

planet
: specifies the replacement value for *world*.

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

Hello world!
: specifies the source string to be searched.

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

Another Example of Using Basic Syntax for Searching and Replacing Text

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

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

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

The Perl regular expression can be divided into its elements:

s
: specifies a substitution regular expression.

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

,<space>
: matches a comma and a space.

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

/
: separator between the regular expression and the replacement string.

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

<space>
: puts a space in the result.

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

Replacing Text

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

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

SAS writes the following output to the log:

```
x=McLAUREN
```

Example 1: Validating Data

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

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

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

```
            putlog "ERROR: Invalid expression " expression;  7
            stop;
         end;
      end;

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

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

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

   datalines;
Jerome Johnson (919)319-1677 (919)846-2198
Romeo Montague 800-899-2164 360-973-6201
Imani Rashid (508)852-2146 (508)366-9821
Palinor Kent . 919-782-3199
Ruby Archuleta . .
Takei Ito 7042982145 .
Tom Joad 209/963/2764 2099-66-8474
;
run;
```

The following items correspond to the lines that are numbered in the DATA step that is shown above.

1 Create a DATA step.

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

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

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

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

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

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

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

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

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

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

Log 1.3 *Output from Validating Data*

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

Example 2: Matching and Replacing Text

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

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

The following items correspond to the numbered lines in the DATA step that is shown above.

1 Create a DATA step.

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

3 Call the PRXCHANGE routine to perform the pattern exchange. The format for the regular expression is **s/regular-expression/replacement-text/**. The **s**

before the regular expression signifies that this is a substitution regular expression. The –1 is a special value that is passed to PRXCHANGE and indicates that all possible replacements should be made.

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

5 Identify the input file.

Log 1.4 *Output from Replacing Text*

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

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

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

Example 3: Extracting a Substring from a String

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

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

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

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

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

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

1 Create a DATA step.

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

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

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

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

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

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

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

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

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

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

9 Search for a valid business phone number.

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

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

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

Log 1.5 *Output from Extracting a Substring from a String*

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

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

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

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

   input first last phone & 16.;
   if prxmatch(re, phone) then do;  4
      area_code = prxposn(re, 1, phone);  5
      if area_code ^in ("828"
                        "336"
                        "704"
                        "910"
                        "919"
                        "252") then
         putlog "NOTE: Not in North Carolina: "
                first last phone;  6
```

```
    end;
   datalines;  7
Thomas Archer    (919)319-1677
Lucy Mallory       (800)899-2164
Tom Joad              (508)852-2146
Laurie Jorgensen  (252)352-7583
;
run;
```

The following items correspond to the numbered lines in the DATA step that is shown above.

1 Create a DATA step.

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

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

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

4 Return the position at which the string begins.

5 Identify the position at which the area code begins.

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

7 Identify the input file.

Log 1.6 *Output from Extracting a Substring from a String*

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

Writing Perl Debug Output to the SAS Log

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

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

```
data _null_;

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

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

SAS writes the following output to the log.

Log 1.7 *SAS Debugging Output*

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

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

For a detailed explanation of Perl debug output, see "CALL PRXDEBUG Routine" on page 200.

Perl Artistic License Compliance

Perl regular expressions are supported beginning with SAS® 9.

The PRX functions use a modified version of Perl 5.6.1 to perform regular expression compilation and matching. Perl is compiled into a library for use with SAS. This library is shipped with SAS® 9. The modified and original Perl 5.6.1 files are freely available in a ZIP file from the **Technical Support Web site**. The ZIP file is provided to comply with the Perl Artistic License and is not required in order to use the PRX functions. Each of the modified files has a comment block at the top of the file describing how and when the file was changed. The executables were given nonstandard Perl names. The standard version of Perl can be obtained from the Perl Web site.

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

- Perl variables (except the capture buffer variables $1 - $n, which are supported).
- The regular expression options /c and /g, and the /e option with substitutions.
- The regular expression option /o in SAS 9.0. (It is supported in SAS 9.1 and later.)
- Named characters, which use the \N{name} syntax.

- The metacharacters \pP, \PP, and \X.
- Executing Perl code within a regular expression, which includes the syntax (?{code}), (??{code}), and (?p{code}).
- Unicode pattern matching.
- Using ?PATTERN?. ? is treated like an ordinary regular expression start and end delimiter.
- The metacharacter \G.
- Perl comments between a pattern and replacement text. For example: s{regexp} # perl comment {replacement} is not supported.
- Matching backslashes with m/\\\/. Instead use m/\\/ to match a backslash.

Base SAS Functions for Web Applications

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

Chapter 2

Dictionary of SAS Functions and CALL Routines

SAS Functions and CALL Routines Documented in Other SAS Publications

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

- SAS Companion for Windows
- SAS Companion for z/OS
- Data Quality Server Reference
- Logging: Configuration and Programming Reference
- SAS Macro Language Reference
- National Language Support: Reference Guide

SAS Functions and CALL Routines by Category

Here are the categories for SAS functions and CALL routines:

Arithmetic	returns the result of a division that handles special missing values for ODS output.
Array	returns information about arrays.
Bitwise Logical Operations	returns the bitwise logical result for an argument.
Character	returns information based on character data.
Character String Matching	returns information from Perl regular expressions.
Combinatorial	generates combinations and permutations.
Date and Time	returns date and time values, including time intervals.
Descriptive Statistics	returns statistical values, such as mean, median, and standard deviation.
Distance	returns the geodetic distance.
External Files	returns information that is associated with external files.
External Routines	returns a character or numeric value, or calls a routine without any return code.
Financial	calculates financial values such as interest, periodic payments, depreciation, and prices for European options on stocks.
Hyperbolic	performs hyperbolic calculations such as sine, cosine, and tangent.
Macro	assigns a value to a macro variable, returns the value of a macro variable, determines whether a macro variable is global or local in scope, and identifies whether a macro variable exists.
Mathematical	performs mathematical calculations such as factorials, absolute value, fuzzy comparisons, and logarithms.
Numeric	returns a numeric value based on whether an expression is true, false, or missing, or determines whether a software image exists in the installed version of SAS.
Probability	returns probability calculations, such as from a chi-square or Poisson distribution.
Quantile	returns a quantile from specific distributions.
Random Number	returns random variates from specific distributions.
SAS File I/O	returns information about SAS files.
Search	searches for character or numeric values.
Sort	sorts the values of character or numeric arguments.

Special	returns and stores memory addresses, writes a value directly into memory, suspends execution of a program, submits an operating-environment command for execution, returns the value of a SAS system or graphics option, specifies formats and informats at run time, returns the system return code, returns the UUID, determines whether a product is licensed, as well as returns other information about SAS processing.
State and ZIP Code	returns ZIP codes, FIPS codes, state and city names, postal codes, and the geodetic distance between ZIP codes.
Trigonometric	returns trigonometric values such as sine, cosine, and tangent.
Truncation	truncates numeric values and returns numeric values, often using fuzzing or zero fuzzing.
Variable Control	assigns variable labels, links SAS data set variables to DATA step or macro variables, and assigns variable names.
Variable Information	returns a name, type, length, informat name, label, and other variable information.
Web Service	calls a Web service or a SAS registered Web service.
Web Tools	encodes and decodes a string of data.

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

Category	Language Elements	Description
Arithmetic	DIVIDE Function (p. 380)	Returns the result of a division that handles special missing values for ODS output.
Array	DIM Function (p. 377)	Returns the number of elements in an array.
	HBOUND Function (p. 528)	Returns the upper bound of an array.
	LBOUND Function (p. 612)	Returns the lower bound of an array.
Bitwise Logical Operations	BAND Function (p. 136)	Returns the bitwise logical AND of two arguments.
	BLSHIFT Function (p. 146)	Returns the bitwise logical left shift of two arguments.
	BNOT Function (p. 147)	Returns the bitwise logical NOT of an argument.
	BOR Function (p. 147)	Returns the bitwise logical OR of two arguments.
	BRSHIFT Function (p. 148)	Returns the bitwise logical right shift of two arguments.
	BXOR Function (p. 149)	Returns the bitwise logical EXCLUSIVE OR of two arguments.
Character	ANYALNUM Function (p. 99)	Searches a character string for an alphanumeric character, and returns the first position at which the character is found.
	ANYALPHA Function (p. 101)	Searches a character string for an alphabetic character, and returns the first position at which the character is found.

Category	Language Elements	Description
	ANYCNTRL Function (p. 103)	Searches a character string for a control character, and returns the first position at which that character is found.
	ANYDIGIT Function (p. 105)	Searches a character string for a digit, and returns the first position at which the digit is found.
	ANYFIRST Function (p. 106)	Searches a character string for a character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.
	ANYGRAPH Function (p. 108)	Searches a character string for a graphical character, and returns the first position at which that character is found.
	ANYLOWER Function (p. 110)	Searches a character string for a lowercase letter, and returns the first position at which the letter is found.
	ANYNAME Function (p. 112)	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.
	ANYPRINT Function (p. 113)	Searches a character string for a printable character, and returns the first position at which that character is found.
	ANYPUNCT Function (p. 116)	Searches a character string for a punctuation character, and returns the first position at which that character is found.
	ANYSPACE Function (p. 118)	Searches a character string for a white-space character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.
	ANYUPPER Function (p. 120)	Searches a character string for an uppercase letter, and returns the first position at which the letter is found.
	ANYXDIGIT Function (p. 121)	Searches a character string for a hexadecimal character that represents a digit, and returns the first position at which that character is found.
	BYTE Function (p. 149)	Returns one character in the ASCII or the EBCDIC collating sequence.
	CALL CATS Routine (p. 159)	Removes leading and trailing blanks, and returns a concatenated character string.
	CALL CATT Routine (p. 161)	Removes trailing blanks, and returns a concatenated character string.
	CALL CATX Routine (p. 163)	Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.
	CALL COMPCOST Routine (p. 165)	Sets the costs of operations for later use by the COMPGED function

Category	Language Elements	Description
	CALL MISSING Routine (p. 191)	Assigns missing values to the specified character or numeric variables.
	CALL SCAN Routine (p. 237)	Returns the position and length of the nth word from a character string.
	CAT Function (p. 263)	Does not remove leading or trailing blanks, and returns a concatenated character string.
	CATQ Function (p. 266)	Concatenates character or numeric values by using a delimiter to separate items and by adding quotation marks to strings that contain the delimiter.
	CATS Function (p. 270)	Removes leading and trailing blanks, and returns a concatenated character string.
	CATT Function (p. 272)	Removes trailing blanks, and returns a concatenated character string.
	CATX Function (p. 274)	Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.
	CHAR Function (p. 298)	Returns a single character from a specified position in a character string.
	CHOOSEC Function (p. 299)	Returns a character value that represents the results of choosing from a list of arguments.
	CHOOSEN Function (p. 300)	Returns a numeric value that represents the results of choosing from a list of arguments.
	COALESCEC Function (p. 307)	Returns the first non-missing value from a list of character arguments.
	COLLATE Function (p. 308)	Returns a character string in ASCII or EBCDIC collating sequence.
	COMPARE Function (p. 311)	Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.
	COMPBL Function (p. 314)	Removes multiple blanks from a character string.
	COMPGED Function (p. 317)	Returns the generalized edit distance between two strings.
	COMPLEV Function (p. 323)	Returns the Levenshtein edit distance between two strings.
	COMPRESS Function (p. 327)	Returns a character string with specified characters removed from the original string.
	COUNT Function (p. 338)	Counts the number of times that a specified substring appears within a character string.
	COUNTC Function (p. 340)	Counts the number of characters in a string that appear or do not appear in a list of characters.

Category	Language Elements	Description
	COUNTW Function (p. 343)	Counts the number of words in a character string.
	DEQUOTE Function (p. 368)	Removes matching quotation marks from a character string that begins with a quotation mark, and deletes all characters to the right of the closing quotation mark.
	FIND Function (p. 457)	Searches for a specific substring of characters within a character string.
	FINDC Function (p. 459)	Searches a string for any character in a list of characters.
	FINDW Function (p. 467)	Returns the character position of a word in a string, or returns the number of the word in a string.
	FIRST Function (p. 479)	Returns the first character in a character string.
	IFC Function (p. 538)	Returns a character value based on whether an expression is true, false, or missing.
	INDEX Function (p. 543)	Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.
	INDEXC Function (p. 545)	Searches a character expression for any of the specified characters, and returns the position of that character.
	INDEXW Function (p. 546)	Searches a character expression for a string that is specified as a word, and returns the position of the first character in the word.
	LEFT Function (p. 616)	Left-aligns a character string.
	LENGTH Function (p. 617)	Returns the length of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string.
	LENGTHC Function (p. 618)	Returns the length of a character string, including trailing blanks.
	LENGTHM Function (p. 619)	Returns the amount of memory (in bytes) that is allocated for a character string.
	LENGTHN Function (p. 621)	Returns the length of a character string, excluding trailing blanks.
	LOWCASE Function (p. 646)	Converts all letters in an argument to lowercase.
	MD5 Function (p. 656)	Returns the result of the message digest of a specified string.
	MISSING Function (p. 662)	Returns a numeric result that indicates whether the argument contains a missing value.
	MVALID Function (p. 676)	Checks the validity of a character string for use as a SAS member name.
	NLITERAL Function (p. 681)	Converts a character string that you specify to a SAS name literal.

Category	Language Elements	Description
	NOTALNUM Function (p. 685)	Searches a character string for a non-alphanumeric character, and returns the first position at which the character is found.
	NOTALPHA Function (p. 687)	Searches a character string for a nonalphabetic character, and returns the first position at which the character is found.
	NOTCNTRL Function (p. 689)	Searches a character string for a character that is not a control character, and returns the first position at which that character is found.
	NOTDIGIT Function (p. 690)	Searches a character string for any character that is not a digit, and returns the first position at which that character is found.
	NOTFIRST Function (p. 694)	Searches a character string for an invalid first character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.
	NOTGRAPH Function (p. 695)	Searches a character string for a non-graphical character, and returns the first position at which that character is found.
	NOTLOWER Function (p. 697)	Searches a character string for a character that is not a lowercase letter, and returns the first position at which that character is found.
	NOTNAME Function (p. 699)	Searches a character string for an invalid character in a SAS variable name under VALIDVARNAME=V7, and returns the first position at which that character is found.
	NOTPRINT Function (p. 701)	Searches a character string for a nonprintable character, and returns the first position at which that character is found.
	NOTPUNCT Function (p. 702)	Searches a character string for a character that is not a punctuation character, and returns the first position at which that character is found.
	NOTSPACE Function (p. 704)	Searches a character string for a character that is not a white-space character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.
	NOTUPPER Function (p. 707)	Searches a character string for a character that is not an uppercase letter, and returns the first position at which that character is found.
	NOTXDIGIT Function (p. 708)	Searches a character string for a character that is not a hexadecimal character, and returns the first position at which that character is found.
	NVALID Function (p. 711)	Checks the validity of a character string for use as a SAS variable name.
	PROPCASE Function (p. 773)	Converts all words in an argument to proper case.
	QUOTE Function (p. 802)	Adds double quotation marks to a character value.

Category	Language Elements	Description
	RANK Function (p. 820)	Returns the position of a character in the ASCII or EBCDIC collating sequence.
	REPEAT Function (p. 829)	Returns a character value that consists of the first argument repeated n+1 times.
	REVERSE Function (p. 830)	Reverses a character string.
	RIGHT Function (p. 832)	Right aligns a character expression.
	SCAN Function (p. 848)	Returns the nth word from a character string.
	SOUNDEX Function (p. 876)	Encodes a string to facilitate searching.
	SPEDIS Function (p. 878)	Determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words.
	STRIP Function (p. 888)	Returns a character string with all leading and trailing blanks removed.
	SUBPAD Function (p. 890)	Returns a substring that has a length you specify, using blank padding if necessary.
	SUBSTR (left of =) Function (p. 891)	Replaces character value contents.
	SUBSTR (right of =) Function (p. 892)	Extracts a substring from an argument.
	SUBSTRN Function (p. 894)	Returns a substring, allowing a result with a length of zero.
	TRANSLATE Function (p. 917)	Replaces specific characters in a character string.
	TRANSTRN Function (p. 919)	Replaces or removes all occurrences of a substring in a character string.
	TRANWRD Function (p. 921)	Replaces all occurrences of a substring in a character string.
	TRIM Function (p. 924)	Removes trailing blanks from a character string, and returns one blank if the string is missing.
	TRIMN Function (p. 926)	Removes trailing blanks from character expressions, and returns a string with a length of zero if the expression is missing.
	UPCASE Function (p. 928)	Converts all letters in an argument to uppercase.
	VERIFY Function (p. 944)	Returns the position of the first character in a string that is not in any of several other strings.
Character String Matching	CALL PRXCHANGE Routine (p. 198)	Performs a pattern-matching replacement.

Category	Language Elements	Description
	CALL PRXDEBUG Routine (p. 200)	Enables Perl regular expressions in a DATA step to send debugging output to the SAS log.
	CALL PRXFREE Routine (p. 202)	Frees memory that was allocated for a Perl regular expression.
	CALL PRXNEXT Routine (p. 203)	Returns the position and length of a substring that matches a pattern, and iterates over multiple matches within one string.
	CALL PRXPOSN Routine (p. 205)	Returns the start position and length for a capture buffer.
	CALL PRXSUBSTR Routine (p. 208)	Returns the position and length of a substring that matches a pattern.
	PRXCHANGE Function (p. 775)	Performs a pattern-matching replacement.
	PRXMATCH Function (p. 780)	Searches for a pattern match and returns the position at which the pattern is found.
	PRXPAREN Function (p. 784)	Returns the last bracket match for which there is a match in a pattern.
	PRXPARSE Function (p. 786)	Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.
	PRXPOSN Function (p. 788)	Returns a character string that contains the value for a capture buffer.
Combinatorial	ALLCOMB Function (p. 95)	Generates all combinations of the values of n variables taken k at a time in a minimal change order.
	ALLPERM Function (p. 97)	Generates all permutations of the values of several variables in a minimal change order.
	CALL ALLCOMB Routine (p. 150)	Generates all combinations of the values of n variables taken k at a time in a minimal change order.
	CALL ALLCOMBI Routine (p. 153)	Generates all combinations of the indices of n objects taken k at a time in a minimal change order.
	CALL ALLPERM Routine (p. 156)	Generates all permutations of the values of several variables in a minimal change order.
	CALL GRAYCODE Routine (p. 168)	Generates all subsets of n items in a minimal change order.
	CALL LEXCOMB Routine (p. 177)	Generates all distinct combinations of the nonmissing values of n variables taken k at a time in lexicographic order.
	CALL LEXCOMBI Routine (p. 180)	Generates all combinations of the indices of n objects taken k at a time in lexicographic order.

Category	Language Elements	Description
	CALL LEXPERK Routine (p. 183)	Generates all distinct permutations of the nonmissing values of n variables taken k at a time in lexicographic order.
	CALL LEXPERM Routine (p. 187)	Generates all distinct permutations of the nonmissing values of several variables in lexicographic order.
	CALL RANCOMB Routine (p. 215)	Permutes the values of the arguments, and returns a random combination of k out of n values.
	CALL RANPERK Routine (p. 224)	Permutes the values of the arguments, and returns a random permutation of k out of n values.
	CALL RANPERM Routine (p. 226)	Randomly permutes the values of the arguments.
	COMB Function (p. 310)	Computes the number of combinations of n elements taken r at a time.
	GRAYCODE Function (p. 523)	Generates all subsets of n items in a minimal change order.
	LCOMB Function (p. 615)	Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of n objects taken r at a time.
	LEXCOMB Function (p. 622)	Generates all distinct combinations of the non-missing values of n variables taken k at a time in lexicographic order.
	LEXCOMBI Function (p. 625)	Generates all combinations of the indices of n objects taken k at a time in lexicographic order.
	LEXPERK Function (p. 627)	Generates all distinct permutations of the non-missing values of n variables taken k at a time in lexicographic order.
	LEXPERM Function (p. 629)	Generates all distinct permutations of the non-missing values of several variables in lexicographic order.
	LFACT Function (p. 632)	Computes the logarithm of the FACT (factorial) function.
	LPERM Function (p. 647)	Computes the logarithm of the PERM function, which is the logarithm of the number of permutations of n objects, with the option of including r number of elements.
	PERM Function (p. 743)	Computes the number of permutations of n items that are taken r at a time.
Date and Time	CALL IS8601_CONVERT Routine (p. 172)	Converts an ISO 8601 interval to datetime and duration values, or converts datetime and duration values to an ISO 8601 interval.
	DATDIF Function (p. 355)	Returns the number of days between two dates after computing the difference between the dates according to specified day count conventions.
	DATE Function (p. 358)	Returns the current date as a SAS date value.

Category	Language Elements	Description
	DATEJUL Function (p. 358)	Converts a Julian date to a SAS date value.
	DATEPART Function (p. 359)	Extracts the date from a SAS datetime value.
	DATETIME Function (p. 360)	Returns the current date and time of day as a SAS datetime value.
	DAY Function (p. 360)	Returns the day of the month from a SAS date value.
	DHMS Function (p. 373)	Returns a SAS datetime value from date, hour, minute, and second values.
	HMS Function (p. 530)	Returns a SAS time value from hour, minute, and second values.
	HOLIDAY Function (p. 531)	Returns a SAS date value of a specified holiday for a specified year.
	HOUR Function (p. 534)	Returns the hour from a SAS time or datetime value.
	INTCINDEX Function (p. 556)	Returns the cycle index when a date, time, or datetime interval and value are specified.
	INTCK Function (p. 559)	Returns the number of interval boundaries of a given kind that lie between two dates, times, or datetime values.
	INTCYCLE Function (p. 565)	Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.
	INTFIT Function (p. 567)	Returns a time interval that is aligned between two dates.
	INTFMT Function (p. 571)	Returns a recommended SAS format when a date, time, or datetime interval is specified.
	INTGET Function (p. 573)	Returns a time interval based on three date or datetime values.
	INTINDEX Function (p. 574)	Returns the seasonal index when a date, time, or datetime interval and value are specified.
	INTNX Function (p. 580)	Increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value.
	INTSEAS Function (p. 589)	Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.
	INTSHIFT Function (p. 592)	Returns the shift interval that corresponds to the base interval.
	INTTEST Function (p. 594)	Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.
	JULDATE Function (p. 602)	Returns the Julian date from a SAS date value.
	JULDATE7 Function (p. 603)	Returns a seven-digit Julian date from a SAS date value.
	MDY Function (p. 657)	Returns a SAS date value from month, day, and year values.

Category	Language Elements	Description
	MINUTE Function (p. 661)	Returns the minute from a SAS time or datetime value.
	MONTH Function (p. 669)	Returns the month from a SAS date value.
	NWKDOM Function (p. 713)	Returns the date for the nth occurrence of a weekday for the specified month and year.
	QTR Function (p. 798)	Returns the quarter of the year from a SAS date value.
	SECOND Function (p. 859)	Returns the second from a SAS time or datetime value.
	TIME Function (p. 912)	Returns the current time of day as a numeric SAS time value.
	TIMEPART Function (p. 912)	Extracts a time value from a SAS datetime value.
	TODAY Function (p. 917)	Returns the current date as a numeric SAS date value.
	WEEK Function (p. 976)	Returns the week-number value.
	WEEKDAY Function (p. 981)	From a SAS date value, returns an integer that corresponds to the day of the week.
	YEAR Function (p. 984)	Returns the year from a SAS date value.
	YRDIF Function (p. 986)	Returns the difference in years between two dates according to specified day count conventions; returns a person's age.
	YYQ Function (p. 989)	Returns a SAS date value from year and quarter year values.
Descriptive Statistics	CMISS Function (p. 303)	Counts the number of missing arguments.
	CSS Function (p. 346)	Returns the corrected sum of squares.
	CV Function (p. 350)	Returns the coefficient of variation.
	EUCLID Function (p. 395)	Returns the Euclidean norm of the nonmissing arguments.
	GEOMEAN Function (p. 511)	Returns the geometric mean.
	GEOMEANZ Function (p. 513)	Returns the geometric mean, using zero fuzzing.
	HARMEAN Function (p. 526)	Returns the harmonic mean.
	HARMEANZ Function (p. 527)	Returns the harmonic mean, using zero fuzzing.
	IQR Function (p. 599)	Returns the interquartile range.
	KURTOSIS Function (p. 604)	Returns the kurtosis.
	LARGEST Function (p. 611)	Returns the kth largest non-missing value.

Category	Language Elements	Description
	LPNORM Function (p. 648)	Returns the Lp norm of the second argument and subsequent non-missing arguments.
	MAD Function (p. 650)	Returns the median absolute deviation from the median.
	MAX Function (p. 655)	Returns the largest value.
	MEAN Function (p. 658)	Returns the arithmetic mean (average).
	MEDIAN Function (p. 659)	Returns the median value.
	MIN Function (p. 660)	Returns the smallest value.
	MISSING Function (p. 662)	Returns a numeric result that indicates whether the argument contains a missing value.
	N Function (p. 679)	Returns the number of nonmissing numeric values.
	NMISS Function (p. 683)	Returns the number of missing numeric values.
	ORDINAL Function (p. 718)	Returns the kth smallest of the missing and nonmissing values.
	PCTL Function (p. 720)	Returns the percentile that corresponds to the percentage.
	RANGE Function (p. 820)	Returns the range of the nonmissing values.
	RMS Function (p. 833)	Returns the root mean square of the nonmissing arguments.
	SKEWNESS Function (p. 862)	Returns the skewness of the nonmissing arguments.
	SMALLEST Function (p. 864)	Returns the kth smallest nonmissing value.
	STD Function (p. 883)	Returns the standard deviation of the nonmissing arguments.
	STDERR Function (p. 884)	Returns the standard error of the mean of the nonmissing arguments.
	SUM Function (p. 898)	Returns the sum of the nonmissing arguments.
	SUMABS Function (p. 899)	Returns the sum of the absolute values of the non-missing arguments.
	USS Function (p. 932)	Returns the uncorrected sum of squares of the nonmissing arguments.
	VAR Function (p. 933)	Returns the variance of the nonmissing arguments.
Distance	GEODIST Function (p. 509)	Returns the geodetic distance between two latitude and longitude coordinates.
	ZIPCITYDISTANCE Function (p. 992)	Returns the geodetic distance between two ZIP code locations.

Category	Language Elements	Description
External Files	DCLOSE Function (p. 361)	Closes a directory that was opened by the DOPEN function.
	DCREATE Function (p. 362)	Returns the complete pathname of a new, external directory.
	DINFO Function (p. 378)	Returns information about a directory.
	DNUM Function (p. 381)	Returns the number of members in a directory.
	DOPEN Function (p. 382)	Opens a directory, and returns a directory identifier value.
	DOPTNAME Function (p. 384)	Returns directory attribute information.
	DOPTNUM Function (p. 385)	Returns the number of information items that are available for a directory.
	DREAD Function (p. 386)	Returns the name of a directory member.
	DROPNOTE Function (p. 387)	Deletes a note marker from a SAS data set or an external file.
	FAPPEND Function (p. 400)	Appends the current record to the end of an external file.
	FCLOSE Function (p. 402)	Closes an external file, directory, or directory member.
	FCOL Function (p. 403)	Returns the current column position in the File Data Buffer (FDB).
	FDELETE Function (p. 404)	Deletes an external file or an empty directory.
	FEXIST Function (p. 408)	Verifies the existence of an external file that is associated with a fileref.
	FGET Function (p. 409)	Copies data from the File Data Buffer (FDB) into a variable.
	FILEEXIST Function (p. 410)	Verifies the existence of an external file by its physical name.
	FILENAME Function (p. 411)	Assigns or deassigns a fileref to an external file, directory, or output device.
	FILEREF Function (p. 414)	Verifies whether a fileref has been assigned for the current SAS session.
	FINFO Function (p. 473)	Returns the value of a file information item.
	FNOTE Function (p. 484)	Identifies the last record that was read, and returns a value that the FPOINT function can use.
	FOPEN Function (p. 485)	Opens an external file and returns a file identifier value.
	FOPTNAME Function (p. 488)	Returns the name of an item of information about a file.
	FOPTNUM Function (p. 489)	Returns the number of information items that are available for an external file.

Category	Language Elements	Description
	FPOINT Function (p. 490)	Positions the read pointer on the next record to be read.
	FPOS Function (p. 492)	Sets the position of the column pointer in the File Data Buffer (FDB).
	FPUT Function (p. 494)	Moves data to the File Data Buffer (FDB) of an external file, starting at the FDB's current column position.
	FREAD Function (p. 495)	Reads a record from an external file into the File Data Buffer (FDB).
	FREWIND Function (p. 496)	Positions the file pointer to the start of the file.
	FRLEN Function (p. 498)	Returns the size of the last record that was read, or, if the file is opened for output, returns the current record size.
	FSEP Function (p. 499)	Sets the token delimiters for the FGET function.
	FWRITE Function (p. 501)	Writes a record to an external file.
	MOPEN Function (p. 670)	Opens a file by directory ID and member name, and returns either the file identifier or a 0.
	PATHNAME Function (p. 719)	Returns the physical name of an external file or a SAS library, or returns a blank.
	RENAME Function (p. 827)	Renames a member of a SAS library, an entry in a SAS catalog, an external file, or a directory.
	SYSMSG Function (p. 905)	Returns error or warning message text from processing the last data set or external file function.
	SYSRC Function (p. 909)	Returns a system error number.
External Routines	CALL MODULE Routine (p. 192)	Calls an external routine without any return code.
	MODULEC Function (p. 666)	Calls an external routine and returns a character value.
	MODULEN Function (p. 667)	Calls an external routine and returns a numeric value.
Financial	BLACKCLPRC Function (p. 138)	Calculates call prices for European options on futures, based on the Black model.
	BLACKPTPRC Function (p. 140)	Calculates put prices for European options on futures, based on the Black model.
	BLKSHCLPRC Function (p. 142)	Calculates call prices for European options on stocks, based on the Black-Scholes model.
	BLKSHPTPRC Function (p. 144)	Calculates put prices for European options on stocks, based on the Black-Scholes model.

Category	Language Elements	Description
	COMPOUND Function (p. 325)	Returns compound interest parameters.
	CONVX Function (p. 334)	Returns the convexity for an enumerated cash flow.
	CONVXP Function (p. 335)	Returns the convexity for a periodic cash flow stream, such as a bond.
	CUMIPMT Function (p. 347)	Returns the cumulative interest paid on a loan between the start and end period.
	CUMPRINC Function (p. 348)	Returns the cumulative principal paid on a loan between the start and end period.
	DACCDB Function (p. 350)	Returns the accumulated declining balance depreciation.
	DACCDBSL Function (p. 351)	Returns the accumulated declining balance with conversion to a straight-line depreciation.
	DACCSL Function (p. 352)	Returns the accumulated straight-line depreciation.
	DACCSYD Function (p. 353)	Returns the accumulated sum-of-years-digits depreciation.
	DACCTAB Function (p. 354)	Returns the accumulated depreciation from specified tables.
	DEPDB Function (p. 363)	Returns the declining balance depreciation.
	DEPDBSL Function (p. 364)	Returns the declining balance with conversion to a straight-line depreciation.
	DEPSL Function (p. 365)	Returns the straight-line depreciation.
	DEPSYD Function (p. 366)	Returns the sum-of-years-digits depreciation.
	DEPTAB Function (p. 367)	Returns the depreciation from specified tables.
	DUR Function (p. 389)	Returns the modified duration for an enumerated cash flow.
	DURP Function (p. 390)	Returns the modified duration for a periodic cash flow stream, such as a bond.
	EFFRATE Function (p. 391)	Returns the effective annual interest rate.
	FINANCE Function (p. 416)	Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.
	GARKHCLPRC Function (p. 504)	Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.
	GARKHPTPRC Function (p. 506)	Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

Category	Language Elements	Description
	INTRR Function (p. 587)	Returns the internal rate of return as a fraction.
	IPMT Function (p. 598)	Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.
	IRR Function (p. 600)	Returns the internal rate of return as a percentage.
	MARGRCLPRC Function (p. 650)	Calculates call prices for European options on stocks, based on the Margrabe model.
	MARGRPTPRC Function (p. 653)	Calculates put prices for European options on stocks, based on the Margrabe model.
	MORT Function (p. 672)	Returns amortization parameters.
	NETPV Function (p. 680)	Returns the net present value as a percent.
	NOMRATE Function (p. 684)	Returns the nominal annual interest rate.
	NPV Function (p. 710)	Returns the net present value with the rate expressed as a percentage.
	PMT Function (p. 745)	Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.
	PPMT Function (p. 748)	Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.
	PVP Function (p. 797)	Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.
	SAVING Function (p. 845)	Returns the future value of a periodic saving.
	SAVINGS Function (p. 846)	Returns the balance of a periodic savings by using variable interest rates.
	TIMEVALUE Function (p. 913)	Returns the equivalent of a reference amount at a base date by using variable interest rates.
	YIELDP Function (p. 985)	Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.
Hyperbolic	ARCOSH Function (p. 123)	Returns the inverse hyperbolic cosine.
	ARSINH Function (p. 125)	Returns the inverse hyperbolic sine.
	ARTANH Function (p. 126)	Returns the inverse hyperbolic tangent.
	COSH Function (p. 337)	Returns the hyperbolic cosine.
	SINH Function (p. 861)	Returns the hyperbolic sine.
	TANH Function (p. 911)	Returns the hyperbolic tangent.

Category	Language Elements	Description
Macro	CALL EXECUTE Routine (p. 168)	Resolves the argument, and issues the resolved value for execution at the next step boundary.
	CALL SYMPUT Routine (p. 256)	Assigns DATA step information to a macro variable.
	CALL SYMPUTX Routine (p. 257)	Assigns a value to a macro variable, and removes both leading and trailing blanks.
	RESOLVE Function (p. 829)	Returns the resolved value of the argument after it has been processed by the macro facility.
	SYMEXIST Function (p. 900)	Returns an indication of the existence of a macro variable.
	SYMGET Function (p. 901)	Returns the value of a macro variable during DATA step execution.
	SYMGLOBL Function (p. 901)	Returns an indication of whether a macro variable is in global scope to the DATA step during DATA step execution.
	SYMLOCAL Function (p. 902)	Returns an indication of whether a macro variable is in local scope to the DATA step during DATA step execution.
Mathematical	ABS Function (p. 92)	Returns the absolute value.
	AIRY Function (p. 94)	Returns the value of the Airy function.
	BETA Function (p. 136)	Returns the value of the beta function.
	CALL LOGISTIC Routine (p. 190)	Applies the logistic function to each argument.
	CALL SOFTMAX Routine (p. 248)	Returns the softmax value.
	CALL STDIZE Routine (p. 251)	Standardizes the values of one or more variables.
	CALL TANH Routine (p. 259)	Returns the hyperbolic tangent.
	CNONCT Function (p. 304)	Returns the noncentrality parameter from a chi-square distribution.
	COALESCE Function (p. 306)	Returns the first non-missing value from a list of numeric arguments.
	COMPFUZZ Function (p. 315)	Performs a fuzzy comparison of two numeric values.
	CONSTANT Function (p. 330)	Computes machine and mathematical constants.
	DAIRY Function (p. 355)	Returns the derivative of the AIRY function.
	DEVIANCE Function (p. 370)	Returns the deviance based on a probability distribution.

Category	Language Elements	Description
	DIGAMMA Function (p. 376)	Returns the value of the digamma function.
	ERF Function (p. 393)	Returns the value of the (normal) error function.
	ERFC Function (p. 394)	Returns the value of the complementary (normal) error function.
	EXP Function (p. 399)	Returns the value of the exponential function.
	FACT Function (p. 399)	Computes a factorial.
	FNONCT Function (p. 482)	Returns the value of the noncentrality parameter of an F distribution.
	GAMMA Function (p. 503)	Returns the value of the gamma function.
	GCD Function (p. 508)	Returns the greatest common divisor for one or more integers.
	IBESSEL Function (p. 538)	Returns the value of the modified Bessel function.
	JBESSEL Function (p. 601)	Returns the value of the Bessel function.
	LCM Function (p. 614)	Returns the least common multiple.
	LGAMMA Function (p. 633)	Returns the natural logarithm of the Gamma function.
	LOG Function (p. 637)	Returns the natural (base e) logarithm.
	LOG1PX Function (p. 637)	Returns the log of 1 plus the argument.
	LOG10 Function (p. 638)	Returns the logarithm to the base 10.
	LOG2 Function (p. 639)	Returns the logarithm to the base 2.
	LOGBETA Function (p. 640)	Returns the logarithm of the beta function.
	MOD Function (p. 663)	Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.
	MODZ Function (p. 667)	Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.
	MSPLINT Function (p. 674)	Returns the ordinate of a monotonicity-preserving interpolating spline.
	SIGN Function (p. 860)	Returns the sign of a value.
	SQRT Function (p. 880)	Returns the square root of a value.
	TNONCT Function (p. 915)	Returns the value of the noncentrality parameter from the Student's t distribution.

Category	Language Elements	Description
	TRIGAMMA Function (p. 924)	Returns the value of the trigamma function.
Numeric	IFN Function (p. 541)	Returns a numeric value based on whether an expression is true, false, or missing.
	MODEXIST Function (p. 665)	Determines whether a software image exists in the version of SAS that you have installed.
Probability	CDF Function (p. 277)	Returns a value from a cumulative probability distribution.
	LOGCDF Function (p. 640)	Returns the logarithm of a left cumulative distribution function.
	LOGPDF Function (p. 642)	Returns the logarithm of a probability density (mass) function.
	LOGSDF Function (p. 644)	Returns the logarithm of a survival function.
	PDF Function (p. 722)	Returns a value from a probability density (mass) distribution.
	POISSON Function (p. 747)	Returns the probability from a Poisson distribution.
	PROBBETA Function (p. 749)	Returns the probability from a beta distribution.
	PROBBNML Function (p. 750)	Returns the probability from a binomial distribution.
	PROBBNRM Function (p. 751)	Returns a probability from a bivariate normal distribution.
	PROBCHI Function (p. 752)	Returns the probability from a chi-square distribution.
	PROBF Function (p. 753)	Returns the probability from an F distribution.
	PROBGAM Function (p. 754)	Returns the probability from a gamma distribution.
	PROBHYPR Function (p. 755)	Returns the probability from a hypergeometric distribution.
	PROBMC Function (p. 758)	Returns a probability or a quantile from various distributions for multiple comparisons of means.
	PROBNEGB Function (p. 770)	Returns the probability from a negative binomial distribution.
	PROBNORM Function (p. 771)	Returns the probability from the standard normal distribution.
	PROBT Function (p. 772)	Returns the probability from a t distribution.
	SDF Function (p. 856)	Returns a survival function.
Quantile	BETAINV Function (p. 137)	Returns a quantile from the beta distribution.
	CINV Function (p. 301)	Returns a quantile from the chi-square distribution.
	FINV Function (p. 474)	Returns a quantile from the F distribution.

Category	Language Elements	Description
	GAMINV Function (p. 502)	Returns a quantile from the gamma distribution.
	PROBIT Function (p. 757)	Returns a quantile from the standard normal distribution.
	QUANTILE Function (p. 799)	Returns the quantile from a distribution when you specify the left probability (CDF).
	SQUANTILE Function (p. 881)	Returns the quantile from a distribution when you specify the right probability (SDF).
	TINV Function (p. 914)	Returns a quantile from the t distribution.
Random Number	CALL RANBIN Routine (p. 210)	Returns a random variate from a binomial distribution.
	CALL RANCAU Routine (p. 212)	Returns a random variate from a Cauchy distribution.
	CALL RANEXP Routine (p. 217)	Returns a random variate from an exponential distribution.
	CALL RANGAM Routine (p. 219)	Returns a random variate from a gamma distribution.
	CALL RANNOR Routine (p. 222)	Returns a random variate from a normal distribution.
	CALL RANPOI Routine (p. 228)	Returns a random variate from a Poisson distribution.
	CALL RANTBL Routine (p. 230)	Returns a random variate from a tabled probability distribution.
	CALL RANTRI Routine (p. 233)	Returns a random variate from a triangular distribution.
	CALL RANUNI Routine (p. 235)	Returns a random variate from a uniform distribution.
	CALL STREAMINIT Routine (p. 254)	Specifies a seed value to use for subsequent random number generation by the RAND function.
	NORMAL Function (p. 685)	Returns a random variate from a normal, or Gaussian, distribution.
	RANBIN Function (p. 804)	Returns a random variate from a binomial distribution.
	RANCAU Function (p. 805)	Returns a random variate from a Cauchy distribution.
	RAND Function (p. 806)	Generates random numbers from a distribution that you specify.
	RANEXP Function (p. 817)	Returns a random variate from an exponential distribution.
	RANGAM Function (p. 818)	Returns a random variate from a gamma distribution.

Category	Language Elements	Description
	RANNOR Function (p. 821)	Returns a random variate from a normal distribution.
	RANPOI Function (p. 822)	Returns a random variate from a Poisson distribution.
	RANTBL Function (p. 823)	Returns a random variate from a tabled probability distribution.
	RANTRI Function (p. 825)	Returns a random variate from a triangular distribution.
	RANUNI Function (p. 826)	Returns a random variate from a uniform distribution.
	UNIFORM Function (p. 928)	Returns a random variate from a uniform distribution.
SAS File I/O	ATTRC Function (p. 129)	Returns the value of a character attribute for a SAS data set.
	ATTRN Function (p. 131)	Returns the value of a numeric attribute for a SAS data set.
	CEXIST Function (p. 297)	Verifies the existence of a SAS catalog or SAS catalog entry.
	CLOSE Function (p. 303)	Closes a SAS data set.
	CUROBS Function (p. 349)	Returns the observation number of the current observation.
	DROPNOTE Function (p. 387)	Deletes a note marker from a SAS data set or an external file.
	DSNAME Function (p. 388)	Returns the SAS data set name that is associated with a data set identifier.
	ENVLEN Function (p. 392)	Returns the length of an environment variable.
	EXIST Function (p. 396)	Verifies the existence of a SAS library member.
	FETCH Function (p. 405)	Reads the next non-deleted observation from a SAS data set into the Data Set Data Vector (DDV).
	FETCHOBS Function (p. 406)	Reads a specified observation from a SAS data set into the Data Set Data Vector (DDV).
	GETVARC Function (p. 521)	Returns the value of a SAS data set character variable.
	GETVARN Function (p. 522)	Returns the value of a SAS data set numeric variable.
	IORCMSG Function (p. 597)	Returns a formatted error message for _IORC_.
	LIBNAME Function (p. 633)	Assigns or deassigns a libref for a SAS library.
	LIBREF Function (p. 636)	Verifies that a libref has been assigned.
	NOTE Function (p. 692)	Returns an observation ID for the current observation of a SAS data set.
	OPEN Function (p. 716)	Opens a SAS data set.

Category	Language Elements	Description
	PATHNAME Function (p. 719)	Returns the physical name of an external file or a SAS library, or returns a blank.
	POINT Function (p. 746)	Locates an observation that is identified by the NOTE function.
	RENAME Function (p. 827)	Renames a member of a SAS library, an entry in a SAS catalog, an external file, or a directory.
	REWIND Function (p. 831)	Positions the data set pointer at the beginning of a SAS data set.
	SYSEXIST Function (p. 903)	Returns a value that indicates whether an operating-environment variable exists in your environment.
	SYSMSG Function (p. 905)	Returns error or warning message text from processing the last data set or external file function.
	SYSRC Function (p. 909)	Returns a system error number.
	VARFMT Function (p. 934)	Returns the format that is assigned to a SAS data set variable.
	VARINFMT Function (p. 935)	Returns the informat that is assigned to a SAS data set variable.
	VARLABEL Function (p. 936)	Returns the label that is assigned to a SAS data set variable.
	VARLEN Function (p. 937)	Returns the length of a SAS data set variable.
	VARNAME Function (p. 939)	Returns the name of a SAS data set variable.
	VARNUM Function (p. 940)	Returns the number of a variable's position in a SAS data set.
	VARTYPE Function (p. 943)	Returns the data type of a SAS data set variable.
Search	WHICHC Function (p. 982)	Searches for a character value that is equal to the first argument, and returns the index of the first matching value.
	WHICHN Function (p. 983)	Searches for a numeric value that is equal to the first argument, and returns the index of the first matching value.
Sort	CALL SORTC Routine (p. 249)	Sorts the values of character arguments.
	CALL SORTN Routine (p. 250)	Sorts the values of numeric arguments.
Special	ADDR Function (p. 92)	Returns the memory address of a variable on a 32-bit platform.
	ADDRLONG Function (p. 93)	Returns the memory address of a variable on 32-bit and 64-bit platforms.
	CALL POKE Routine (p. 195)	Writes a value directly into memory on a 32-bit platform.
	CALL POKELONG Routine (p. 197)	Writes a value directly into memory on 32-bit and 64-bit platforms.

Category	Language Elements	Description
	CALL SLEEP Routine (p. 247)	For a specified period of time, suspends the execution of a program that invokes this CALL routine.
	CALL SYSTEM Routine (p. 259)	Submits an operating environment command for execution.
	DIF Function (p. 374)	Returns differences between an argument and its nth lag.
	GETOPTION Function (p. 514)	Returns the value of a SAS system or graphics option.
	INPUT Function (p. 550)	Returns the value that is produced when SAS converts an expression using the specified informat.
	INPUTC Function (p. 552)	Enables you to specify a character informat at run time.
	INPUTN Function (p. 554)	Enables you to specify a numeric informat at run time.
	LAG Function (p. 605)	Returns values from a queue.
	PEEK Function (p. 738)	Stores the contents of a memory address in a numeric variable on a 32-bit platform.
	PEEKC Function (p. 739)	Stores the contents of a memory address in a character variable on a 32-bit platform.
	PEEKCLONG Function (p. 741)	Stores the contents of a memory address in a character variable on 32-bit and 64-bit platforms.
	PEEKLONG Function (p. 742)	Stores the contents of a memory address in a numeric variable on 32-bit and 64-bit platforms.
	PTRLONGADD Function (p. 791)	Returns the pointer address as a character variable on 32-bit and 64-bit platforms.
	PUT Function (p. 791)	Returns a value using a specified format.
	PUTC Function (p. 793)	Enables you to specify a character format at run time.
	PUTN Function (p. 795)	Enables you to specify a numeric format at run time.
	SLEEP Function (p. 863)	For a specified period of time, suspends the execution of a program that invokes this function.
	SYSEXIST Function (p. 903)	Returns a value that indicates whether an operating-environment variable exists in your environment.
	SYSGET Function (p. 904)	Returns the value of the specified operating environment variable.
	SYSPARM Function (p. 906)	Returns the system parameter string.
	SYSPROCESSID Function (p. 906)	Returns the process ID of the current process.

Category	Language Elements	Description
	SYSPROCESSNAME Function (p. 907)	Returns the process name that is associated with a given process ID, or returns the name of the current process.
	SYSPROD Function (p. 908)	Determines whether a product is licensed.
	SYSTEM Function (p. 910)	Issues an operating environment command during a SAS session, and returns the system return code.
	UUIDGEN Function (p. 932)	Returns the short or binary form of a Universal Unique Identifier (UUID).
State and ZIP Code	FIPNAME Function (p. 475)	Converts two-digit FIPS codes to uppercase state names.
	FIPNAMEL Function (p. 476)	Converts two-digit FIPS codes to mixed case state names.
	FIPSTATE Function (p. 477)	Converts two-digit FIPS codes to two-character state postal codes.
	STFIPS Function (p. 884)	Converts state postal codes to FIPS state codes.
	STNAME Function (p. 886)	Converts state postal codes to uppercase state names.
	STNAMEL Function (p. 887)	Converts state postal codes to mixed case state names.
	ZIPCITY Function (p. 990)	Returns a city name and the two-character postal code that corresponds to a ZIP code.
	ZIPCITYDISTANCE Function (p. 992)	Returns the geodetic distance between two ZIP code locations.
	ZIPFIPS Function (p. 993)	Converts ZIP codes to two-digit FIPS codes.
	ZIPNAME Function (p. 994)	Converts ZIP codes to uppercase state names.
	ZIPNAMEL Function (p. 996)	Converts ZIP codes to mixed case state names.
	ZIPSTATE Function (p. 998)	Converts ZIP codes to two-character state postal codes.
Trigonometric	ARCOS Function (p. 123)	Returns the arccosine.
	ARSIN Function (p. 124)	Returns the arcsine.
	ATAN Function (p. 127)	Returns the arc tangent.
	ATAN2 Function (p. 128)	Returns the arc tangent of the ratio of two numeric variables.
	COS Function (p. 336)	Returns the cosine.
	SIN Function (p. 860)	Returns the sine.
	TAN Function (p. 911)	Returns the tangent.
Truncation	CEIL Function (p. 294)	Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results.

Category	Language Elements	Description
	CEILZ Function (p. 296)	Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.
	FLOOR Function (p. 480)	Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results.
	FLOORZ Function (p. 481)	Returns the largest integer that is less than or equal to the argument, using zero fuzzing.
	FUZZ Function (p. 500)	Returns the nearest integer if the argument is within 1E−12 of that integer.
	INT Function (p. 555)	Returns the integer value, fuzzed to avoid unexpected floating-point results.
	INTZ Function (p. 596)	Returns the integer portion of the argument, using zero fuzzing.
	ROUND Function (p. 833)	Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.
	ROUNDE Function (p. 840)	Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.
	ROUNDZ Function (p. 843)	Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.
	TRUNC Function (p. 927)	Truncates a numeric value to a specified number of bytes.
Variable Control	CALL LABEL Routine (p. 176)	Assigns a variable label to a specified character variable.
	CALL SET Routine (p. 246)	Links SAS data set variables to DATA step or macro variables that have the same name and data type.
	CALL VNAME Routine (p. 260)	Assigns a variable name as the value of a specified variable.
Variable Information	CALL VNEXT Routine (p. 261)	Returns the name, type, and length of a variable that is used in a DATA step.
	VARRAY Function (p. 941)	Returns a value that indicates whether the specified name is an array.
	VARRAYX Function (p. 942)	Returns a value that indicates whether the value of the specified argument is an array.
	VFORMAT Function (p. 945)	Returns the format that is associated with the specified variable.
	VFORMATD Function (p. 946)	Returns the decimal value of the format that is associated with the specified variable.

Category	Language Elements	Description
	VFORMATDX Function (p. 947)	Returns the decimal value of the format that is associated with the value of the specified argument.
	VFORMATN Function (p. 948)	Returns the format name that is associated with the specified variable.
	VFORMATNX Function (p. 949)	Returns the format name that is associated with the value of the specified argument.
	VFORMATW Function (p. 951)	Returns the format width that is associated with the specified variable.
	VFORMATWX Function (p. 952)	Returns the format width that is associated with the value of the specified argument.
	VFORMATX Function (p. 953)	Returns the format that is associated with the value of the specified argument.
	VINARRAY Function (p. 954)	Returns a value that indicates whether the specified variable is a member of an array.
	VINARRAYX Function (p. 955)	Returns a value that indicates whether the value of the specified argument is a member of an array.
	VINFORMAT Function (p. 956)	Returns the informat that is associated with the specified variable.
	VINFORMATD Function (p. 957)	Returns the decimal value of the informat that is associated with the specified variable.
	VINFORMATDX Function (p. 958)	Returns the decimal value of the informat that is associated with the value of the specified variable.
	VINFORMATN Function (p. 959)	Returns the informat name that is associated with the specified variable.
	VINFORMATNX Function (p. 960)	Returns the informat name that is associated with the value of the specified argument.
	VINFORMATW Function (p. 961)	Returns the informat width that is associated with the specified variable.
	VINFORMATWX Function (p. 962)	Returns the informat width that is associated with the value of the specified argument.
	VINFORMATX Function (p. 963)	Returns the informat that is associated with the value of the specified argument.
	VLABEL Function (p. 965)	Returns the label that is associated with the specified variable.
	VLABELX Function (p. 966)	Returns the label that is associated with the value of the specified argument.
	VLENGTH Function (p. 967)	Returns the compile-time (allocated) size of the specified variable.

Category	Language Elements	Description
	VLENGTHX Function (p. 968)	Returns the compile-time (allocated) size for the variable that has a name that is the same as the value of the argument.
	VNAME Function (p. 969)	Returns the name of the specified variable.
	VNAMEX Function (p. 970)	Validates the value of the specified argument as a variable name.
	VTYPE Function (p. 971)	Returns the type (character or numeric) of the specified variable.
	VTYPEX Function (p. 972)	Returns the type (character or numeric) for the value of the specified argument.
	VVALUE Function (p. 973)	Returns the formatted value that is associated with the variable that you specify.
	VVALUEX Function (p. 975)	Returns the formatted value that is associated with the argument that you specify.
Web Service	SOAPWEB Function (p. 865)	Calls a Web service by using basic Web authentication; credentials are provided in the arguments.
	SOAPWEBMETA Function (p. 867)	Calls a Web service by using basic Web authentication; credentials for the authentication domain are retrieved from metadata.
	SOAPWIPSERVICE Function (p. 869)	Calls a SAS registered Web service by using WS-Security authentication; credentials are provided in the arguments.
	SOAPWIPSRS Function (p. 871)	Calls a SAS registered Web service by using WS-Security authentication; credentials are provided in the arguments.
	SOAPWS Function (p. 873)	Calls a Web service by using WS-Security authentication; credentials are provided in the arguments.
	SOAPWSMETA Function (p. 875)	Calls a Web service by using WS-Security authentication; credentials for the provided authentication domain are retrieved from metadata.
Web Tools	HTMLDECODE Function (p. 535)	Decodes a string that contains HTML numeric character references or HTML character entity references, and returns the decoded string.
	HTMLENCODE Function (p. 536)	Encodes characters using HTML character entity references, and returns the encoded string.
	URLDECODE Function (p. 929)	Returns a string that was decoded using the URL escape syntax.
	URLENCODE Function (p. 930)	Returns a string that was encoded using the URL escape syntax.

Dictionary

ABS Function

Returns the absolute value.

Category: Mathematical

Syntax

ABS (*argument*)

Required Argument

argument
specifies a 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

The following SAS statements produce these results.

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

ADDR Function

Returns the memory address of a variable on a 32-bit platform.

Category: Special

Restriction: Use on 32-bit platforms only.

Syntax

ADDR(*variable*)

Required Argument

variable
specifies a variable name.

Details

The value that is returned is numeric. Because the storage location of a variable can vary from one execution to the next, the value that is returned by ADDR can vary. The ADDR function is used mostly in combination with the PEEK and PEEKC functions and the CALL POKE routine.

You cannot use the ADDR function on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use ADDR, change the applications and use ADDRLONG instead. You can use ADDRLONG on both 32-bit and 64-bit platforms.

Comparisons

The ADDR function returns the memory address of a variable on a 32-bit platform. ADDRLONG returns the memory address of a variable on 32-bit and 64-bit platforms.

Note: SAS recommends that you use ADDRLONG instead of ADDR because ADDRLONG can be used on both 32-bit and 64-bit platforms.

Example

The following example returns the address at which the variable FIRST is stored:

```
data numlist;
   first=3;
   x=addr(first);
run;
```

See Also

Functions:

- “PEEK Function” on page 738
- “PEEKC Function” on page 739
- “ADDRLONG Function” on page 93

CALL Routines:

- “CALL POKE Routine” on page 195

ADDRLONG Function

Returns the memory address of a variable on 32-bit and 64-bit platforms.

Category: Special

Syntax

ADDRLONG(*variable*)

Required Argument

variable
specifies a variable.

Details

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

Example

The following example returns the pointer address for the variable ITEM, and formats the value.

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

The following line is written to the SAS log:

```
480063B020202020
```

AIRY Function

Returns the value of the Airy function.

Category: Mathematical

Syntax

AIRY(*x*)

Required Argument

x
specifies a numeric constant, variable, or expression.

Details

The AIRY function returns the value of the Airy function. (See a list of References on page 1001.) It is the solution of the differential equation

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

with the conditions

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

and

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

Example

The following SAS statements produce these results.

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

ALLCOMB Function

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

Category: Combinatorial

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

Syntax

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

Required Arguments

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

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

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

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

Requirement Initialize these variables before executing the ALLCOMB function.

Tip After executing ALLCOMB, the first *k* variables contain the values in one combination.

Details

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

For the ALLCOMB function, the following actions occur:

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

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

Comparisons

SAS provides four functions or CALL routines for generating combinations:

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

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

Example

The following is an example of the ALLCOMB function.

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

```
run;
```

SAS writes the following output to the log:

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

See Also

CALL Routines:

- "CALL ALLCOMB Routine" on page 150

ALLPERM Function

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

Category: Combinatorial

Syntax

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

Required Arguments

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

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

Restriction Specify no more than 18 variables.

Requirement Initialize these variables before you execute the ALLPERM function.

Details

The Basics

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

Note: You can compute the number of permutations by using the PERM function. For more information, see the "PERM Function" on page 743. For the ALLPERM function, the following values are returned:

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

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

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

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

Comparisons

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

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

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

Example

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

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

SAS writes the following output to the log:

```
    1  0   ant bee cat dog
    2  3   ant bee dog cat
    3  2   ant dog bee cat
```

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

See Also

Functions and CALL Routines:

- “CALL ALLPERM Routine” on page 156
- “LEXPERM Function” on page 629
- “CALL RANPERK Routine” on page 224
- “CALL RANPERM Routine” on page 226

ANYALNUM Function

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

Category: Character

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

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 the LOCALE system options.

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

If you use only one argument, ANYALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALNUM function searches a character string for an alphanumeric character. The NOTALNUM function searches a character string for a non-alphanumeric character.

Examples

Example 1: Scanning a String from Left to Right

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

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

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=L
j=9 c=a
j=10 c=s
j=11 c=t
j=15 c=1
That's all
```

Example 2: Scanning a String from Right to Left

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

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

The following lines are written to the SAS log:

```
j=15 c=1
j=11 c=t
j=10 c=s
j=9 c=a
j=8 c=L
j=4 c=t
j=3 c=x
j=2 c=e
j=1 c=N
That's all
```

See Also

Functions:

- "NOTALNUM Function" on page 685

ANYALPHA Function

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

Category: Character

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

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 the LOCALE system options.

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

If you use only one argument, ANYALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYALPHA function searches a character string for an alphabetic character. The NOTALPHA function searches a character string for a non-alphabetic character.

Examples

Example 1: Searching a String for Alphabetic Characters

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

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

The following lines are written to the SAS log:

```
j=1 c=N
   j=2 c=e
   j=3 c=x
   j=4 c=t
   j=9 c=n
   j=16 c=E
   That's all
```

Example 2: Identifying Control Characters by Using the ANYALPHA Function

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

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

 proc print data=test;
 run;
```

See Also

Functions:

- "NOTALPHA Function" on page 687

ANYCNTRL Function

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

Category: Character

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

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 the LOCALE system options.

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

If you use only one argument, ANYCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYCNTRL returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYCNTRL function searches a character string for a control character. The NOTCNTRL function searches a character string for a character that is not a control character.

Example

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

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

```
  anycntrl=anycntrl(byte);
  if anycntrl then output;
end;

proc print data=test;
run;
```

See Also

Functions:

- "NOTCNTRL Function" on page 689

ANYDIGIT Function

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

Category: Character

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

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.

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

The following lines are written to the SAS log:

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

See Also

Functions:

- "NOTDIGIT Function" on page 690

ANYFIRST Function

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

Category: Character

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

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.

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

```
            c=substr(string,j,1);
            put +3 j= c=;
         end;
      end;
   run;
```

The following lines are written to the SAS log:

```
   j=1 c=N
   j=2 c=e
   j=3 c=x
   j=4 c=t
   j=8 c=_
   j=9 c=n
   j=10 c=_
   j=16 c=E
   That's all
```

See Also

Functions:

- "NOTFIRST Function" on page 694

ANYGRAPH Function

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

Category: Character

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

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 the LOCALE system options.

The ANYGRAPH function searches a string for the first occurrence of a graphical character. A graphical character is defined as any printable character other than white

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

If you use only one argument, ANYGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYGRAPH function searches a character string for a graphical character. The NOTGRAPH function searches a character string for a non-graphical character.

Examples

Example 1: Searching a String for Graphical Characters

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

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

The following lines are written to the SAS log:

```
   j=1 c=N
   j=2 c=e
   j=3 c=x
   j=4 c=t
   j=6 c==
   j=8 c=_
   j=9 c=n
   j=10 c=_
   j=12 c=+
   j=14 c=1
   j=15 c=2
```

```
j=16 c=E
j=17 c=3
j=18 c=;
That's all
```

Example 2: Identifying Control Characters by Using the ANYGRAPH Function

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

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

 proc print data=test;
 run;
```

See Also

Functions:

- "NOTGRAPH Function" on page 695

ANYLOWER Function

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

Category: Character

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

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.

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

The following lines are written to the SAS log:

```
   j=2 c=e
   j=3 c=x
   j=4 c=t
   j=9 c=n
   That's all
```

See Also

Functions:

- "NOTLOWER Function" on page 697

ANYNAME Function

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

Category: Character

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

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.

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

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=8 c=_
j=9 c=n
j=10 c=_
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
That's all
```

See Also

Functions:

- “NOTNAME Function” on page 699

ANYPRINT Function

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

Category: Character

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

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 the LOCALE system options.

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

If you use only one argument, ANYPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYPRINT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYPRINT function searches a character string for a printable character. The NOTPRINT function searches a character string for a non-printable character.

Examples

Example 1: Searching a String for a Printable Character

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

```
data _null_;
   string='Next = _n_ + 12E3;';
```

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

The following lines are written to the SAS log:

```
   j=1 c=N
   j=2 c=e
   j=3 c=x
   j=4 c=t
   j=5 c=
   j=6 c==
   j=7 c=
   j=8 c=_
   j=9 c=n
   j=10 c=_
   j=11 c=
   j=12 c=+
   j=13 c=
   j=14 c=1
   j=15 c=2
   j=16 c=E
   j=17 c=3
   j=18 c=;
   That's all
```

Example 2: Identifying Control Characters by Using the ANYPRINT Function

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

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

 proc print data=test;
 run;
```

See Also

Functions:

- “NOTPRINT Function” on page 701

ANYPUNCT Function

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

Category: Character

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

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 the LOCALE system options.

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

If you use only one argument, ANYPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

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

Examples

Example 1: Searching a String for Punctuation Characters

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

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

The following lines are written to the SAS log:

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

Example 2: Identifying Control Characters by Using the ANYPUNCT Function

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

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

 proc print data=test;
 run;
```

See Also

Functions:

- “NOTPUNCT Function” on page 702

ANYSPACE Function

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

Category: Character

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

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 the LOCALE system options.

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

If you use only one argument, ANYSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Examples

Example 1: Searching a String for a White-Space Character

The following example uses the ANYSPACE function to search a string for a character that is a white-space character.

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

The following lines are written to the SAS log:

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

Example 2: Identifying Control Characters by Using the ANYSPACE Function

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

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

 proc print data=test;
 run;
```

See Also

Functions:

- “NOTSPACE Function” on page 704

ANYUPPER Function

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

Category: Character

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

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 the LOCALE system options.

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

If you use only one argument, ANYUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

ANYUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The ANYUPPER function searches a character string for an uppercase letter. The NOTUPPER function searches a character string for a character that is not an uppercase letter.

Example

The following example uses the ANYUPPER function to search a string for an uppercase letter.

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

The following lines are written to the SAS log:

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

See Also

Functions:

- "NOTUPPER Function" on page 707

ANYXDIGIT Function

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

Category: Character

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

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.

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

The following lines are written to the SAS log:

```
j=2 c=e
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
That's all
```

See Also

Functions:

- "NOTXDIGIT Function" on page 708

ARCOS Function

Returns the arccosine.

Category: Trigonometric

Syntax

ARCOS (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range between −1 and 1

Details

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

Example

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.

Category: Hyperbolic

Syntax

ARCOSH(*x*)

Required Argument

x

specifies a numeric constant, variable, or expression.

Range $x >= 1$

Details

The ARCOSH function computes the inverse hyperbolic cosine. The ARCOSH function is mathematically 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.

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

SAS writes the following output to the log:

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

See Also

Functions:

- “COSH Function” on page 337
- “SINH Function” on page 861
- “TANH Function” on page 911
- “ARSINH Function” on page 125
- “ARTANH Function” on page 126

ARSIN Function

Returns the arcsine.

Category: Trigonometric

Syntax

ARSIN (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range between −1 and 1

Details

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

Example

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.

Category: Hyperbolic

Syntax

ARSINH(*x*)

Required Argument

x
specifies a numeric constant, variable, or expression.

Range $-\infty < x < \infty$

Details

The ARSINH function computes the inverse hyperbolic sine. The ARSINH function is mathematically 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.

```
data _null_;
   x=arsinh(5);
   x1=arsinh(-5);
   put x=;
   put x1=;
run;
```

SAS writes the following output to the log:

```
x=2.3124383413
x1=-2.312438341
```

See Also

Functions:

- "COSH Function" on page 337
- "SINH Function" on page 861
- "TANH Function" on page 911
- "ARCOSH Function" on page 123
- "ARTANH Function" on page 126

ARTANH Function

Returns the inverse hyperbolic tangent.

Category: Hyperbolic

Syntax

ARTANH(*x*)

Required Argument

x

specifies a numeric constant, variable, or expression.

Range $-1 < x < 1$

Details

The ARTANH function computes the inverse hyperbolic tangent. The ARTANH function is mathematically 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.

```
data _null_;
   x=artanh(0.5);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=0.5493061443
```

See Also

Functions:

- “COSH Function” on page 337
- “SINH Function” on page 861
- “TANH Function” on page 911
- “ARCOSH Function” on page 123
- “ARSINH Function” on page 125

ATAN Function

Returns the arc tangent.

Category: Trigonometric

Syntax

ATAN (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The ATAN function returns the 2-quadrant arc tangent (inverse tangent) of the argument. The value that is returned is the angle (in radians) whose tangent is *x* and whose value ranges from $-\pi/2$ to $\pi/2$. If the argument is missing, 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
`x=atan(0);`	`0`
`x=atan(1);`	`0.7853981634`
`x=atan(-9.0);`	`-1.460139106`

See Also

Function

- "ATAN2 Function" on page 128

ATAN2 Function

Returns the arc tangent of the ratio of two numeric variables.

Category: Trigonometric

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

See Also

Functions:

- "ATAN Function" on page 127

ATTRC Function

Returns the value of a character attribute for a SAS data set.

Category: SAS File I/O

Syntax

ATTRC(*data-set-id*,*attr-name*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

attr-name
is the name of a SAS data set attribute. If the value of *attr-name* is invalid, a missing value is returned. The following is a list of SAS data set attribute names and their values:

CHARSET
returns a value for the character set of the computer that created the data set.

empty string	Data set not sorted
ASCII	ASCII character set
EBCDIC	EBCDIC character set
ANSI	OS/2 ANSI standard ASCII character set
OEM	OS/2 OEM code format

ENCRYPT
returns 'YES' or 'NO' depending on whether the SAS data set is encrypted.

ENGINE
returns the name of the engine that is used to access the data set.

LABEL
returns the label assigned to the data set.

LIB
returns the libref of the SAS library in which the data set resides.

MEM
returns the SAS data set name.

MODE
returns the mode in which the SAS data set was opened, such as:

I	INPUT mode allows random access if the engine supports it. Otherwise, it defaults to IN mode.
IN	INPUT mode reads sequentially and allows revisiting observations.
IS	INPUT mode reads sequentially but does not allow revisiting observations.
N	NEW mode creates a new data set.
U	UPDATE mode allows random access if the engine supports it. Otherwise, it defaults to UN mode.
UN	UPDATE mode reads sequentially and allows revisiting observations.
US	UPDATE mode reads sequentially but does not allow revisiting observations.
V	UTILITY mode allows modification of variable attributes and indexes associated with the data set.

MTYPE
returns the SAS library member type.

SORTEDBY
returns an empty string if the data set is not sorted. Otherwise, it returns the names of the BY variables in the standard BY statement format.

SORTLVL
returns a value that indicates how a data set was sorted:

Empty string	Data set is not sorted.
WEAK	Sort order of the data set was established by the user (for example, through the SORTEDBY data set option). The system cannot validate its correctness, so the order of observations cannot be depended on.
STRONG	Sort order of the data set was established by the software (for example, through PROC SORT or the OUT= option in the CONTENTS procedure).

SORTSEQ
returns an empty string if the data set is sorted on the native computer or if the sort collating sequence is the default for the operating environment. Otherwise, it returns the name of the alternate collating sequence used to sort the file.

TYPE
returns the SAS data set type.

Examples

Example 1: Writing a Message about Input Sequential Mode to the SAS Log

This example generates a message if the SAS data set has not been opened in INPUT SEQUENTIAL mode. The message is written to the SAS log as follows:

```
%let mode=%sysfunc(attrc(&dsid,MODE));
%if &mode ne IS %then
   %put Data set has not been opened in INPUT SEQUENTIAL mode.;
```

Example 2: Testing Whether a Data Set Has Been Sorted

This example tests whether a data set has been sorted and writes the result to the SAS log:

```
data _null_;
   dsid=open("sasdata.sortcars","i");
   charset=attrc(dsid,"CHARSET");
   if charset = "" then
      put "Data set has not been sorted.";
   else put "Data set sorted with " charset
            "character set.";
   rc=close(dsid);
run;
```

See Also

Functions

- "ATTRN Function" on page 131
- "OPEN Function" on page 716

ATTRN Function

Returns the value of a numeric attribute for a SAS data set.

Category: SAS File I/O

Syntax

ATTRN(*data-set-id*,*attr-name*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

attr-name
is the name of the SAS data set attribute whose numeric value is returned. If the value of *attr-name* is invalid, a missing value is returned. The following is a list of SAS data set attribute names and their values:

ALTERPW
specifies whether a password is required to alter the data set.

1 the data set is alter protected.

0 the data set is not alter protected.

ANOBS
specifies whether the engine knows the number of observations.

1 the engine knows the number of observations.

0 the engine does not know the number of observations.

ANY
specifies whether the data set has observations or variables.

−1 the data set has no observations or variables.

0 the data set has no observations.

1 the data set has observations and variables.

Alias VAROBS

ARAND
specifies whether the engine supports random access.

1 the engine supports random access.

0 the engine does not support random access.

Alias RANDOM

ARWU
specifies whether the engine can manipulate files.

1 the engine is not read-only. It can create or update SAS files.

0 the engine is read-only.

AUDIT
specifies whether logging to an audit file is enabled.

1 logging is enabled.

0 logging is suspended.

AUDIT_DATA
specifies whether after-update record images are stored.

1 after-update record images are stored.

0 after-update record images are not stored.

AUDIT_BEFORE
specifies whether before-update record images are stored.

1 before-update record images are stored.

0 before-update record images are not stored.

AUDIT_ERROR
specifies whether unsuccessful after-update record images are stored.

1 unsuccessful after-update record images are stored.

0 unsuccessful after-update record images are not stored.

CRDTE
specifies the date that the data set was created. The value that is returned is the internal SAS datetime value for the creation date.

Tip Use the DATETIME. format to display this value.

ICONST
returns information about the existence of integrity constraints for a SAS data set.

0 no integrity constraints.

1 one or more general integrity constraints.

2 one or more referential integrity constraints.

3 both one or more general integrity constraints and one or more referential integrity constraints.

INDEX
specifies whether the data set supports indexing.

1 indexing is supported.

0 indexing is not supported.

ISINDEX
specifies whether the data set is indexed.

1 at least one index exists for the data set.

0 the data set is not indexed.

ISSUBSET
specifies whether the data set is a subset.

1 at least one WHERE clause is active.

0 no WHERE clause is active.

LRECL
specifies the logical record length.

LRID
specifies the length of the record ID.

MAXGEN
specifies the maximum number of generations.

MAXRC
specifies whether an application checks return codes.

1 an application checks return codes.

0 an application does not check return codes.

MODTE
specifies the last date and time that the data set was modified. The value returned is the internal SAS datetime value.

Tip Use the DATETIME. format to display this value.

NDEL
specifies the number of observations in the data set that are marked for deletion.

NEXTGEN
specifies the next generation number to generate.

NLOBS
specifies the number of logical observations (the observations that are not marked for deletion). An active WHERE clause does not affect this number.

−1 the number of observations is not available.

NLOBSF
specifies the number of logical observations (the observations that are not marked for deletion) by forcing each observation to be read and by taking the FIRSTOBS system option, the OBS system option, and the WHERE clauses into account.

Tip Passing NLOBSF to ATTRN requires the engine to read every observation from the data set that matches the WHERE clause. Based on the file type and file size, reading these observations can be a time-consuming process.

NOBS
specifies the number of physical observations (including the observations that are marked for deletion). An active WHERE clause does not affect this number.

−1 the number of observations is not available.

NVARS
specifies the number of variables in the data set.

PW
specifies whether a password is required to access the data set.

1 the data set is protected.

0 the data set is not protected.

RADIX
specifies whether access by observation number (radix addressability) is allowed.

1 access by observation number is allowed.

0 access by observation number is not allowed.

Note: A data set that is accessed by a tape engine is index addressable although it cannot be accessed by an observation number.

READPW
specifies whether a password is required to read the data set.

1 the data set is read protected.

0 the data set is not read protected.

TAPE
specifies the status of the data set tape.

1 the data set is a sequential file.

0 the data set is not a sequential file.

WHSTMT
specifies the active WHERE clauses.

0 no WHERE clause is active.

1 a permanent WHERE clause is active.

2 a temporary WHERE clause is active.

3 both permanent and temporary WHERE clauses are active.

WRITEPW
specifies whether a password is required to write to the data set.

1 the data set is write protected.

0 the data set is not write protected.

Examples

Example 1: Checking for an Active WHERE Clause

This example checks whether a WHERE clause is currently active for a data set.

```
%let iswhere=%sysfunc(attrn(&dsid,whstmt));
%if &iswhere %then
   %put A WHERE clause is currently active.;
```

Example 2: Checking for an Indexed Data Set

This example checks whether a data set is indexed.

```
data _null_;
   dsid=open("mydata");
   isindex=attrn(dsid,"isindex");
   if isindex then put "data set is indexed";
   else put "data set is not indexed";
run;
```

Example 3: Checking a Data Set for Password Protection

This example checks whether a data set is protected with a password.

```
data _null_;
   dsid=open("mydata");
   pw=attrn(dsid,"pw");
   if pw then put "data set is protected";
run;
```

See Also

Functions:

- “ATTRC Function” on page 129
- “OPEN Function” on page 716

BAND Function

Returns the bitwise logical AND of two arguments.

Category: Bitwise Logical Operations

Syntax

band(*argument-1,argument-2*)

Required Argument

argument-1, argument-2
specifies a numeric constant, variable, or expression.

Range between 0 and (2^{32})-1 inclusive

Details

If either argument contains a missing value, then the function returns a missing value and sets _ERROR_ equal to 1.

Example

The following SAS statements produce this result.

SAS Statement	Result
`x=band(0Fx,05x);` `put x=hex.;`	`x=00000005`

BETA Function

Returns the value of the beta function.

Category: Mathematical

Syntax

BETA(*a*,*b*)

Required Arguments

a
is the first shape parameter, where $a>0$.

b
is the second shape parameter, where $b>0$.

Details

The BETA function is mathematically given by the equation

$$\beta(a, b) = \int_0^1 x^{a-1}(1-x)^{b-1}\,dx$$

with $a>0$, $b>0$. It should be noted that

$$\beta(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

where $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, BETA returns a missing value.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=beta(5,3);`	`0.9523809524e-2`

See Also

Functions:

- “LOGBETA Function” on page 640

BETAINV Function

Returns a quantile from the beta distribution.

Category: Quantile

Syntax

BETAINV (*p*,*a*,*b*)

Required Arguments

p

is a numeric probability.

Range $0 \le p \le 1$

a

is a numeric shape parameter.

Range $a > 0$

b
is a numeric shape parameter.

Range $b > 0$

Details

The BETAINV function returns the *p*th quantile from the beta distribution with shape parameters *a* and *b*. The probability that an observation from a beta distribution is less than or equal to the returned quantile is *p*.

Note: BETAINV is the inverse of the PROBBETA function.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=betainv(0.001,2,4);`	`0.0101017879`

See Also

Functions:

- "QUANTILE Function" on page 799

BLACKCLPRC Function

Calculates call prices for European options on futures, based on the Black model.

Category: Financial

Syntax

BLACKCLPRC(*E*, *t*, *F*, *r*, *sigma*)

Required Arguments

E
is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

t
is a nonmissing value that specifies time to maturity.

F
is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

r
is a nonmissing, positive fraction that specifies the risk-free interest rate between the present time and *t*.

Requirement Specify a value for *r* for the same time period as the unit of *t*.

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

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:

$$CALL = \varepsilon^{-rt}\left(FN(d_1) - EN(d_2)\right)$$

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 for period *t*.

t
specifies the time to expiration.

$$d_1 = \frac{\left(\ln\left(\frac{F}{E}\right) + \left(\frac{\sigma^2}{2}\right)t\right)}{\sigma\sqrt{t}}$$

$$d_2 = d_1 - \sigma\sqrt{t}$$

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:

$$CALL = \max((F - E), 0)$$

For information about the basics of pricing, see Using Pricing Functions on page 8.

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 SAS statements produce these results.

SAS Statement	Result
	----+----1----+---2--
a=blackclprc(1000, .5, 950, 4, 2); put a;	65.335687119
b=blackclprc(850, 2.5, 125, 3, 1); put b;	0.012649067
c=blackclprc(7500, .9, 950, 3, 2); put c;	17.880939441
d=blackclprc(5000, -.5, 237, 3, 2); put d;	0

See Also

Functions:

- "BLACKPTPRC Function" on page 140

BLACKPTPRC Function

Calculates put prices for European options on futures, based on the Black model.

Category: Financial

Syntax

BLACKPTPRC(*E*, *t*, *F*, *r*, *sigma*)

Required Arguments

E

is a nonmissing, positive value that specifies exercise price.

Requirement Specify *E* and *F* in the same units.

t

is a nonmissing value that specifies time to maturity.

F

is a nonmissing, positive value that specifies future price.

Requirement Specify *F* and *E* in the same units.

r
is a nonmissing, positive fraction that specifies the risk-free interest rate between the present time and *t*.

Requirement Specify a value for *r* for the same time period as the unit of *t*.

sigma
is a nonmissing, positive fraction that specifies the volatility (the square root of the variance of *r*).

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

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:

$$PUT = CALL + \varepsilon^{-rt}(E - F)$$

Arguments

E
specifies the exercise price of the option.

r
specifies the risk-free interest rate for period *t*.

t
specifies the time to expiration.

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:

$$PUT = \max((E - F), 0)$$

For information about the basics of pricing, see Using Pricing Functions on page 8.

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 SAS statements produce these results.

SAS Statement	Result
	----+----1----+---2--
a=blackptprc(1000, .5, 950, 4, 2); put a;	72.102451281
b=blackptprc(850, 2.5, 125, 3, 1); put b;	0.4136352354
c=blackptprc(7500, .9, 950, 3, 2); put c;	458.07704789
d=blackptprc(5000, -.5, 237, 3, 2); put d;	0

See Also

Functions:

- "BLACKCLPRC Function" on page 138

BLKSHCLPRC Function

Calculates call prices for European options on stocks, based on the Black-Scholes model.

Category: Financial

Syntax

BLKSHCLPRC(*E*, *t*, *S*, *r*, *sigma*)

Required Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t

is a nonmissing value that specifies the time to maturity.

S

is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

r

is a nonmissing, positive fraction that specifies the risk-free interest rate for period *t*.

Requirement Specify a value for *r* for the same time period as the unit of *t*.

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

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:

$$CALL = SN(d_1) - EN(d_2)\varepsilon^{-rt}$$

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.

r
specifies the risk-free interest rate for period *t*.

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:

$$CALL = \max((S - E), 0)$$

For information about the basics of pricing, see Using Pricing Functions on page 8 .

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 SAS statements produce these results.

SAS Statement	Result
	----+----1----+---2--
a=blkshclprc(1000, .5, 950, 4, 2); put a;	831.05008469
b=blkshclprc(850, 2.5, 125, 3, 1); put b;	124.53035232
c=blkshclprc(7500, .9, 950, 3, 2); put c;	719.40891129
d=blkshclprc(5000, -.5, 237, 3, 2); put d;	0

See Also

Functions:

- "BLKSHPTPRC Function" on page 144

BLKSHPTPRC Function

Calculates put prices for European options on stocks, based on the Black-Scholes model.

Category: Financial

Syntax

BLKSHPTPRC(*E*, *t*, *S*, *r*, *sigma*)

Required Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t

is a nonmissing value that specifies the time to maturity.

S

is a nonmissing, positive value that specifies the share price.

Requirement Specify *S* and *E* in the same units.

r

is a nonmissing, positive fraction that specifies the risk-free interest rate for period *t*.

Requirement Specify a value for *r* for the same time period as the unit of *t*.

sigma
is a nonmissing, positive fraction that specifies the volatility of the underlying asset.

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

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:

$$PUT = CALL - S + E\varepsilon^{-rt}$$

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.

r
specifies the risk-free interest rate for period *t*.

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:

$$PUT = \max((E - S), 0)$$

For information about the basics of pricing, see Using Pricing Functions on page 8.

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 SAS statements produce these results.

SAS Statement	Result
	`----+----1----+---2--`

SAS Statement	Result
`a=blkshptprc(1000, .5, 950, 4, 2);` `put a;`	16.385367922
`b=blkshptprc(850, 1.2, 125, 3, 1);` `put b;`	1.426971358
`c=blkshptprc(7500, .9, 950, 3, 2);` `put c;`	273.45025684
`d=blkshptprc(5000, -.5, 237, 3, 2);` `put d;`	0

See Also

Functions:

- "BLKSHCLPRC Function" on page 142

BLSHIFT Function

Returns the bitwise logical left shift of two arguments.

Category: Bitwise Logical Operations

Syntax

BLSHIFT(*argument-1*,*argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range between 0 and (2^{32})-1 inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range 0 to 31, inclusive

Details

If either argument contains a missing value, then the function returns a missing value and sets _ERROR_ equal to 1.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=blshift(07x,2);` `put x=hex.;`	`x=0000001C`

BNOT Function

Returns the bitwise logical NOT of an argument.

Category: Bitwise Logical Operations

Syntax

BNOT(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range between 0 and (2^{32})-1 inclusive

Details

If the argument contains a missing value, then the function returns a missing value and sets _ERROR_ equal to 1.

Example

The following SAS statements produce this result.

SAS Statement	Result
`x=bnot(0F000000Fx);` `put x=hex.;`	`x=0FFFFFF0`

BOR Function

Returns the bitwise logical OR of two arguments.

Category: Bitwise Logical Operations

Syntax

BOR(*argument-1*,*argument-2*)

Required Argument

argument-1, argument-2
specifies a numeric constant, variable, or expression.

Range between 0 and (2^{32})-1 inclusive

Details

If either argument contains a missing value, then the function returns a missing value and sets _ERROR_ equal to 1.

Example

The following SAS statements produce this result.

SAS Statement	Result
`x=bor(01x,0F4x);` `put x=hex.;`	`x=000000F5`

BRSHIFT Function

Returns the bitwise logical right shift of two arguments.

Category: Bitwise Logical Operations

Syntax

BRSHIFT(*argument-1*,*argument-2*)

Required Arguments

argument-1
specifies a numeric constant, variable, or expression.

Range between 0 and (2^{32})-1 inclusive

argument-2
specifies a numeric constant, variable, or expression.

Range 0 to 31, inclusive

Details

If either argument contains a missing value, then the function returns a missing value and sets _ERROR_ equal to 1.

Example

The following SAS statements produce this result.

SAS Statement	Result
`x=brshift(01Cx,2);` `put x=hex.;`	`x=00000007`

BXOR Function

Returns the bitwise logical EXCLUSIVE OR of two arguments.

Category: Bitwise Logical Operations

Syntax

BXOR(*argument-1, argument-2*)

Required Argument

argument-1, argument-2
specifies a numeric constant, variable, or expression.

Range between 0 and (2^{32})-1 inclusive

Details

If either argument contains a missing value, then the function returns a missing value and sets _ERROR_ equal to 1.

Example

The following SAS statements produce this result.

SAS Statement	Result
`x=bxor(03x,01x);` `put x=hex.;`	`x=00000002`

BYTE Function

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

Category: Character

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

See: “BYTE Function: UNIX” in *SAS Companion for UNIX Environments*
“BYTE Function: Windows” in *SAS Companion for Windows*

Syntax

BYTE (*n*)

Required Argument

n
specifies an integer that represents a specific ASCII or EBCDIC character.

Range 0–255

Details

Length of Returned Variable

In a DATA step, if the BYTE function returns a value to a variable that has not previously been assigned a length, then that variable is assigned a length of 1.

ASCII and EBCDIC Collating Sequences

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 SAS statements produce these results.

SAS Statement	Result	
	ASCII	EBCDIC
	`----+----1----+----2`	`----+----1----+----2`
`x=byte(80);` `put x;`	`P`	`&`

See Also

Functions:

- “COLLATE Function” on page 308
- “RANK Function” on page 820

CALL ALLCOMB Routine

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

Category: Combinatorial

Syntax

CALL ALLCOMB(*count*, *k*, *variable-1*, …,*variable-n*);

Required Arguments

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

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

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

Restriction	Specify no more than 33 items. If you need to find combinations of more than 33 items, use the CALL ALLCOMBI routine.
Requirement	Initialize these variables before calling the ALLCOMB routine.
Tip	After calling the ALLCOMB routine, the first *k* variables contain the values in one combination.

Details

CALL ALLCOMB Processing

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

If you call the ALLCOMB routine with the first argument out of sequence, the results are not useful. In particular, if you initialize the variables and then immediately call ALLCOMB with a first argument of *j*, then you will not get the j^{th} combination (except when *j* is 1). To get the j^{th} combination, you must call ALLCOMB *j* times, with the first argument taking values from 1 through *j* in that exact order.

Using the CALL ALLCOMB Routine with Macros

You can call the ALLCOMB routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL ALLCOMB routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, then &SYSERR is set to zero, and &SYSINFO is set to one of the following values:

- 0 if *count*=1
- *j* if the values of *variable-j* and *variable-k* were interchanged, where *j*<*k*

- –1 if no values were interchanged because all distinct combinations were already generated

Comparisons

SAS provides four functions or CALL routines for generating combinations:

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

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

Examples

Example 1: Using CALL ALLCOMB in a DATA Step

The following is an example of the CALL ALLCOMB routine that is used with the DATA step.

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

SAS writes the following output to the log:

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

Example 2: Using CALL ALLCOMB with Macros and Displaying the Return Code

The following is an example of the CALL ALLCOMB routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=2;
   %let ncomb=%sysfunc(comb(5,&k));
   %do j=1 %to &ncomb+1;
      %syscall allcomb(j, k, x1, x2, x3, x4, x5);
      %let jfmt=%qsysfunc(putn(&j,5.));
      %let pad=%qsysfunc(repeat(%str(),30-%length(&x1 &x2)));
      %put &jfmt:  &x1 &x2 &pad sysinfo=&sysinfo;
   %end;
%mend;

%test
```

SAS writes the following output to the log:

```
1:  ant -0.1234  sysinfo=0
 2:  ant zebra  sysinfo=2
 3:  ant hippopotamus  sysinfo=2
 4:  ant 10000000000  sysinfo=2
 5:  -0.1234 10000000000  sysinfo=1
 6:  -0.1234 zebra  sysinfo=2
 7:  -0.1234 hippopotamus  sysinfo=2
 8:  10000000000 hippopotamus  sysinfo=1
 9:  10000000000 zebra  sysinfo=2
10:  hippopotamus zebra  sysinfo=1
11:  hippopotamus zebra  sysinfo=-1
```

See Also

Functions:

- “ALLCOMB Function” on page 95

CALL ALLCOMBI Routine

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

Category: Combinatorial

Syntax

CALL ALLCOMBI(*N*, *K*, *index-1*, …, *index-K*, <, *index-added*, *index-removed*>);

Required Arguments

N
is a numeric constant, variable, or expression that specifies the total number of objects.

K
is a numeric constant, variable, or expression that specifies the number of objects in each combination.

index
is a numeric variable that contains indices of the objects in the returned combination. Indices are integers between 1 and N inclusive.

Tip If *index-1* is missing or zero, then ALLCOMBI initializes the indices to **`index-1=1`** through **`index-K=K`**. Otherwise, ALLCOMBI creates a new combination by removing one index from the combination and adding another index.

Optional Arguments

index-added
is a numeric variable in which ALLCOMBI returns the value of the index that was added.

index-removed
is a numeric variable in which ALLCOMBI returns the value of the index that was removed.

Details

CALL ALLCOMBI Processing

Before you make the first call to ALLCOMBI, complete one of the following tasks:

- Set *index-1* equal to zero or to a missing value.
- Initialize *index-1* through *index-K* to distinct integers between 1 and N inclusive.

The number of combinations of N objects taken K at a time can be computed as COMB(*N*, *K*). To generate all combinations of N objects taken K at a time, call ALLCOMBI in a loop that executes COMB(*N*, *K*) times.

Using the CALL ALLCOMBI Routine with Macros

If you call ALLCOMBI from the macro processor with %SYSCALL, then you must initialize all arguments to numeric values. &SYSCALL reformats the values that are returned.

If an error occurs during the execution of the CALL ALLCOMBI routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, then &SYSERR and &SYSINFO are set to zero.

Comparisons

The CALL ALLCOMBI routine generates all combinations of the indices of N objects taken K at a time in a minimal change order. The CALL ALLCOMB routine generates

all combinations of the values of N variables taken K at a time in a minimal change order.

Examples

Example 1: Using CALL ALLCOMBI in a DATA Step

The following is an example of the CALL ALLCOMBI routine that is used in a DATA step.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   array c[3] $3;
   array i[3];
   n=dim(x);
   k=dim(i);
   i[1]=0;
   ncomb=comb(n,k);    /* The one extra call goes back */
   do j=1 to ncomb+1;  /* to the first combination. */
      call allcombi(n, k, of i[*], add, remove);
      do h=1 to k;
         c[h]=x[i[h]];
      end;
      put @4 j= @10 'i= ' i[*] +3 'c= ' c[*] +3 add= remove=;
   end;
run;
```

SAS writes the following output to the log:

```
j=1   i= 1 2 3    c= ant bee cat    add=0 remove=0
j=2   i= 1 3 4    c= ant cat dog    add=4 remove=2
j=3   i= 2 3 4    c= bee cat dog    add=2 remove=1
j=4   i= 1 2 4    c= ant bee dog    add=1 remove=3
j=5   i= 1 4 5    c= ant dog ewe    add=5 remove=2
j=6   i= 2 4 5    c= bee dog ewe    add=2 remove=1
j=7   i= 3 4 5    c= cat dog ewe    add=3 remove=2
j=8   i= 1 3 5    c= ant cat ewe    add=1 remove=4
j=9   i= 2 3 5    c= bee cat ewe    add=2 remove=1
j=10  i= 1 2 5    c= ant bee ewe    add=1 remove=3
j=11  i= 1 2 3    c= ant bee cat    add=3 remove=5
```

Example 2: Using CALL ALLCOMBI with Macros

The following is an example of the CALL ALLCOMBI routine that is used with macros.

```
%macro test;
   %let x1=0;
   %let x2=0;
   %let x3=0;
   %let add=0;
   %let remove=0;
   %let n=5;
   %let k=3;
   %let ncomb=%sysfunc(comb(&n,&k));
   %do j=1 %to &ncomb;
      %syscall allcombi(n,k,x1,x2,x3,add,remove);
      %let jfmt=%qsysfunc(putn(&j,5.));
      %put &jfmt: &x1 &x2 &x3 add=&add remove=&remove;
```

```
   %end;
%mend;
%test
```

SAS writes the following output to the log:

```
 1: 1 2 3 add=0 remove=0
 2: 1 3 4 add=4 remove=2
 3: 2 3 4 add=2 remove=1
 4: 1 2 4 add=1 remove=3
 5: 1 4 5 add=5 remove=2
 6: 2 4 5 add=2 remove=1
 7: 3 4 5 add=3 remove=2
 8: 1 3 5 add=1 remove=4
 9: 2 3 5 add=2 remove=1
10: 1 2 5 add=1 remove=3
```

See Also

CALL Routines:

- "CALL ALLCOMB Routine" on page 150

CALL ALLPERM Routine

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

Category: Combinatorial

Syntax

CALL ALLPERM(*count*, *variable–1*<, *variable–2 ...*>);

Required Arguments

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

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

Restriction Specify no more than 18 variables.

Requirement Initialize these variables before you call the ALLPERM routine.

Details

CALL ALLPERM Processing

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

Note: You can compute the number of permutations by using the PERM function. See PERM Function on page 743 for more information.

If you call the ALLPERM routine and the first argument is out of sequence, the results are not useful. In particular, if you initialize the variables and then immediately call the ALLPERM routine with a first argument of K, your result will not be the K*th* permutation (except when K is 1). To get the K*th* permutation, you must call the ALLPERM routine K times, with the first argument taking values from 1 through K in that exact order.

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

Using the CALL ALLPERM Routine with Macros

You can call the ALLPERM routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL ALLPERM routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than -100.

If there are no errors, then &SYSERR is set to zero, and &SYSINFO is set to one of the following values:

- 0 if *count*=1
- J if 1<*count*<=N! and the values of *variable-J* and *variable-K* were interchanged, where J+1=K
- -1 if *count*>N!

Comparisons

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

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

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

Examples

Example 1: Using CALL ALLPERM in a DATA Step

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

```
data _null_;
```

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

SAS writes the following output to the log:

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

Example 2: Using CALL ALLPERM with Macros

The following is an example of the CALL ALLPERM routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let nperm=%sysfunc(perm(4));
   %do j=1 %to &nperm+1;
      %syscall allperm(j, x1, x2, x3, x4);
      %let jfmt=%qsysfunc(putn(&j,5.));
      %put &jfmt:   &x1 &x2 &x3 &x4 sysinfo=&sysinfo;
   %end;
%mend;

%test;
```

SAS writes the following output to the log:

```
1:   ant -0.1234 10000000000 hippopotamus sysinfo=0
```

```
 2:   ant -0.1234 hippopotamus 10000000000 sysinfo=3
 3:   ant hippopotamus -0.1234 10000000000 sysinfo=2
 4:   hippopotamus ant -0.1234 10000000000 sysinfo=1
 5:   hippopotamus ant 10000000000 -0.1234 sysinfo=3
 6:   ant hippopotamus 10000000000 -0.1234 sysinfo=1
 7:   ant 10000000000 hippopotamus -0.1234 sysinfo=2
 8:   ant 10000000000 -0.1234 hippopotamus sysinfo=3
 9:   10000000000 ant -0.1234 hippopotamus sysinfo=1
10:   10000000000 ant hippopotamus -0.1234 sysinfo=3
11:   10000000000 hippopotamus ant -0.1234 sysinfo=2
12:   hippopotamus 10000000000 ant -0.1234 sysinfo=1
13:   hippopotamus 10000000000 -0.1234 ant sysinfo=3
14:   10000000000 hippopotamus -0.1234 ant sysinfo=1
15:   10000000000 -0.1234 hippopotamus ant sysinfo=2
16:   10000000000 -0.1234 ant hippopotamus sysinfo=3
17:   -0.1234 10000000000 ant hippopotamus sysinfo=1
18:   -0.1234 10000000000 hippopotamus ant sysinfo=3
19:   -0.1234 hippopotamus 10000000000 ant sysinfo=2
20:   hippopotamus -0.1234 10000000000 ant sysinfo=1
21:   hippopotamus -0.1234 ant 10000000000 sysinfo=3
22:   -0.1234 hippopotamus ant 10000000000 sysinfo=1
23:   -0.1234 ant hippopotamus 10000000000 sysinfo=2
24:   -0.1234 ant 10000000000 hippopotamus sysinfo=3
25:   -0.1234 ant 10000000000 hippopotamus sysinfo=-1
```

See Also

Functions:

- “LEXPERM Function” on page 629
- “ALLPERM Function” on page 97

CALL Routines:

- “CALL RANPERK Routine” on page 224
- “CALL RANPERM Routine” on page 226

CALL CATS Routine

Removes leading and trailing blanks, and returns a concatenated character string.

Category: Character

Syntax

CALL CATS(*result* <, *item-1*, …, *item-n*>);

Required Argument

result
specifies a character variable.

Restriction The CALL CATS routine accepts only a character variable as a valid argument for *result*. Do not use a constant or a SAS expression because CALL CATS is unable to update these arguments.

Optional Argument

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

Details

The CALL CATS routine returns the result in the first argument, *result*. The routine appends the values of the arguments that follow to *result*. If the length of *result* is not large enough to contain the entire result, SAS does the following:

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause

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

Comparisons

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.

CALL Routine	Equivalent Statement
`CALL CATS(OF X1-X4);`	`X1=TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4));`
`CALL CATT(OF X1-X4);`	`X1=TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4);`
`CALL CATX(SP, OF X1-X4); *`	`X1=TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4));`

Note: If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding separator. For example, **`CALL CATX("+","X"," ", "Z"," ");`** produces **`X+Z`**.

Example

The following example shows how the CALL CATS routine concatenates strings.

```
data _null_;
   length answer $ 36;
   x='Athens is t  ';
   y=' he Olym   ';
   z='   pic site for 2004. ';
   call cats(answer,x,y,z);
   put answer;
run;
```

The following line is written to the SAS log:

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7
Athens is the Olympic site for 2004.
```

See Also

Functions:

- “CAT Function” on page 263
- “CATQ Function” on page 266
- “CATS Function” on page 270
- “CATT Function” on page 272
- “CATX Function” on page 274

CALL Routines:

- “CALL CATX Routine” on page 163
- “CALL CATT Routine” on page 161

CALL CATT Routine

Removes trailing blanks, and returns a concatenated character string.

Category: Character

Syntax

CALL CATT(*result* <, *item-1*, … *item-n*>);

Required Argument

result
specifies a character variable.

Restriction The CALL CATT routine accepts only a character variable as a valid argument for *result*. Do not use a constant or a SAS expression because CALL CATT is unable to update these arguments.

Optional Argument

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

Details

The CALL CATT routine returns the result in the first argument, *result*. The routine appends the values of the arguments that follow to *result*. If the length of *result* is not large enough to contain the entire result, SAS does the following:

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause

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

Comparisons

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a separator, such as a blank or comma.

CALL Routine	Equivalent Statement
`CALL CATS(OF X1-X4);`	`X1=TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4));`
`CALL CATT(OF X1-X4);`	`X1=TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4);`
`CALL CATX(SP, OF X1-X4); *`	`X1=TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4));`

Note: If any of the arguments is blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding separator. For example, **`CALL CATX("+","X"," ", "Z"," ");`** produces **`X+Z`**.

Example

The following example shows how the CALL CATT routine concatenates strings.

```
data _null_;
   length answer $ 36;
   x='London is t  ';
   y='he Olym   ';
   z='pic site for 2012. ';
```

```
    call catt(answer,x,y,z);
    put answer;
run;
```

The following line is written to the SAS log:

```
----+----1----+----2----+----3----+----4
London is the Olympic site for 2012.
```

See Also

Functions:

- “CAT Function” on page 263
- “CATQ Function” on page 266
- “CATS Function” on page 270
- “CATT Function” on page 272
- “CATX Function” on page 274

CALL Routines:

- “CALL CATX Routine” on page 163
- “CALL CATS Routine” on page 159

CALL CATX Routine

Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.

Category: Character

Syntax

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

Required Arguments

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

result
specifies a character variable.

Restriction The CALL CATX routine accepts only a character variable as a valid argument for *result*. Do not use a constant or a SAS expression because CALL CATX is unable to update these arguments.

Optional Argument

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

Details

The CALL CATX routine returns the result in the second argument, *result*. The routine appends the values of the arguments that follow to *result*. If the length of *result* is not large enough to contain the entire result, SAS does the following:

- writes a warning message to the log stating that the result was truncated
- writes a note to the log that shows the location of the function call and lists the argument that caused the truncation, except in SQL or in a WHERE clause
- sets _ERROR_ to 1 in the DATA step, except in a WHERE clause

The CALL CATX routine removes leading and trailing blanks from numeric arguments after formatting the numeric value with the BEST*w*. format.

Comparisons

The results of the CALL CATS, CALL CATT, and CALL CATX routines are usually equivalent to statements that use the concatenation operator (||) and the TRIM and LEFT functions. However, using the CALL CATS, CALL CATT, and CALL CATX routines is faster than using TRIM and LEFT.

The following table shows statements that are equivalent to CALL CATS, CALL CATT, and CALL CATX. The variables X1 through X4 specify character variables, and SP specifies a delimiter, such as a blank or comma.

CALL Routine	Equivalent Statement
`CALL CATS(OF X1-X4);`	`X1=TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4));`
`CALL CATT(OF X1-X4);`	`X1=TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4);`
`CALL CATX(SP, OF X1-X4); *`	`X1=TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4));`

Note: If any of the arguments are blank, the results that are produced by CALL CATX differ slightly from the results that are produced by the concatenated code. In this case, CALL CATX omits the corresponding delimiter. For example, **`CALL CATX("+",newvar,"X"," ", "Z"," ");`** produces **`X+Z`**.

Example

The following example shows how the CALL CATX routine concatenates strings.

```
data _null_;
   length answer $ 50;
   separator='%%$%%';
   x='Athens is t  ';
   y='he Olym     ';
   z='  pic site for 2004.  ';
   call catx(separator,answer,x,y,z);
   put answer;
run;
```

The following line is written to the SAS log:

```
----+----1----+----2----+----3----+----4----+----5
Athens is t%%$%%he Olym%%$%%pic site for 2004.
```

See Also

Functions:

- "CAT Function" on page 263
- "CATQ Function" on page 266
- "CATS Function" on page 270
- "CATT Function" on page 272
- "CATX Function" on page 274

CALL Routines:

- "CALL CATS Routine" on page 159
- "CALL CATT Routine" on page 161

CALL COMPCOST Routine

Sets the costs of operations for later use by the COMPGED function

Category: Character

Restriction: Use with the COMPGED function

Interaction: When invoked by the %SYSCALL macro statement, CALL COMPCOST removes quotation marks from its arguments. For more information, see "Using CALL Routines and the %SYSCALL Macro Statement" on page 9.

Syntax

CALL COMPCOST(*operation-1*, *value-1* <,*operation-2*, *value-2* …>);

Required Arguments

operation
is a character constant, variable, or expression that specifies an operation that is performed by the COMPGED function.

value
is a numeric constant, variable, or expression that specifies the cost of the operation that is indicated by the preceding argument.

Restriction Must be an integer that ranges from –32767 through 32767, or a missing value

Details

Computing the Cost of Operations

Each argument that specifies an operation must have a value that is a character string. The character string corresponds to one of the terms that is used to denote an operation that the COMPGED function performs. See "Computing the Generalized Edit Distance" on page 318 to view a table of operations that the COMPGED function uses.

The character strings that specify operations can be in uppercase, lowercase, or mixed case. Blanks are ignored. Each character string must end with an equal sign (=). Valid values for operations, and the default cost of the operations are listed in the following table.

Operation	Default Cost
APPEND=	very large
BLANK=	very large
DELETE=	100
DOUBLE=	very large
FDELETE=	equal to DELETE
FINSERT=	equal to INSERT
FREPLACE=	equal to REPLACE
INSERT=	100
MATCH=	0
PUNCTUATION=	very large
REPLACE=	100
SINGLE=	very large
SWAP=	very large
TRUNCATE=	very large

If an operation does not appear in the call to the COMPCOST routine, or if the operation appears and is followed by a missing value, then that operation is assigned a default cost. A "very large" cost indicates a cost that is sufficiently large that the COMPGED function will not use the corresponding operation.

After your program calls the COMPCOST routine, the costs that are specified remain in effect until your program calls the COMPCOST routine again, or until the step that contains the call to COMPCOST terminates.

Abbreviating Character Strings

You can abbreviate character strings. That is, you can use the first one or more letters of a specific operation rather than use the entire term. You must, however, use as many letters as necessary to uniquely identify the term. For example, you can specify the INSERT= operation as “in=”, and the REPLACE= operation as “r=”. To specify the DELETE= or the DOUBLE= operation, you must use the first two letters because both DELETE= and DOUBLE= begin with “d”. The character string must always end with an equal sign.

Example

The following example calls the COMPCOST routine to compute the generalized edit distance for the operations that are specified.

```
options pageno=1 nodate linesize=80 pagesize=60;
data test;
   length String $8 Operation $40;
   if _n_ = 1 then call compcost('insert=',10,'DEL=',11,'r=', 12);
   input String Operation;
   GED=compged(string, 'baboon');
   datalines;
baboon  match
xbaboon insert
babon   delete
baXoon  replace
;
proc print data=test label;
   label GED='Generalized Edit Distance';
   var String Operation GED;
run;
```

The following output shows the results.

Display 2.1 *Generalized Edit Distance Based on Operation*

The SAS System

Obs	String	Operation	Generalized Edit Distance
1	baboon	match	0
2	xbaboon	insert	10
3	babon	delete	11
4	baXoon	replace	12

See Also

Functions:

- "COMPGED Function" on page 317
- "COMPARE Function" on page 311
- "COMPLEV Function" on page 323

CALL EXECUTE Routine

Resolves the argument, and issues the resolved value for execution at the next step boundary.

Category: Macro

Syntax

CALL EXECUTE(*argument*);

Required Argument

argument
specifies a character expression or a constant that yields a macro invocation or a SAS statement. *Argument* can be:

- a character string, enclosed in quotation marks.
- the name of a DATA step character variable. Do not enclose the name of the DATA step variable in quotation marks.
- a character expression that the DATA step resolves to a macro text expression or a SAS statement.

Details

If *argument* resolves to a macro invocation, the macro executes immediately and DATA step execution pauses while the macro executes. If *argument* resolves to a SAS statement or if execution of the macro generates SAS statements, the statement(s) execute after the end of the DATA step that contains the CALL EXECUTE routine. CALL EXECUTE is fully documented in *SAS Macro Language: Reference*.

CALL GRAYCODE Routine

Generates all subsets of *n* items in a minimal change order.

Category: Combinatorial

Syntax

CALL GRAYCODE(*k*, *numeric-variable-1*, ..., *numeric-variable-n*);

CALL GRAYCODE(*k*, *character-variable* <, *n* <, *in-out*> >);

Required Arguments

k
specifies a numeric variable. Initialize *k* to either of the following values before executing the CALL GRAYCODE routine:

- a negative number to cause CALL GRAYCODE to initialize the subset to be empty
- the number of items in the initial set indicated by *numeric-variable-1* through *numeric-variable-n*, or *character-variable*, which must be an integer value between 0 and N inclusive

The value of *k* is updated when CALL GRAYCODE is executed. The value that is returned is the number of items in the subset.

numeric-variable
specifies numeric variables that have values of 0 or 1 which are updated when CALL GRAYCODE is executed. A value of 1 for *numeric-variable-j* indicates that the j^{th} item is in the subset. A value of 0 for *numeric-variable-j* indicates that the j^{th} item is not in the subset.

If you assign a negative value to *k* before you execute CALL GRAYCODE, then you do not need to initialize *numeric-variable-1* through *numeric-variable-n* before executing CALL GRAYCODE unless you want to suppress the note about uninitialized variables.

If you assign a value between 0 and *n* inclusive to *k* before you execute CALL GRAYCODE, then you must initialize *numeric-variable-1* through *numeric-variable-n* to *k* values of 1 and *n-k* values of 0.

character-variable
specifies a character variable that has a length of at least *n* characters. The first *n* characters indicate which items are in the subset. By default, an "I" in the j^{th} position indicates that thej^{th} item is in the subset, and an "O" in the j^{th} position indicates that the j^{th} item is out of the subset. You can change the two characters by specifying the *in-out* argument.

If you assign a negative value to *k* before you execute CALL GRAYCODE, then you do not need to initialize *character-variable* before executing CALL GRAYCODE unless you want to suppress the note about an uninitialized variable.

If you assign a value between 0 and *n* inclusive to *k* before you execute CALL GRAYCODE, then you must initialize *character-variable* to *k* characters that indicate an item is in the subset, and *k-k* characters that indicate an item is out of the subset.

Optional Arguments

n
specifies a numeric constant, variable, or expression. By default, *n* is the length of *character-variable*.

in-out
specifies a character constant, variable, or expression. The default value is "IO." The first character is used to indicate that an item is in the subset. The second character is used to indicate that an item is out of the subset.

Details

Using CALL GRAYCODE in a DATA Step

When you execute the CALL GRAYCODE routine with a negative value of *k*, the subset is initialized to be empty.

When you execute the CALL GRAYCODE routine with an integer value of *k* between 0 and *n* inclusive, one item is either added to the subset or removed from the subset, and the value of *k* is updated to equal the number of items in the subset.

To generate all subsets of *n* items, you can initialize *k* to a negative value and execute CALL GRAYCODE in a loop that iterates 2**n* times. If you want to start with a non-empty subset, then initialize *k* to be the number of items in the subset, initialize the other arguments to specify the desired initial subset, and execute CALL GRAYCODE in a loop that iterates 2***n*-1 times. The sequence of subsets that are generated by CALL GRAYCODE is cyclical, so you can begin with any subset that you want.

Using the CALL GRAYCODE Routine with Macros

You can call the GRAYCODE routine when you use the %SYSCALL macro. Differences exist when you use CALL GRAYCODE in a DATA step and when you use the routine with macros. The following list describes usage with macros:

- All arguments must be initialized to nonblank values.
- If you use the *character-variable* argument, then it must be initialized to a nonblank, nonnumeric character string that contains at least *n* characters.
- If you use the *in-out* argument, then it must be initialized to a string that contains two characters that are not blanks, digits, decimal points, or plus and minus signs.

If %SYSCALL identifies an argument as being the wrong type, or if %SYSCALL is unable to identify the type of argument, then &SYSERR and &SYSINFO are *not* set.

Otherwise, if an error occurs during the execution of the CALL GRAYCODE routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, then &SYSERR is set to zero, and &SYSINFO is set to one of the following values:

- 0 if the value of *k* on input is negative
- the index of the item that was added or removed from the subset if the value of *k* on input is a valid nonnegative integer.

Examples

Example 1: Using a Character Variable and Positive Initial k with CALL GRAYCODE

The following example uses the CALL GRAYCODE routine to generate subsets in a minimal change order.

```
data _null_;
   x='++++';
   n=length(x);
   k=countc(x, '+');
   put '    1' +3 k= +2 x=;
   nsubs=2**n;
   do i=2 to nsubs;
      call graycode(k, x, n, '+-');
      put i 5. +3 k= +2 x=;
   end;
run;
```

SAS writes the following output to the log:

```
 1   k=4   x=++++
 2   k=3   x=-+++
 3   k=2   x=-+-+
 4   k=3   x=++-+
 5   k=2   x=+--+
 6   k=1   x=---+
 7   k=0   x=----
 8   k=1   x=+---
 9   k=2   x=++--
10   k=1   x=-+--
11   k=2   x=-++-
12   k=3   x=+++-
13   k=2   x=+-+-
14   k=1   x=--+-
15   k=2   x=--++
16   k=3   x=+-++
```

Example 2: Using %SYSCALL with Numeric Variables and Negative k

The following example uses the %SYSCALL macro with numeric variables to generate subsets in a minimal change order.

```
%macro test;
   %let n=3;
   %let x1=.;
   %let x2=.;
   %let x3=.;
   %let k=-1;
   %let nsubs=%eval(2**&n + 1);
   %put nsubs=&nsubs k=&k x: &x1 &x2 &x3;
   %do j=1 %to &nsubs;
      %syscall graycode(k, x1, x2, x3);
      %put &j: k=&k x: &x1 &x2 &x3 sysinfo=&sysinfo;
   %end;
%mend;
%test;
```

SAS writes the following output to the log:

```
nsubs=9 k=-1 x: . . .
1: k=0 x: 0 0 0 sysinfo=0
2: k=1 x: 1 0 0 sysinfo=1
3: k=2 x: 1 1 0 sysinfo=2
4: k=1 x: 0 1 0 sysinfo=1
5: k=2 x: 0 1 1 sysinfo=3
6: k=3 x: 1 1 1 sysinfo=1
7: k=2 x: 1 0 1 sysinfo=2
8: k=1 x: 0 0 1 sysinfo=1
9: k=0 x: 0 0 0 sysinfo=3
```

Example 3: Using %SYSCALL with a Character Variable and Negative k

The following example uses the %SYSCALL macro with a character variable to generate subsets in a minimal change order.

```
%macro test(n);
   %*** Initialize the character variable to a
        sufficiently long nonblank, nonnumeric value. ;
   %let x=%sysfunc(repeat(_, &n-1));
   %let k=-1;
   %let nsubs=%eval(2**&n + 1);
   %put nsubs=&nsubs k=&k x="&x";
   %do j=1 %to &nsubs;
      %syscall graycode(k, x, n);
      %put &j: k=&k x="&x" sysinfo=&sysinfo;
   %end;
%mend;
%test(3);
```

SAS writes the following output to the log:

```
nsubs=9 k=-1 x="___"
1: k=0 x="OOO" sysinfo=0
2: k=1 x="IOO" sysinfo=1
3: k=2 x="IIO" sysinfo=2
4: k=1 x="OIO" sysinfo=1
5: k=2 x="OII" sysinfo=3
6: k=3 x="III" sysinfo=1
7: k=2 x="IOI" sysinfo=2
8: k=1 x="OOI" sysinfo=1
9: k=0 x="OOO" sysinfo=3
```

See Also

Functions:

- "GRAYCODE Function" on page 523

CALL IS8601_CONVERT Routine

Converts an ISO 8601 interval to datetime and duration values, or converts datetime and duration values to an ISO 8601 interval.

Category: Date and Time

Syntax

CALL IS8601_CONVERT(*convert-from*, *convert-to*, <*from-variables*> , <*to-variables*>, <*date-time-replacements*>

Required Arguments

convert-from

specifies a keyword in single quotation marks that indicates whether the source for the conversion is an interval, a datetime and duration value, or a duration value. *convert-from* can have one of the following values:

'intvl' specifies that the source value for the conversion is an interval value.

'dt/du' specifies that the source value for the conversion is a datetime/duration value.

'du/dt' specifies that the source value for the conversion is a duration/datetime value.

'dt/dt' specifies that the source value for the conversion is a datetime/datetime value.

'du' specifies that the source value for the conversion is a duration value.

convert-to

specifies a keyword in single quotation marks that indicates the results of the conversion. *convert-to* can have one of the following values:

'intvl' specifies to create an interval value.

'dt'/du' specifies to create a datetime/duration interval.

'du/dt' specifies to create a duration/datetime interval.

'dt/dt' specifies to create a datetime/datetime interval.

'du' specifies to create a duration.

'start' specifies to create a value that is the beginning datetime or duration of an interval value.

'end' specifies to create a value that is the ending datetime or duration of an interval value.

Optional Arguments

from-variable

specifies one or two variables that contain the source value. Specify one variable for an interval value and two variables, one each, for datetime and duration values. The datetime and duration values are interval components where the first value is the beginning value of the interval and the second value is the ending value of the interval.

Requirements

An integer variable must be at least a 16-byte character variable whose value is determined by reading the value using either the $N8601B informat or the $N8601E informat, or the integer variable is an integer value that is returned from invoking the CALL ISO8601_CONVERT routine.

A datetime value must be either a SAS datetime value or an 8-byte character value that is read by the $N8601B informat or the $N8601E informat, or by invoking the CALL ISO8601_CONVERT routine.

A duration value must be a numeric value that represents the number of seconds in the duration or an 8–byte character value whose value is determined by reading the value using either the $N8601B informat or the $N8601E informat, or by invoking the CALL ISO8601_CONVERT routine.

to-variable

specifies one or two variables that contain converted values. Specify one variable for in interval value and two variables, one each, for datetime and duration values.

Requirement The interval variable must be a character variable that is 16-bytes in length or greater.

Tip The datetime and duration variables can be either numeric or character. To avoid losing precision of a numeric value, the length of a numeric variable needs to be at least eight characters. Datetime and duration character variables must be at least 16 bytes; they are padded with blank characters for values that are less than the length of the variable.

date-time-replacements
specifies date or time component values to use when a month, day, or time component is omitted from an interval, datetime, or duration value. *date-time-replacements* is specified as a series of numbers separated by a comma to represent, in this order, the year, month, day, hour, minute, or second. Components of *date-time-replacements* can be omitted only in the reverse order, seconds, minutes, hours, day, and month. If no substitute values are specified, the conversion is done using default values.

Default The following are default values for omitted date and time components:

1	month
1	day
0	hour
0	minute
0	second

Requirement A year component must be part of the datetime or duration value, and therefore is not valid in *date-time-replacements*. A comma is required as a placeholder for the year in *date-time-replacements*. For example, in the replacement value string, *',9,4,,2,'*, the first comma is a placeholder for a year value.

Example

This DATA step uses the ISO8601_CONVERT function to perform the following tasks:

- create an interval by using datetime and duration values
- create datetime and duration values from an interval that was created using the CALL IS8601_CONVERT routine
- create an interval from datetime and duration values, using replacement values for omitted date and time components in the datetime value

For easier reading, numeric variables end with an N and character variables end with a C.

```
data _null_;
   /**  declare variable length and type                          **/
   /**     Character datetime and duration values must be at least  **/
   /**  16 characters. In order not to lose precision, the      **/
   /**  numeric datetime value has a length of 8.                 **/
   length dtN duN 8 dtC duC $16 intervalC $32;
   /** assign a numeric datetime value and a      **/
   /** character duration value.                  **/
```

```
   dtN='15Sep2008:09:00:00'dt;
   duC=input('P2y3m4dT5h6m7s', $n8601b.);
   put dtN=;
   put duC=;
   /** Create an interval from a datetime and duration value   **/
   /** and format it using the ISO 8601 extended notation for  **/
   /** character values.                                       **/
   call is8601_convert('dt/du', 'intvl', dtN, duC, intervalC);
   put '**  Character interval created from datetime and duration values **/';
   put intervalC $n8601e.;
   put ' ';
   /** Create numeric datetime and duration values from an interval    **/
   /** and format it using the ISO 8601 extended notation for          **/
   /** numeric values.                                                 **/

   call is8601_convert('intvl', 'dt/du', intervalC, dtN, duN);
   put '**  Character datetime and duration created from an interval  **/';
   put dtN=;
   put duN=;
   put ' ';
   /** assign a new datetime value with omitted components             **/
   dtC=input('2009---15T10:-:-', $n8601b.);
   put '** This datetime is a character value. **';
   put dtC $n8601h.;
   put ' ';
   /** Create an interval by reading in a datetime value       **/
   /** with omitted date and time components.  Use replacement **/
   /** values for the month, minutes, and seconds.             **/

   call is8601_convert('du/dt', 'intvl', duC, dtC, intervalC,,7,,,35,45);
   put '**  Interval created using a datetime with omitted values,      **';
   put '**  inserting replacement values for month (7), minute (35)     **';
   put '**  seconds (45).                                               **';
   put intervalC $n8601e.;
   put ' ';
run;
```

The following output appears in the SAS log:

```
dtN=1537088400
duC=0002304050607FFC
**  Character interval created from datetime and duration values **/
2008-09-15T09:00:00.000/P2Y3M4DT5H6M7S
**  Character datetime and duration created from an interval  **/
dtN=1537088400
duN=71211967
** This datetime is a character value. **
2009---15T10:-:-
**  Interval created using a datetime with omitted values,      **
**  inserting replacement values for month (7), minute (35)     **
**  seconds (45).                                               **
P2Y3M4DT5H6M7S/2009-07-15T10:35:45
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.03 seconds
```

CALL LABEL Routine

Assigns a variable label to a specified character variable.

Category: Variable Control

Syntax

CALL LABEL(*variable-1*,*variable-2*);

Required Arguments

variable-1
specifies any SAS variable. If *variable-1* does not have a label, the variable name is assigned as the value of *variable-2*.

variable-2
specifies any SAS character variable. Variable labels can be up to 256 characters long. Therefore, the length of *variable-2* should be at least 256 characters to avoid truncating variable labels.

Note To conserve space, you should set the length of *variable-2* to the length of the label for *variable-1*, if it is known.

Details

The CALL LABEL routine assigns the label of the *variable-1* variable to the character variable *variable-2*.

Example: Examples

This example uses the CALL LABEL routine with array references to assign the labels of all variables in the data set OLD as values of the variable LAB in data set NEW:

```
data new;
   set old;
      /* lab is not in either array */
   length lab $256;
      /* all character variables in old */
   array abc{*} _character_;
      /* all numeric variables in old */
   array def{*} _numeric_;
   do i=1 to dim(abc);
         /* get label of character variable */
      call label(abc{i},lab);
         /* write label to an observation */
      output;
   end;
   do j=1 to dim(def);
         /* get label of numeric variable */
      call label(def{j},lab);
         /* write label to an observation */
      output;
```

```
    end;
    stop;
    keep lab;
run;
```

See Also

Functions:

- "VLABEL Function" on page 965

CALL LEXCOMB Routine

Generates all distinct combinations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

Interaction: When invoked by the %SYSCALL macro statement, CALL LEXCOMB removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 9.

Syntax

CALL LEXCOMB(*count*, *k*, *variable-1*, …, *variable-n*);

Required Arguments

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

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

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

Requirement Initialize these variables before you call the LEXCOMB routine.

Tip After calling LEXCOMB, the first *k* variables contain the values in one combination.

Details

The Basics

Use the CALL LEXCOMB routine in a loop where the first argument to CALL LEXCOMB takes each integral value from 1 to the number of distinct combinations of the nonmissing values of the variables. In each call to LEXCOMB within this loop, *k* should have the same value.

Number of Combinations

When all of the variables have nonmissing, unequal values, then the number of combinations is COMB(*n*,*k*). If the number of variables that have missing values is *m*, and all the nonmissing values are unequal, then LEXCOMB produces COMB(*n*-*m*,*k*) combinations because the missing values are omitted from the combinations.

When some of the variables have equal values, the exact number of combinations is difficult to compute. If you cannot compute the exact number of combinations, use the LEXCOMB function instead of the CALL LEXCOMB routine.

CALL LEXCOMB Processing

On the first call to the LEXCOMB routine, the following actions occur:

- The argument types and lengths are checked for consistency.
- The *m* missing values are assigned to the last *m* arguments.
- The *n*-*m* nonmissing values are assigned in ascending order to the first *n*-*m* arguments following *count*.

On subsequent calls, up to and including the last combination, the next distinct combination of the nonmissing values is generated in lexicographic order.

If you call the LEXCOMB routine with the first argument out of sequence, then the results are not useful. In particular, if you initialize the variables and then immediately call the LEXCOMB routine with a first argument of *j*, you will not get the j^{th} combination (except when *j* is 1). To get the j^{th} combination, you must call the LEXCOMB routine *j* times, with the first argument taking values from 1 through *j* in that exact order.

Using the CALL LEXCOMB Routine with Macros

You can call the LEXCOMB routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same length, but they are required to be the same type. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL LEXCOMB routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, then &SYSERR is set to zero, and &SYSINFO is set to one of the following values:

- 1 if *count*=1 and at lease one variable has a nonmissing value
- 1 if the value of *variable-1* changed
- *j* if *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j*= *i*+1
- –1 if all distinct combinations have already been generated

Comparisons

The CALL LEXCOMB routine generates all distinct combinations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order. The CALL ALLCOMB routine generates all combinations of the values of *n* variables taken *k* at a time in a minimal change order.

Examples

Example 1: Using CALL LEXCOMB in a DATA Step

The following example calls the LEXCOMB routine to generate distinct combinations in lexicographic order.

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

SAS writes the following output to the log:

```
    1   ant bee cat
    2   ant bee dog
    3   ant bee ewe
    4   ant cat dog
    5   ant cat ewe
    6   ant dog ewe
    7   bee cat dog
    8   bee cat ewe
    9   bee dog ewe
   10   cat dog ewe
```

Example 2: Using CALL LEXCOMB with Macros

The following is an example of the CALL LEXCOMB routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=baboon;
   %let x3=baboon;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=2;
   %let ncomb=%sysfunc(comb(5,&k));
      %do j=1 %to &ncomb;
         %syscall lexcomb(j, k, x1, x2, x3, x4, x5);
         %let jfmt=%qsysfunc(putn(&j, 5. ));
         %let pad=%qsysfunc(repeat(%str( ), 20-%length(&x1 &x2)));
         %put &jfmt: &x1 &x2 &pad sysinfo=&sysinfo;
         %if &sysinfo < 0 %then %let j=%eval(&ncomb+1);
      %end;
%mend;
%test
```

SAS writes the following output to the log:

```
    1: ant baboon            sysinfo=1
    2: ant hippopotamus      sysinfo=2
    3: ant zebra             sysinfo=2
```

```
4: baboon baboon          sysinfo=1
5: baboon hippopotamus    sysinfo=2
6: baboon zebra           sysinfo=2
7: hippopotamus zebra     sysinfo=1
8: hippopotamus zebra     sysinfo=-1
```

See Also

Functions:

- "LEXCOMB Function" on page 622

CALL Routines:

- "CALL ALLCOMB Routine" on page 150

CALL LEXCOMBI Routine

Generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order.

Category: Combinatorial

Syntax

CALL LEXCOMBI(*n*, *k*, *index-1*, …, *index-k*);

Required Arguments

n
is a numeric constant, variable, or expression that specifies the total number of objects.

k
is a numeric constant, variable, or expression that specifies the number of objects in each combination.

index
is a numeric variable that contains indices of the objects in the combination that is returned. Indices are integers between 1 and *n*, inclusive.

Tip If *index-1* is missing or zero, then the CALL LEXCOMBI routine initializes the indices to **`index-1=1`** through **`index-k=k`**. Otherwise, CALL LEXCOMBI creates a new combination by removing one index from the combination and adding another index.

Details

CALL LEXCOMBI Processing

Before the first call to the LEXCOMBI routine, complete one of the following tasks:

- Set *index-1* equal to zero or to a missing value.
- Initialize *index-1* through *index-k* to distinct integers between 1 and *n* inclusive.

The number of combinations of *n* objects taken *k* at a time can be computed as COMB(*n*,*k*). To generate all combinations of *n* objects taken *k* at a time, call LEXCOMBI in a loop that executes COMB(*n*,*k*) times.

Using the CALL LEXCOMBI Routine with Macros

If you call the LEXCOMBI routine from the macro processor with %SYSCALL, then you must initialize all arguments to numeric values. %SYSCALL reformats the values that are returned.

If an error occurs during the execution of the CALL LEXCOMBI routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, then &SYSERR is set to zero, and &SYSINFO is set to one of the following values:

- 1 if the value of *variable-1* changed
- *j* if *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j=i* +1
- –1 if all distinct combinations have already been generated

Comparisons

The CALL LEXCOMBI routine generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order. The CALL ALLCOMBI routine generates all combinations of the indices of *n* objects taken *k* at a time in a minimum change order.

Examples

Example 1: Using the CALL LEXCOMBI Routine with the DATA Step

The following example uses the CALL LEXCOMBI routine to generate combinations of indices in lexicographic order.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   array c[3] $3;
   array i[3];
   n=dim(x);
   k=dim(i);
   i[1]=0;
   ncomb=comb(n,k);
   do j=1 to ncomb;
      call lexcombi(n, k, of i[*]);
      do h=1 to k;
         c[h]=x[i[h]];
      end;
      put @4 j= @10 'i= ' i[*] +3 'c= ' c[*];
   end;
run;
```

SAS writes the following output to the log:

```
   j=1   i= 1 2 3    c= ant bee cat
   j=2   i= 1 2 4    c= ant bee dog
```

```
j=3   i= 1 2 5    c= ant bee ewe
j=4   i= 1 3 4    c= ant cat dog
j=5   i= 1 3 5    c= ant cat ewe
j=6   i= 1 4 5    c= ant dog ewe
j=7   i= 2 3 4    c= bee cat dog
j=8   i= 2 3 5    c= bee cat ewe
j=9   i= 2 4 5    c= bee dog ewe
j=10  i= 3 4 5    c= cat dog ewe
```

Example 2: Using the CALL LEXCOMBI Routine with Macros and Displaying the Return Code

The following example uses the CALL LEXCOMBI routine with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=0;
   %let x2=0;
   %let x3=0;
   %let n=5;
   %let k=3;
   %let ncomb=%sysfunc(comb(&n,&k));
   %do j=1 %to &ncomb+1;
      %syscall lexcombi(n,k,x1,x2,x3);
      %let jfmt=%qsysfunc(putn(&j,5.));
      %let pad=%qsysfunc(repeat(%str(),6-%length(&x1 &x2 &x3)));
      %put &jfmt: &x1 &x2 &x3 &pad sysinfo=&sysinfo;
   %end;
%mend;
%test
```

SAS writes the following output to the log:

```
    1: 1 2 3  sysinfo=1
    2: 1 2 4  sysinfo=3
    3: 1 2 5  sysinfo=3
    4: 1 3 4  sysinfo=2
    5: 1 3 5  sysinfo=3
    6: 1 4 5  sysinfo=2
    7: 2 3 4  sysinfo=1
    8: 2 3 5  sysinfo=3
    9: 2 4 5  sysinfo=2
   10: 3 4 5  sysinfo=1
   11: 3 4 5  sysinfo=-1
```

See Also

CALL Routines:

- “CALL LEXCOMB Routine” on page 177
- “CALL ALLCOMBI Routine” on page 153

CALL LEXPERK Routine

Generates all distinct permutations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

Interaction: When invoked by the %SYSCALL macro statement, CALL LEXPERK removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 9.

Syntax

CALL LEXPERK(*count*, *k*, *variable-1*, …, *variable-n*);

Required Arguments

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

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

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

Requirement Initialize these variables before you call the LEXPERK routine.

Tip After calling LEXPERK, the first *k* variables contain the values in one permutation.

Details

The Basics

Use the CALL LEXPERK routine in a loop where the first argument to CALL LEXPERK accepts each integral value from 1 to the number of distinct permutations of *k* nonmissing values of the variables. In each call to LEXPERK within this loop, *k* should have the same value.

Number of Permutations

When all of the variables have nonmissing, unequal values, the number of permutations is PERM(,*k*). If the number of variables that have missing values is *m*, and all the nonmissing values are unequal, CALL LEXPERK produces PERM(*n*-*m*,*k*) permutations because the missing values are omitted from the permutations. When some of the variables have equal values, the exact number of permutations is difficult to compute. If you cannot compute the exact number of permutations, use the LEXPERK function instead of the CALL LEXPERK routine.

CALL LEXPERK Processing

On the first call to the LEXPERK routine, the following actions occur:

- The argument types and lengths are checked for consistency.
- The *m* missing values are assigned to the last *m* arguments.
- The *n-m* nonmissing values are assigned in ascending order to the first *n-m* arguments following *count*.

On subsequent calls, up to and including the last permutation, the next distinct permutation of *k* nonmissing values is generated in lexicographic order.

If you call the LEXPERK routine with the first argument out of sequence, then the results are not useful. In particular, if you initialize the variables and then immediately call the LEXPERK routine with a first argument of *j*, you will not get the j^{th} permutation (except when *j* is 1). To get the j^{th} permutation, you must call LEXPERK *j* times, with the first argument taking values from 1 through *j* in that exact order.

Using the CALL LEXPERK Routine with Macros

You can call the LEXPERK routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same length, but they are required to be the same type. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL LEXPERK routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, then &SYSERR is set to zero, and &SYSINFO is set to one of the following values:

- 1 if *count*=1 and at least one variable has a nonmissing value
- 1 if *count*>1 and the value of *variable-1* changed
- *j* if *count*>1 and *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j*=*i*+1
- –1 if all distinct permutations were already generated

Comparisons

The CALL LEXPERK routine generates all distinct permutations of the nonmissing values of *n* variables taken *k* at a time in lexicographic order. The CALL ALLPERM routine generates all permutations of the values of several variables in a minimal change order.

Examples

Example 1: Using CALL LEXPERK in a DATA Step

The following is an example of the CALL LEXPERK routine.

```
data _null_;
   array x[5] $3 ('V' 'W' 'X' 'Y' 'Z');
   n=dim(x);
   k=3;
   nperm=perm(n,k);
   do j=1 to nperm;
      call lexperk(j, k, of x[*]);
      put j 5. +3 x1-x3;
```

```
   end;
run;
```

SAS writes the following output to the log:

```
 1   V W X
 2   V W Y
 3   V W Z
 4   V X W
 5   V X Y
 6   V X Z
 7   V Y W
 8   V Y X
 9   V Y Z
10   V Z W
11   V Z X
12   V Z Y
13   W V X
14   W V Y
15   W V Z
16   W X V
17   W X Y
18   W X Z
19   W Y V
20   W Y X
21   W Y Z
22   W Z V
23   W Z X
24   W Z Y
25   X V W
26   X V Y
27   X V Z
28   X W V
29   X W Y
30   X W Z
31   X Y V
32   X Y W
33   X Y Z
34   X Z V
35   X Z W
36   X Z Y
37   Y V W
38   Y V X
39   Y V Z
40   Y W V
41   Y W X
42   Y W Z
43   Y X V
44   Y X W
45   Y X Z
46   Y Z V
47   Y Z W
48   Y Z X
49   Z V W
50   Z V X
51   Z V Y
52   Z W V
```

```
53   Z W X
54   Z W Y
55   Z X V
56   Z X W
57   Z X Y
58   Z Y V
59   Z Y W
60   Z Y X
```

Example 2: Using CALL LEXPERK with Macros

The following is an example of the CALL LEXPERK routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
   %let x1=ant;
   %let x2=baboon;
   %let x3=baboon;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=2;
   %let nperk=%sysfunc(perm(5,&k));
   %do j=1 %to &nperk;
      %syscall lexperk(j, k, x1, x2, x3, x4, x5);
      %let jfmt=%qsysfunc(putn(&j,5.));
      %let pad=%qsysfunc(repeat(%str(),20-%length(&x1 &x2)));
      %put &jfmt: &x1 &x2 &pad sysinfo=&sysinfo;
      %if &sysinfo<0 %then %let j=%eval(&nperk+1);
   %end;
%mend;
%test
```

SAS writes the following output to the log:

```
 1: ant baboon  sysinfo=1
 2: ant hippopotamus  sysinfo=2
 3: ant zebra  sysinfo=2
 4: baboon ant  sysinfo=1
 5: baboon baboon  sysinfo=2
 6: baboon hippopotamus  sysinfo=2
 7: baboon zebra  sysinfo=2
 8: hippopotamus ant  sysinfo=1
 9: hippopotamus baboon  sysinfo=2
10: hippopotamus zebra  sysinfo=2
11: zebra ant  sysinfo=1
12: zebra baboon  sysinfo=2
13: zebra hippopotamus  sysinfo=2
14: zebra hippopotamus  sysinfo=-1
```

See Also

Functions:

- “LEXPERM Function” on page 629

CALL Routines:

- “CALL ALLPERM Routine” on page 156

- "CALL RANPERK Routine" on page 224
- "CALL RANPERM Routine" on page 226

CALL LEXPERM Routine

Generates all distinct permutations of the nonmissing values of several variables in lexicographic order.

Category: Combinatorial

Interaction: When invoked by the %SYSCALL macro statement, CALL LEXPERM removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 9.

Syntax

CALL LEXPERM(*count*, *variable-1* <, …, *variable-N*>);

Required Arguments

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

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

Requirement Initialize these variables before you call the LEXPERM routine.

Details

Determine the Number of Distinct Permutations

These variables are defined for use in the equation that follows:

N
specifies the number of variables that are being permuted—that is, the number of arguments minus one.

M
specifies the number of missing values among the variables that are being permuted.

d
specifies the number of distinct nonmissing values among the arguments.

N_i
for i=1 through i=d, N_i specifies the number of instances of the i*th* distinct value.

The number of distinct permutations of nonmissing values of the arguments is expressed as follows:

$$P = \frac{(N_1 + N_2 + \ldots + N_d)!}{N_1!\, N_2! \ldots N_d!} <= N!$$

CALL LEXPERM Processing

Use the CALL LEXPERM routine in a loop where the argument *count* accepts each integral value from 1 to P. You do not need to compute P provided you exit the loop when CALL LEXPERM returns a value that is less than zero.

For 1=*count*<P, the following actions occur:

- The argument types and lengths are checked for consistency.
- The M missing values are assigned to the last M arguments.
- The N-M nonmissing values are assigned in ascending order to the first N-M arguments following *count*.
- CALL LEXPERM returns 1.

For 1<*count*<=P, the following actions occur:

- The next distinct permutation of the nonmissing values is generated in lexicographic order.
- If *variable-1* through *variable-I* did not change, but *variable-J* did change, where J=I +1, then CALL LEXPERM returns J.

For *count*>P, CALL LEXPERM returns –1.

If the CALL LEXPERM routine is executed with the first argument out of sequence, the results might not be useful. In particular, if you initialize the variables and then immediately execute CALL LEXPERM with a first argument of K, you will not get the K*th* permutation (except when K is 1). To get the K*th* permutation, you must execute CALL LEXPERM K times, with the first argument accepting values from 1 through K in that exact order.

Using the CALL LEXPERM Routine with Macros

You can call the LEXPERM routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same length, but they must be the same type. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL LEXPERM routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than –100.

If there are no errors, then &SYSERR is set to zero, and &SYSINFO is set to one of the following values:

- 1 if 1=*count*<P
- 1 if 1<*count*<=P and the value of *variable-1* changed
- J if 1<*count*<=P and *variable-1* through *variable-I* did not change, but *variable-J* did change, where J=I+1
- –1 if *count*>P

Comparisons

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

- ALLPERM generates all *possible* permutations of the values, *missing or non-missing*, of several variables. Each permutation is formed from the previous permutation by interchanging two consecutive values.

- LEXPERM generates all *distinct* permutations of the *non-missing* values of several variables. The permutations are generated in lexicographic order.
- LEXPERK generates all *distinct* permutations of K of the *non-missing* values of N variables. The permutations are generated in lexicographic order.

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

Examples

Example 1: Using CALL LEXPERM in a DATA Step

The following example uses the DATA step to generate all distinct permutations of the nonmissing values of several variables in lexicographic order.

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

SAS writes the following output to the log:

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

Example 2: Using CALL LEXPERM with Macros

The following is an example of the CALL LEXPERM routine that is used with macros. The output includes values for the %SYSINFO macro.

```
%macro test;
```

```
  %let x1=ant;
  %let x2=baboon;
  %let x3=baboon;
  %let x4=hippopotamus;
  %let n=4;
  %let nperm=%sysfunc(perm(4));
  %do j=1 %to &nperm;
     %syscall lexperm(j,x1,x2,x3,x4);
     %let jfmt=%qsysfunc(putn(&j,5.));
     %put &jfmt: &x1 &x2 &x3 &x4 sysinfo=&sysinfo;
     %if &sysinfo<0 %then %let j=%eval(&nperm+1);
  %end;
%mend;

%test;
```

SAS writes the following output to the log:

```
    1: ant baboon baboon hippopotamus sysinfo=1
    2: ant baboon hippopotamus baboon sysinfo=3
    3: ant hippopotamus baboon baboon sysinfo=2
    4: baboon ant baboon hippopotamus sysinfo=1
    5: baboon ant hippopotamus baboon sysinfo=3
    6: baboon baboon ant hippopotamus sysinfo=2
    7: baboon baboon hippopotamus ant sysinfo=3
    8: baboon hippopotamus ant baboon sysinfo=2
    9: baboon hippopotamus baboon ant sysinfo=3
   10: hippopotamus ant baboon baboon sysinfo=1
   11: hippopotamus baboon ant baboon sysinfo=2
   12: hippopotamus baboon baboon ant sysinfo=3
   13: hippopotamus baboon baboon ant sysinfo=-1
```

See Also

Functions:

- “LEXPERM Function” on page 629
- “LEXPERK Function” on page 627

CALL Routines:

- “CALL ALLPERM Routine” on page 156
- “CALL RANPERK Routine” on page 224
- “CALL RANPERM Routine” on page 226

CALL LOGISTIC Routine

Applies the logistic function to each argument.

Category: Mathematical

Syntax

CALL LOGISTIC(*argument*<, *argument*, ...>);

Required Argument

argument
is a numeric variable.

Restriction The CALL LOGISTIC routine only accepts variables as valid arguments. Do not use a constant or a SAS expression because the CALL routine is unable to update these arguments.

Details

The CALL LOGISTIC routine replaces each argument by the logistic value of that argument. For example x_j is replaced by

$$\frac{\varepsilon^{x_j}}{1+\varepsilon^{x_j}}$$

If any argument contains a missing value, then CALL LOGISTIC returns missing values for all the arguments.

Example

The following SAS statements produce these results.

SAS Statement	Result

`x=0.5;`
`y=-0.5;`
`call logistic(x,y);`
`put x= y=;`
	`x=0.6224593312 y=0.3775406688`

CALL MISSING Routine

Assigns missing values to the specified character or numeric variables.

Category: Character

Syntax

CALL MISSING(*varname1*<, *varname2*, ...>);

Required Argument

varname
specifies the name of SAS character or numeric variables.

Details

The CALL MISSING routine assigns an ordinary numeric missing value (.) to each numeric variable in the argument list.

The CALL MISSING routine assigns a character missing value (a blank) to each character variable in the argument list. If the current length of the character variable equals the maximum length, the current length is not changed. Otherwise, the current length is set to 1.

You can mix character and numeric variables in the argument list.

Comparisons

The MISSING function checks whether the argument has a missing value but does not change the value of the argument.

Example

The following SAS statements produce these results.

SAS Statement	Result

`prod='shoes';`
`invty=7498;`
`sales=23759;`
`call missing(sales);`
`put prod= invty= sales=;`
	`prod=shoes invty=7498 sales=.`

`prod='shoes';`
`invty=7498;`
`sales=23759;`
`call missing(prod,invty);`
`put prod= invty= sales=;`
	`prod= invty=. sales=23759`

`prod='shoes';`
`invty=7498;`
`sales=23759;`
`call missing(of _all_);`
`put prod= invty= sales=;`
	`prod= invty=. sales=.`

See Also

Functions:

- “MISSING Function” on page 662
- “How to Set Variable Values to Missing in a DATA Step” in Chapter 5 of *SAS Language Reference: Concepts*

CALL MODULE Routine

Calls an external routine without any return code.

Category: External Routines

Syntax

CALL MODULE(<*cntl-string*,> *module-name*<,*argument-1, ..., argument-n*>);

Required Argument

module-name
is the name of the external module to use.

Optional Arguments

cntl-string
is an optional control string whose first character must be an asterisk (*), followed by any combination of the following characters:

I prints the hexadecimal representations of all arguments to the CALL MODULE routine. You can use this option to help diagnose problems caused by incorrect arguments or attribute tables. If you specify the I option, the E option is implied.

E prints detailed error messages. Without the E option (or the I option, which supersedes it), the only error message that the CALL MODULE routine generates is "Invalid argument to function," which is usually not enough information to determine the cause of the error. The E option is useful for a production environment, while the I option is preferable for a development or debugging environment.

H provides brief help information about the syntax of the CALL MODULE routine, the attribute file format, and suggested SAS formats and informats.

argument
is one or more arguments to pass to the requested routine.

CAUTION:
Use the correct arguments and attributes. If you use incorrect arguments or attributes, you can cause the SAS System, and possibly your operating system, to fail.

Details

The CALL MODULE routine executes a routine *module-name* that resides in an external library with the specified arguments.

CALL MODULE builds a parameter list using the information in the arguments and a routine description and argument attribute table that you define in a separate file. The attribute table is a sequential text file that contains descriptions of the routines that you can invoke with the CALL MODULE routine. The purpose of the table is to define how CALL MODULE should interpret its supplied arguments when it builds a parameter list to pass to the external routine. The attribute table should contain a description for each external routine that you intend to call, and descriptions of each argument associated with that routine.

Before you invoke CALL MODULE, you must define the fileref of SASCBTBL to point to the external file that contains the attribute table. You can name the file whatever you want when you create it. This way, you can use SAS variables and formats as arguments to CALL MODULE and ensure that these arguments are properly converted before

being passed to the external routine. If you do not define this fileref, CALL MODULE calls the requested routine without altering the arguments.

CAUTION:

Using the CALL MODULE routine without a defined attribute table can cause the SAS System to fail or force you to reset your computer. You need to use an attribute table for all external functions that you want to invoke.

Comparisons

The two CALL routines and four functions share identical syntax:

- The MODULEN and MODULEC functions return a number and a character, respectively, while the routine CALL MODULE does not return a value.
- The CALL MODULEI routine and the functions MODULEIC and MODULEIN permit vector and matrix arguments. Their return values are scalar. You can invoke CALL MODULEI, MODULEIC, and MODULEIN only from the IML procedure.

Examples

Example 1: Using the CALL MODULE Routine

This example calls the **xyz** routine. Use the following attribute table:

```
routine xyz minarg=2 maxarg=2;
arg 1 input num byvalue format=ib4.;
arg 2 output char format=$char10.;
```

The following is the sample SAS code that calls the **xyz** function:

```
data _null_;
   call module('xyz',1,x);
run;
```

Example 2: Using the MODULEIN Function in the IML Procedure

This example invokes the **changi** routine from the TRYMOD.DLL module on a Windows platform. Use the following attribute table:

```
routine changi module=trymod returns=long;
arg 1 input num format=ib4. byvalue;
arg 2 update num format=ib4.;
```

The following PROC IML code calls the **changi** function:

```
proc iml;
   x1=J(4,5,0);
   do i=1 to 4;
      do j=1 to 5;
         x1[i,j]=i*10+j+3;
      end;
   end;
   y1=x1;
   x2=x1;
   y2=y1;
   rc=modulein('*i','changi',6,x2);
```

Example 3: Using the MODULEN Function

This example calls the **Beep** routine, which is part of the Win32 API in the KERNEL32 Dynamic Link Library on a Windows platform. Use the following attribute table:

```
routine Beep
   minarg=2
   maxarg=2
   stackpop=called
   callseq=byvalue
   module=kernel32;
arg 1 num format=pib4.;
arg 2 num format=pib4.;
```

Assume that you name the attribute table file 'myatttbl.dat'. The following is the sample SAS code that calls the **Beep** function:

```
filename sascbtbl 'myatttbl.dat';
data _null_;
   rc=modulen("*e","Beep",1380,1000);
run;
```

The previous code causes the computer speaker to beep.

See Also

Functions:

- "MODULEC Function" on page 666
- "MODULEN Function" on page 667

CALL POKE Routine

Writes a value directly into memory on a 32-bit platform.

Category: Special

Restriction: Use on 32-bit platforms only.

Syntax

CALL POKE(*source*,*pointer*<,*length*><,*floating-point*>);

Required Arguments

source
specifies a constant, variable, or expression that contains a value to write into memory.

pointer
specifies a numeric expression that contains the virtual address of the data that the CALL POKE routine alters.

Optional Arguments

length
specifies a numeric constant, variable, or expression that contains the number of bytes to write from the *source* to the address that is indicated by *pointer*. If you omit *length*, the action that the CALL POKE routine takes depends on whether *source* is a character value or a numeric value:

- If *source* is a character value, the CALL POKE routine copies the entire value of *source* to the specified memory location.
- If *source* is a numeric value, the CALL POKE routine converts *source* into a long integer and writes into memory the number of bytes that constitute a pointer.

z/OS Specifics
Under z/OS, pointers are 3 or 4 bytes long, depending on the situation.

floating-point
specifies that the value of *source* is stored as a floating-point number. The value of *floating-point* can be any number.

Tip If you do not use the *floating-point* argument, then *source* is stored as an integer value.

Details

CAUTION:
The CALL POKE routine is intended only for experienced programmers in specific cases. If you plan to use this routine, use extreme care both in your programming and in your typing. Writing directly into memory can cause devastating problems. This routine bypasses the normal safeguards that prevent you from destroying a vital element in your SAS session or in another piece of software that is active at the time.

If you do not have access to the memory location that you specify, the CALL POKE routine returns an "Invalid argument" error.

You cannot use the CALL POKE routine on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use CALL POKE, change the applications and use CALL POKELONG instead. You can use CALL POKELONG on both 32–bit and 64–bit platforms.

If you use the fourth argument, then a floating-point number is assumed to be the value that is stored. If you do not use the fourth argument, then an integer value is assumed to be stored.

See Also

Functions:

- “ADDR Function” on page 92
- “PEEK Function” on page 738
- “PEEKC Function” on page 739

CALL Routines:

- “CALL POKELONG Routine” on page 197

CALL POKELONG Routine

Writes a value directly into memory on 32-bit and 64-bit platforms.

Category: Special

Syntax

CALL POKELONG(*source*,*pointer*<,*length*><,*floating-point*>)

Required Arguments

source
specifies a character constant, variable, or expression that contains a value to write into memory.

pointer
specifies a character string that contains the virtual address of the data that the CALL POKELONG routine alters.

Optional Arguments

length
specifies a numeric SAS expression that contains the number of bytes to write from the *source* to the address that is indicated by the *pointer*. If you omit *length*, the CALL POKELONG routine copies the entire value of *source* to the specified memory location.

floating-point
specifies that the value of *source* is stored as a floating-point number. The value of *floating-point* can be any number.

Tip If you do not use the *floating-point* argument, then *source* is stored as an integer value.

Details

CAUTION:
The CALL POKELONG routine is intended only for experienced programmers in specific cases. If you plan to use this routine, use extreme care both in your programming and in your typing. *Writing directly into memory can cause devastating problems.* It bypasses the normal safeguards that prevent you from destroying a vital element in your SAS session or in another piece of software that is active at the time.

If you do not have access to the memory location that you specify, the CALL POKELONG routine returns an "Invalid argument" error.

If you use the fourth argument, then a floating-point number is assumed to be the value that is stored. If you do not use the fourth argument, then an integer value is assumed to be stored.

See Also

CALL Routines:

- "CALL POKE Routine" on page 195

CALL PRXCHANGE Routine

Performs a pattern-matching replacement.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Interaction: When invoked by the %SYSCALL macro statement, CALL PRXCHANGE removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 9.

Syntax

CALL PRXCHANGE (*regular-expression-id*, *times*, *old-string* <, *new-string* <, *result-length* <, *truncation-value* <, *number-of-changes*> > > >);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned from the PRXPARSE function.

times
is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern.

Tip If the value of *times* is -1, then all matching patterns are replaced.

old-string
specifies the character expression on which to perform a search and replace.

Tip All changes are made to *old-string* if you do not use the *new-string* argument.

Optional Arguments

new-string
specifies a character variable in which to place the results of the change to *old-string*.

Tip If you use the *new-string* argument in the call to the PRXCHANGE routine, then *old-string* is not modified.

result-length
is a numeric variable with a return value that is the number of characters that are copied into the result.

Tip Trailing blanks in the value of *old-string* are not copied to *new-string*, and are therefore not included as part of the length in *result-length*.

truncation-value
is a numeric variable with a returned value that is either 0 or 1, depending on the result of the change operation:

0 if the entire replacement result is not longer than the length of *new-string*.

1 if the entire replacement result is longer than the length of *new-string*.

number-of-changes
is a numeric variable with a returned value that is the total number of replacements that were made. If the result is truncated when it is placed into *new-string*, the value of *number-of-changes* is not changed.

Details

The CALL PRXCHANGE routine matches and replaces a pattern. If the value of *times* is -1, the replacement is performed as many times as possible.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 42.

Comparisons

The CALL PRXCHANGE routine is similar to the PRXCHANGE function except that the CALL routine returns the value of the pattern matching replacement as one of its parameters instead of as a return argument.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following example replaces all occurrences of cat, rat, or bat with the value TREE.

```
data _null_;
      /* Use a pattern to replace all occurrences of cat,      */
      /* rat, or bat with the value TREE.                      */
   length text $ 46;
   RegularExpressionId = prxparse('s/[crb]at/tree/');
   text = 'The woods have a bat, cat, bat, and a rat!';
      /* Use CALL PRXCHANGE to perform the search and replace. */
      /* Because the argument times has a value of -1, the     */
      /* replacement is performed as many times as possible.   */
   call prxchange(RegularExpressionId, -1, text);
   put text;
run;
```

SAS writes the following line to the log:

```
The woods have a tree, tree, tree, and a tree!
```

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXPAREN Function" on page 784

- "PRXMATCH Function" on page 780
- "PRXPARSE Function" on page 786
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202
- "CALL PRXNEXT Routine" on page 203
- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208

CALL PRXDEBUG Routine

Enables Perl regular expressions in a DATA step to send debugging output to the SAS log.

Category: Character String Matching

Restriction: Use with the CALL PRXCHANGE, CALL PRXFREE, CALL PRXNEXT, CALL PRXPOSN, CALL PRXSUBSTR, PRXPARSE, PRXPAREN, and PRXMATCH functions and CALL routines. The PRXPARSE function is not DBCS compatible.

Syntax

CALL PRXDEBUG (*on-off*);

Required Argument

on-off
specifies a numeric constant, variable, or expression. If the value of *on-off* is positive and non-zero, then debugging is turned on. If the value of *on-off* is zero, then debugging is turned off.

Details

The CALL PRXDEBUG routine provides information about how a Perl regular expression is compiled, and about which steps are taken when a pattern is matched to a character value.

You can turn debugging on and off multiple times in your program if you want to see debugging output for particular Perl regular expression function calls.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 42.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following example produces debugging output.

```
data _null_;
      /* Turn the debugging option on. */
   call prxdebug(1);
   putlog 'PRXPARSE: ';
   re = prxparse('/[bc]d(ef*g)+h[ij]k$/');
   putlog 'PRXMATCH: ';
   pos = prxmatch(re, 'abcdefg_gh_');
      /* Turn the debugging option off. */
   call prxdebug(0);
run;
```

The following lines are written to the SAS log.

Log 2.1 *SAS Log Results from CALL PRXDEBUG*

```
PRXPARSE:
Compiling REx '[bc]d(ef*g)+h[ij]k$'  1
size 41 first at 1  2
rarest char g at 0  5
rarest char d at 0
   1: ANYOF[bc](10)  3
  10: EXACT <d>(12)
  12: CURLYX[0] {1,32767}(26)
  14:   OPEN1(16)
  16:     EXACT <e>(18)
  18:     STAR(21)
  19:       EXACT <f>(0)
  21:     EXACT <g>(23)
  23:   CLOSE1(25)
  25:   WHILEM[1/1](0)
  26: NOTHING(27)
  27: EXACT <h>(29)
  29: ANYOF[ij](38)
  38: EXACT <k>(40)
  40: EOL(41)
  41: END(0)
anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)  4
stclass 'ANYOF[bc]' minlen 7  6
PRXMATCH:
Guessing start of match, REx '[bc]d(ef*g)+h[ij]k$' against 'abcdefg_gh_'...
Did not find floating substr 'gh'...
Match rejected by optimizer
```

The following items correspond to the lines that are numbered in the SAS log that is shown above.

1 This line shows the precompiled form of the Perl regular expression.

2 Size specifies a value in arbitrary units of the compiled form of the Perl regular expression. 41 is the label ID of the first node that performs a match.

3 This line begins a list of program nodes in compiled form for regular expressions.

4 These two lines provide optimizer information. In the example above, the optimizer found that the match should contain the substring **de** at offset 1, and the substring **gh** at an offset between 3 and infinity. To rule out a pattern match quickly, Perl checks substring **gh** before it checks substring **de**.

The optimizer might use the information that the match begins at the *first* ID (line 2), with a character class (line 5), and cannot be shorter than seven characters (line 6).

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXPAREN Function" on page 784
- "PRXMATCH Function" on page 780
- "PRXPARSE Function" on page 786
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXFREE Routine" on page 202
- "CALL PRXNEXT Routine" on page 203
- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208

CALL PRXFREE Routine

Frees memory that was allocated for a Perl regular expression.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Syntax

CALL PRXFREE (*regular-expression-id*);

Required Argument

regular-expression-id
specifies a numeric variable with a value that is the identification number that is returned by the PRXPARSE function. *regular-expression-id* is set to missing if the call to the PRXFREE routine occurs without error.

Details

The CALL PRXFREE routine frees unneeded resources that were allocated for a Perl regular expression.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 42.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXPAREN Function" on page 784
- "PRXPAREN Function" on page 784
- "PRXPARSE Function" on page 786
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXNEXT Routine" on page 203
- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208
- "CALL PRXCHANGE Routine" on page 198

CALL PRXNEXT Routine

Returns the position and length of a substring that matches a pattern, and iterates over multiple matches within one string.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Interaction: When invoked by the %SYSCALL macro statement, CALL PRXNEXT removes the quotation marks from arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 9.

Syntax

CALL PRXNEXT (*regular-expression-id*, *start*, *stop*, *source*, *position*, *length*);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is the identification number that is returned by the PRXPARSE function.

start
is a numeric variable that specifies the position at which to start the pattern matching in *source*. If the match is successful, CALL PRXNEXT returns a value of *position* + MAX(1, *length*). If the match is not successful, the value of *start* is not changed.

stop
is a numeric constant, variable, or expression that specifies the last character to use in *source*. If *stop* is -1, then the last character is the last non-blank character in *source*.

source
specifies a character constant, variable, or expression that you want to search.

position
is a numeric variable with a returned value that is the position in *source* at which the pattern begins. If no match is found, CALL PRXNEXT returns zero.

length
is a numeric variable with a returned value that is the length of the string that is matched by the pattern. If no match is found, CALL PRXNEXT returns zero.

Details

The CALL PRXNEXT routine searches the variable *source* with a pattern. It returns the position and length of a pattern match that is located between the *start* and the *stop* positions in *source*. Because the value of the *start* parameter is updated to be the position of the next character that follows a match, CALL PRXNEXT enables you to search a string for a pattern multiple times in succession.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 42.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following example finds all instances of cat, rat, or bat in a text string.

```
data _null_;
   ExpressionID = prxparse('/[crb]at/');
   text = 'The woods have a bat, cat, and a rat!';
   start = 1;
   stop = length(text);
      /* Use PRXNEXT to find the first instance of the pattern, */
      /* then use DO WHILE to find all further instances.       */
      /* PRXNEXT changes the start parameter so that searching  */
      /* begins again after the last match.                     */
   call prxnext(ExpressionID, start, stop, text, position, length);
      do while (position > 0);
         found = substr(text, position, length);
         put found= position= length=;
         call prxnext(ExpressionID, start, stop, text, position, length);
      end;
run;
```

The following lines are written to the SAS log:

```
found=bat position=18 length=3
found=cat position=23 length=3
found=rat position=34 length=3
```

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXPAREN Function" on page 784
- "PRXMATCH Function" on page 780
- "PRXPARSE Function" on page 786
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202
- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208

CALL PRXPOSN Routine

Returns the start position and length for a capture buffer.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Syntax

CALL PRXPOSN (*regular-expression-id*, *capture-buffer*, *start* <, *length*>);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned by the PRXPARSE function.

capture-buffer
is a numeric constant, variable, or expression with a value that identifies the capture buffer from which to retrieve the start position and length:

- If the value of *capture-buffer* is zero, CALL PRXPOSN returns the start position and length of the entire match.
- If the value of *capture-buffer* is between 1 and the number of open parentheses, CALL PRXPOSN returns the start position and length for that capture buffer.
- If the value of *capture-buffer* is greater than the number of open parentheses, CALL PRXPOSN returns missing values for the start position and length.

start
is a numeric variable with a returned value that is the position at which the capture buffer is found:

- If the value of *capture-buffer* is not found, CALL PRXPOSN returns a zero value for the start position.
- If the value of *capture-buffer* is greater than the number of open parentheses in the pattern, CALL PRXPOSN returns a missing value for the start position.

Optional Argument

length
is a numeric variable with a returned value that is the pattern length of the previous pattern match:

- If the pattern match is not found, CALL PRXPOSN returns a zero value for the length.
- If the value of *capture-buffer* is greater than the number of open parentheses in the pattern, CALL PRXPOSN returns a missing value for length.

Details

The CALL PRXPOSN routine uses the results of PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT to return a capture buffer. A match must be found by one of these functions for the CALL PRXPOSN routine to return meaningful information.

A capture buffer is part of a match, enclosed in parentheses, that is specified in a regular expression. CALL PRXPOSN does not return the text for the capture buffer directly. It requires a call to the SUBSTR function to return the text.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 42.

Comparisons

The CALL PRXPOSN routine is similar to the PRXPOSN function, except that CALL PRXPOSN returns the position and length of the capture buffer rather than the capture buffer itself.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Examples

Example 1: Finding Submatches within a Match

The following example searches a regular expression and calls the PRXPOSN routine to find the position and length of three submatches.

```
data _null_;
   patternID = prxparse('/(\d\d):(\d\d)(am|pm)/');
   text = 'The time is 09:56am.';
   if prxmatch(patternID, text) then do;
      call prxposn(patternID, 1, position, length);
      hour = substr(text, position, length);
      call prxposn(patternID, 2, position, length);
      minute = substr(text, position, length);
      call prxposn(patternID, 3, position, length);
      ampm = substr(text, position, length);
```

```
      put hour= minute= ampm=;
      put text=;
   end;
run;
```

SAS writes the following lines to the log:

```
hour=09 minute=56 ampm=am
text=The time is 09:56am.
```

Example 2: Parsing Time Data

The following example parses time data and writes the results to the SAS log.

```
data _null_;
   if _N_ = 1 then
   do;
      retain patternID;
      pattern = "/(\d+):(\d\d)(?:\.(\d+))?/";
      patternID = prxparse(pattern);
   end;

   array match[3] $ 8;
   input minsec $80.;
   position = prxmatch(patternID, minsec);
   if position ^= 0 then
   do;
      do i = 1 to prxparen(patternID);
         call prxposn(patternID, i, start, length);
         if start ^= 0 then
            match[i] = substr(minsec, start, length);
      end;
      put match[1] "minutes, " match[2] "seconds" @;
      if ^missing(match[3]) then
         put ", " match[3] "milliseconds";
   end;
   datalines;
14:56.456
45:32
;
```

SAS writes the following lines to the log:

```
   14 minutes, 56 seconds, 456 milliseconds
   45 minutes, 32 seconds
```

See Also

Functions:

- “PRXCHANGE Function” on page 775
- “PRXPAREN Function” on page 784
- “PRXMATCH Function” on page 780
- “PRXPARSE Function” on page 786
- “PRXPOSN Function” on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202
- "CALL PRXNEXT Routine" on page 203
- "CALL PRXSUBSTR Routine" on page 208

CALL PRXSUBSTR Routine

Returns the position and length of a substring that matches a pattern.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Interaction: When invoked by the %SYSCALL macro statement, CALL PRXSUBSTR removes the quotation marks from its arguments. For more information, see Using CALL Routines and the %SYSCALL Macro Statement on page 9.

Syntax

CALL PRXSUBSTR (*regular-expression-id*, *source*, *position* <, *length*>);

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is an identification number that is returned by the PRXPARSE function.

source
specifies a character constant, variable, or expression that you want to search.

position
is a numeric variable with a returned value that is the position in *source* where the pattern begins. If no match is found, CALL PRXSUBSTR returns zero.

Optional Argument

length
is a numeric variable with a returned value that is the length of the substring that is matched by the pattern. If no match is found, CALL PRXSUBSTR returns zero.

Details

The CALL PRXSUBSTR routine searches the variable *source* with the pattern from PRXPARSE, returns the position of the start of the string, and if specified, returns the length of the string that is matched. By default, when a pattern matches more than one character that begins at a specific position, CALL PRXSUBSTR selects the longest match.

For more information about pattern matching, see Pattern Matching Using Perl Regular Expressions (PRX) on page 42.

Comparisons

CALL PRXSUBSTR performs the same matching as PRXMATCH, but CALL PRXSUBSTR additionally enables you to use the *length* argument to receive more information about the match.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Examples

Example 1: Finding the Position and Length of a Substring

The following example searches a string for a substring, and returns its position and length in the string.

```
data _null_;
      /* Use PRXPARSE to compile the Perl regular expression. */
   patternID = prxparse('/world/');
      /* Use PRXSUBSTR to find the position and length of the string. */
   call prxsubstr(patternID, 'Hello world!', position, length);
   put position= length=;
run;
```

The following line is written to the SAS log:

```
   position=7 length=5
```

Example 2: Finding a Match in a Substring

The following example searches for addresses that contain avenue, drive, or road, and extracts the text that was found.

```
data _null_;
   if _N_ = 1 then
   do;
      retain ExpressionID;
         /* The i option specifies a case insensitive search. */
      pattern = "/ave|avenue|dr|drive|rd|road/i";
      ExpressionID = prxparse(pattern);
   end;
   input street $80.;
   call prxsubstr(ExpressionID, street, position, length);
   if position ^= 0 then
   do;
      match = substr(street, position, length);
      put match:$QUOTE. "found in " street:$QUOTE.;
   end;
   datalines;
153 First Street
6789 64th Ave
4 Moritz Road
7493 Wilkes Place
;
run;
```

The following lines are written to the SAS log:

```
"Ave" found in "6789 64th Ave"
"Road" found in "4 Moritz Road"
```

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXPAREN Function" on page 784
- "PRXMATCH Function" on page 780
- "PRXPARSE Function" on page 786
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202
- "CALL PRXNEXT Routine" on page 203
- "CALL PRXPOSN Routine" on page 205

CALL RANBIN Routine

Returns a random variate from a binomial distribution.

Category: Random Number

Syntax

CALL RANBIN(*seed*,*n*,*p*,*x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANBIN is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See Seed Values on page 11 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 15 for more information about seed values

n

is an integer number of independent Bernoulli trials.

Range $n > 0$

p
is a numeric probability of success parameter.

Range $0<p<1$

x
is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANBIN is executed.

Details

The CALL RANBIN routine updates *seed* and returns a variate *x* that is generated from a binomial distribution with mean *np* and variance *np*(1–*p*). If $n \leq 50$, $np \leq 5$, or $n(1-p) \leq 5$, SAS uses an inverse transform method applied to a RANUNI uniform variate. If $n>50$, $np>5$, and $n(1-p)>5$, SAS uses the normal approximation to the binomial distribution. In that case, the Box-Muller transformation of RANUNI uniform variates is used.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

For a discussion and example of an effective use of the random number CALL routines, see Starting, Stopping, and Restarting a Stream on page 26.

Comparisons

The CALL RANBIN routine gives greater control of the seed and random number streams than does the RANBIN function.

Example

The following example uses the CALL RANBIN routine:

```
data u1 (keep = x);
   seed = 104;
   do i = 1 to 5;
      call ranbin(seed, 2000, 0.2 ,x);
      output;
   end;
   call symputx('seed', seed);
run;

data u2 (keep = x);
   seed = &seed;
   do i = 1 to 5;
      call ranbin(seed, 2000, 0.2 ,x);
      output;
   end;
run;

data all;
   set u1 u2;
   z = ranbin(104, 2000, 0.2);
run;

proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
```

```
run;
```

Display 2.2 *Output from the CALL RANBIN Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	423	423
2	418	418
3	403	403
4	394	394
5	429	429
6	369	369
7	413	413
8	417	417
9	400	400
10	383	383

See Also

Functions:

- “RAND Function” on page 806
- “RANBIN Function” on page 804

CALL RANCAU Routine

Returns a random variate from a Cauchy distribution.

Category: Random Number

Syntax

CALL RANCAU(*seed*,*x*);

Required Arguments

seed
is the seed value. A new value for *seed* is returned each time CALL RANCAU is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See Seed Values on page 11 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 15 for more information about seed values

x
is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANCAU is executed.

Details

The CALL RANCAU routine updates *seed* and returns a variate *x* that is generated from a Cauchy distribution that has a location parameter of 0 and scale parameter of 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

An acceptance-rejection procedure applied to RANUNI uniform variates is used. If u and v are independent uniform $(-1/2, 1/2)$ variables and $u^2+v^2 \leq 1/4$, then u/v is a Cauchy variate.

For a discussion and example of an effective use of the random number CALL routines, see Starting, Stopping, and Restarting a Stream on page 26.

Comparisons

The CALL RANCAU routine gives greater control of the seed and random number streams than does the RANCAU function.

Example

This example uses the CALL RANCAU routine.

```
data case;
   retain Seed_1 Seed_2 Seed_3 45;
   do i=1 to 10;
      call rancau(Seed_1,X1);
      call rancau(Seed_2,X2);
      X3=rancau(Seed_3);
      if i=5 then
          do;
             Seed_2=18;
             Seed_3=18;
          end;
      output;
   end;
run;

proc print;
   id i;
   var Seed_1-Seed_3 X1-X3;
run;
```

Display 2.3 *Output from the CALL RANCAU Routine*

The SAS System

i	Seed_1	Seed_2	Seed_3	X1	X2	X3
1	1404437564	1404437564	45	-1.14736	-1.14736	-1.14736
2	1326029789	1326029789	45	-0.23735	-0.23735	-0.23735
3	1988843719	1988843719	45	-0.15474	-0.15474	-0.15474
4	1233028129	1233028129	45	4.97935	4.97935	4.97935
5	50049159	18	18	0.20402	0.20402	0.20402
6	802575599	991271755	18	3.43645	4.44427	3.43645
7	1233458739	1437043694	18	6.32808	-1.79200	6.32808
8	52428589	959908645	18	0.18815	-1.67610	0.18815
9	1216356463	1225034217	18	0.80689	3.88391	0.80689
10	1711885541	425626811	18	0.92971	-1.31309	0.92971

The following is another example of the CALL RANCAU routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call rancau(seed, X);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call rancau(seed, X);
      output;
   end;
run;

data all;
   set u1 u2;
   z = rancau(104);
run;

proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.4 *Output from the CALL RANCAU Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	-0.6780	-0.6780
2	0.1712	0.1712
3	1.1372	1.1372
4	0.1478	0.1478
5	16.6536	16.6536
6	0.0747	0.0747
7	-0.5872	-0.5872
8	1.4713	1.4713
9	0.1792	0.1792
10	-0.0473	-0.0473

See Also

Functions:

- "RAND Function" on page 806
- "RANCAU Function" on page 805

CALL RANCOMB Routine

Permutes the values of the arguments, and returns a random combination of *k* out of *n* values.

Category: Combinatorial

Syntax

CALL RANCOMB(*seed*, *k*, *variable-1*<, *variable–2, ...*>);

Required Arguments

seed
is a numeric variable that contains the random number seed. For more information about seeds, see "Seed Values" on page 11.

k
is the number of values that you want to have in the random combination.

variable
specifies all numeric variables, or all character variables that have the same length. *K* values of these variables are randomly permuted.

Details

The Basics

If there are *n* variables, CALL RANCOMB permutes the values of the variables in such a way that the first *k* values are sorted in ascending order and form a random combination of *k* out of the *n* values. That is, all $n!/(k!(n-k)!)$ combinations of *k* out of the *n* values are equally likely to be returned as the first *k* values.

If an error occurs during the execution of the CALL RANCOMB routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than -100.

If there are no errors, then &SYSERR and &SYSINFO are set to zero.

Using CALL RANCOMB with Macros

You can call the CALL RANCOMB routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. However, if the first *k* values that are returned include both character and numeric values, then those values are not sorted. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

Examples

Example 1: Using CALL RANCOMB in a DATA Step

The following example shows how to generate random combinations of given values by using the CALL RANCOMB routine.

```
data _null_;
   array x x1-x5 (1 2 3 4 5);
   seed = 1234567890123;
   do n=1 to 10;
      call rancomb(seed, 3, of x1-x5);
      put seed= @20 ' x= ' x1-x3;
   end;
run;
```

Log 2.2 *Log Output from Using the CALL RANCOMB Routine in a DATA Step*

```
seed=1332351321     x= 2 4 5
seed=829042065      x= 1 3 4
seed=767738639      x= 2 3 5
seed=1280236105     x= 2 4 5
seed=670350431      x= 1 2 5
seed=1956939964     x= 2 3 4
seed=353939815      x= 1 3 4
seed=1996660805     x= 1 2 5
seed=1835940555     x= 2 4 5
seed=910897519      x= 2 3 4
```

Example 2: Using CALL RANCOMB with a Macro

The following is an example of the CALL RANCOMB routine that is used with macros.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=3;
   %let seed = 12345;
   %do j=1 %to 10;
      %syscall rancomb(seed, k, x1, x2, x3, x4, x5);
      %put j=&j   &x1 &x2 &x3;
   %end;
   %mend;

%test;
```

SAS writes the following output to the log:

```
j=1   -0.1234 hippopotamus zebra
j=2   hippopotamus -0.1234 10000000000
j=3   hippopotamus ant zebra
j=4   -0.1234 zebra ant
j=5   -0.1234 ant hippopotamus
j=6   10000000000 hippopotamus ant
j=7   10000000000 hippopotamus ant
j=8   ant 10000000000 -0.1234
j=9   zebra -0.1234 10000000000
j=10   zebra hippopotamus 10000000000
```

See Also

Functions:

- “RAND Function” on page 806

CALL Routines:

- “CALL RANPERK Routine” on page 224
- “CALL ALLPERM Routine” on page 156
- “CALL RANPERM Routine” on page 226

CALL RANEXP Routine

Returns a random variate from an exponential distribution.

Category: Random Number

Syntax

CALL RANEXP(*seed*,*x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANEXP is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See Seed Values on page 11 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 15 for more information about seed values

x

is a numeric variable. A new value for the random variate *x* is returned each time CALL RANEXP is executed.

Details

The CALL RANEXP routine updates *seed* and returns a variate *x* that is generated from an exponential distribution that has a parameter of 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

The CALL RANEXP routine uses an inverse transform method applied to a RANUNI uniform variate.

For a discussion and example of an effective use of the random number CALL routines, see Starting, Stopping, and Restarting a Stream on page 26.

Comparisons

The CALL RANEXP routine gives greater control of the seed and random number streams than does the RANEXP function.

Example

This example uses the CALL RANEXP routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call ranexp(seed, x);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call ranexp(seed, x);
      output;
   end;
run;

data all;
```

```
   set u1 u2;
   z = ranexp(104);
run;

proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.5 *Output from the CALL RANEXP Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	1.44347	1.44347
2	0.11740	0.11740
3	0.54175	0.54175
4	0.02280	0.02280
5	0.16645	0.16645

See Also

Functions:

- "RAND Function" on page 806
- "RANEXP Function" on page 817

CALL RANGAM Routine

Returns a random variate from a gamma distribution.

Category: Random Number

Syntax

CALL RANGAM(*seed*,*a*,*x*);

Required Arguments

seed
is the seed value. A new value for *seed* is returned each time CALL RANGAM is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See Seed Values on page 11 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 15 for more information about seed values

a

is a numeric shape parameter.

Range $a > 0$

x

is a numeric variable. A new value for the random variate *x* is returned each time CALL RANGAM is executed.

Details

The CALL RANGAM routine updates *seed* and returns a variate *x* that is generated from a gamma distribution with parameter *a*.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

For $a>1$, an acceptance-rejection method by Cheng is used (Cheng, 1977). For $a \leq 1$, an acceptance-rejection method by Fishman is used (Fishman, 1978). For more information see "References" on page 1001 .

For a discussion and example of an effective use of the random number CALL routines, see Starting. Stopping, and Restarting a Stream on page 26.

Comparisons

The CALL RANGAM routine gives greater control of the seed and random number streams than does the RANGAM function.

Example

This example uses the CALL RANGAM routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call rangam(seed, 1, x);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed = &seed
   do i = 1 to 5;
      call rangam(seed, 1, x);
      output;
   end;
run;

data all;
   set u1 u2;
   z = rangam(104, 1);
run;
```

```
proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.6 *Output from the CALL RANGAM Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	1.44347	1.44347
2	0.11740	0.11740
3	0.54175	0.54175
4	0.02280	0.02280
5	0.16645	0.16645

This is another example that uses the CALL RANGAM routine:

```
data case;
   retain Seed_1 Seed_2 Seed_3 45;
   a=2;
   do i=1 to 10;
      call rangam(Seed_1,a,X1);
      call rangam(Seed_2,a,X2);
      X3=rangam(Seed_3,a);
      if i=5 then
         do;
            Seed_2=18;
            Seed_3=18;
         end;
      output;
   end;
run;

proc print;
   id i;
   var Seed_1-Seed_3 X1-X3;
run;
```

Display 2.7 *Output from the CALL RANGAM Routine*

The SAS System

i	Seed_1	Seed_2	Seed_3	X1	X2	X3
1	1404437564	1404437564	45	1.30569	1.30569	1.30569
2	1326029789	1326029789	45	1.87514	1.87514	1.87514
3	1988843719	1988843719	45	1.71597	1.71597	1.71597
4	50049159	50049159	45	1.59304	1.59304	1.59304
5	802575599	18	18	0.43342	0.43342	0.43342
6	100573943	991271755	18	1.11812	1.32646	1.11812
7	1986749826	1437043694	18	0.68415	0.88806	0.68415
8	52428589	959908645	18	1.62296	2.46091	1.62296
9	1216356463	1225034217	18	2.26455	4.06596	2.26455
10	805366679	425626811	18	2.16723	6.94703	2.16723

Changing Seed_2 for the CALL RANGAM statement, when I=5, forces the stream of the variates for X2 to deviate from the stream of the variates for X1. Changing Seed_3 on the RANGAM function, however, has no effect.

See Also

Functions:

- “RAND Function” on page 806
- “RANGAM Function” on page 818

CALL RANNOR Routine

Returns a random variate from a normal distribution.

Category: Random Number

Syntax

CALL RANNOR(*seed*,*x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANNOR is executed.

Range $seed < 2^{31} - 1$

Note If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See Seed Values on page 11 and Comparison of Seed Values in Random-Number Functions and CALL Routines on page 15 for more information about seed values

x
is a numeric variable. A new value for the random variate *x* is returned each time CALL RANNOR is executed.

Details

The CALL RANNOR routine updates *seed* and returns a variate *x* that is generated from a normal distribution, with mean 0 and variance 1.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

The CALL RANNOR routine uses the Box-Muller transformation of RANUNI uniform variates.

For a discussion and example of an effective use of the random number CALL routines, see Starting, Stopping, and Restarting a Stream on page 26.

Comparisons

The CALL RANNOR routine gives greater control of the seed and random number streams than does the RANNOR function.

Example

This example uses the CALL RANNOR routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call rannor(seed, X);
      output;
   end;
   call symputx('seed', seed);
run;

data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call rannor(seed, X);
      output;
   end;
run;

data all;
   set u1 u2;
   z = rannor(104);
run;

proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.8 *Output from the CALL RANNOR Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	1.30390	1.30390
2	1.03049	1.03049
3	0.19491	0.19491
4	-0.34987	-0.34987
5	1.64273	1.64273
6	-1.75842	-1.75842
7	0.75080	0.75080
8	0.94375	0.94375
9	0.02436	0.02436
10	-0.97256	-0.97256

See Also

Functions:

- "RAND Function" on page 806
- "RANNOR Function" on page 821

CALL RANPERK Routine

Permutes the values of the arguments, and returns a random permutation of *k* out of *n* values.

Category: Combinatorial

Syntax

CALL RANPERK(*seed*, *k*, *variable-1*<, *variable–2, ...*>);

Required Arguments

seed
is a numeric variable that contains the random number seed. For more information about seeds, see "Seed Values" on page 11.

k
is the number of values that you want to have in the random permutation.

variable
specifies all numeric variables, or all character variables that have the same length. *K* values of these variables are randomly permuted.

Details

Using CALL RANPERK with Macros

You can call the RANPERK routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL RANPERK routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than -100.

If there are no errors, then &SYSERR and &SYSINFO are set to zero.

Examples

Example 1: Using CALL RANPERK in a DATA Step

The following example shows how to generate random permutations of given values by using the CALL RANPERK routine.

```
data _null_;
   array x x1-x5 (1 2 3 4 5);
   seed = 1234567890123;
   do n=1 to 10;
      call ranperk(seed, 3, of x1-x5);
      put seed= @20 ' x= ' x1-x3;
   end;
run;
```

Log 2.3 *Log Output from Using the CALL RANPERK Routine in a DATA Step*

```
seed=1332351321     x= 5 4 2
seed=829042065      x= 4 1 3
seed=767738639      x= 5 1 2
seed=1280236105     x= 3 2 5
seed=670350431      x= 4 3 5
seed=1956939964     x= 3 1 2
seed=353939815      x= 4 2 1
seed=1996660805     x= 3 4 5
seed=1835940555     x= 5 1 4
seed=910897519      x= 5 1 2
```

Example 2: Using CALL RANPERK with a Macro

The following is an example of the CALL RANPERK routine that is used with macros.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let k=3;
   %let seed = 12345;
   %do j=1 %to 10;
      %syscall ranperk(seed, k, x1, x2, x3, x4, x5);
```

```
      %put j=&j    &x1 &x2 &x3;
    %end;
    %mend;

%test;
```

Log 2.4 *Log Output from Using the CALL RANPERK Routine with a Macro*

```
j=1   -0.1234 hippopotamus zebra
j=2   hippopotamus -0.1234 10000000000
j=3   hippopotamus ant zebra
j=4   -0.1234 zebra ant
j=5   -0.1234 ant hippopotamus
j=6   10000000000 hippopotamus ant
j=7   10000000000 hippopotamus ant
j=8   ant 10000000000 -0.1234
j=9   zebra -0.1234 10000000000
j=10   zebra hippopotamus 10000000000
```

See Also

Functions:

- “RAND Function” on page 806

Call Routines:

- “CALL ALLPERM Routine” on page 156
- “CALL RANPERM Routine” on page 226

CALL RANPERM Routine

Randomly permutes the values of the arguments.

Category: Combinatorial

Syntax

CALL RANPERM(*seed*, *variable-1*<, *variable–2, ...*>);

Required Arguments

seed
is a numeric variable that contains the random number seed. For more information about seeds, see “Seed Values” on page 11.

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

Details

Using CALL RANPERM with Macros

You can call the RANPERM routine when you use the %SYSCALL macro. In this case, the *variable* arguments are not required to be the same type or length. If %SYSCALL identifies an argument as numeric, then %SYSCALL reformats the returned value.

If an error occurs during the execution of the CALL RANPERM routine, then both of the following values are set:

- &SYSERR is assigned a value that is greater than 4.
- &SYSINFO is assigned a value that is less than -100.

If there are no errors, then &SYSERR and &SYSINFO are set to zero.

Examples

Example 1: Using CALL RANPERM in a DATA Step

The following example generates random permutations of given values by using the CALL RANPERM routine.

```
data _null_;
         array x x1-x4 (1 2 3 4);
         seed = 1234567890123;
         do n=1 to 10;
            call ranperm(seed, of x1-x4);
         put seed= @20 ' x= ' x1-x4;
         end;
run;
```

Log 2.5 *Output from Using the CALL RANPERM Routine in a DATA Step*

```
seed=1332351321     x= 1 3 2 4
seed=829042065      x= 3 4 2 1
seed=767738639      x= 4 2 3 1
seed=1280236105     x= 1 2 4 3
seed=670350431      x= 2 1 4 3
seed=1956939964     x= 2 4 3 1
seed=353939815      x= 4 1 2 3
seed=1996660805     x= 4 3 1 2
seed=1835940555     x= 4 3 2 1
seed=910897519      x= 3 2 1 4
```

Example 2: Using CALL RANPERM with a Macro

The following is an example of the CALL RANPERM routine that is used with the %SYSCALL macro.

```
%macro test;
   %let x1=ant;
   %let x2=-.1234;
   %let x3=1e10;
   %let x4=hippopotamus;
   %let x5=zebra;
   %let seed = 12345;
   %do j=1 %to 10;
      %syscall ranperm(seed, x1, x2, x3, x4, x5);
```

```
      %put j=&j    &x1 &x2 &x3;
   %end;
   %mend;

%test;
```

Log 2.6 *Output from Using the CALL RANPERM Routine with a Macro*

```
j=1   zebra ant hippopotamus
j=2   10000000000 ant -0.1234
j=3   -0.1234 10000000000 ant
j=4   hippopotamus ant zebra
j=5   -0.1234 zebra 10000000000
j=6   -0.1234 hippopotamus ant
j=7   zebra ant -0.1234
j=8   -0.1234 hippopotamus ant
j=9   ant -0.1234 hippopotamus
j=10   -0.1234 zebra 10000000000
```

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- "CALL ALLPERM Routine" on page 156
- "CALL RANPERK Routine" on page 224

CALL RANPOI Routine

Returns a random variate from a Poisson distribution.

Category: Random Number

Syntax

CALL RANPOI(*seed*,*m*,*x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANPOI is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See "Seed Values" on page 11 and "Comparison of Seed Values in Random-Number Functions and CALL Routines" on page 15 for more information about seed values

m
is a numeric mean parameter.

Range $m \geq 0$

x
is a numeric variable. A new value for the random variate *x* is returned each time CALL RANPOI is executed.

Details

The CALL RANPOI routine updates *seed* and returns a variate *x* that is generated from a Poisson distribution, with mean *m*.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

For $m < 85$, an inverse transform method applied to a RANUNI uniform variate is used (Fishman, 1976; see in "References" on page 1001.) For $m \geq 85$, the normal approximation of a Poisson random variable is used. To expedite execution, internal variables are calculated only on initial calls (that is, with each new *m*).

For a discussion and example of an effective use of the random number CALL routines, see "Starting, Stopping, and Restarting a Stream" on page 26.

Comparisons

The CALL RANPOI routine gives greater control of the seed and random number streams than does the RANPOI function.

Example

This example uses the CALL RANPOI routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call ranpoi(seed, 2000, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call ranpoi(seed, 2000, x);
      output;
   end;
run;
data all;
   set u1 u2;
   z = ranpoi(104, 2000);
run;
proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.9 *Output from the CALL RANPOI Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	2058	2058
2	2046	2046
3	2009	2009
4	1984	1984
5	2073	2073
6	1921	1921
7	2034	2034
8	2042	2042
9	2001	2001
10	1957	1957

See Also

Functions:

- “RAND Function” on page 806
- “RANPOI Function” on page 822

CALL RANTBL Routine

Returns a random variate from a tabled probability distribution.

Category: Random Number

Syntax

CALL RANTBL(*seed*,p_1,...p_i,...p_n,*x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANTBL is executed.

Range *seed* < 2^{31} - 1

Note If *seed* ≤ 0, the time of day is used to initialize the seed stream.

See "Seed Values" on page 11 and "Comparison of Seed Values in Random-Number Functions and CALL Routines" on page 15

***p*$_i$**
is a numeric SAS value.

Range $0 \le p_i \le 1$ for $0 < i \le n$

x
is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANTBL is executed.

Details

The CALL RANTBL routine updates *seed* and returns a variate *x* generated from the probability mass function defined by p_1 through p_n.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

An inverse transform method applied to a RANUNI uniform variate is used. The CALL RANTBL routine returns these data:

$$1 \quad with \ probability \ p_1$$

$$2 \quad with \ probability \ p_2$$

.

.

.

$$n \quad with \ probability \ p_n$$

$$n+1 \quad with \ probability \ 1 - \sum_{i=1}^{n} p_i \ if \sum_{i=1}^{n} p_i \le 1$$

If, for some index $j<n$,

$$\sum_{i=1}^{j} p_i \ge 1$$

RANTBL returns only the indices 1 through *j*, with the probability of occurrence of the index *j* equal to

$$1 - \sum_{i=1}^{j-1} p_i$$

For a discussion and example of an effective use of the random number CALL routines, see "Starting, Stopping, and Restarting a Stream" on page 26.

Comparisons

The CALL RANTBL routine gives greater control of the seed and random number streams than does the RANTBL function.

Example

This example uses the CALL RANTBL routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call rantbl(seed, .02, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call rantbl(seed, .02, x);
      output;
   end;
run;
data all;
   set u1 u2;
   z = rantbl(104, .02);
run;
proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.10 *Output from the CALL RANTBL Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	2	2
2	2	2
3	2	2
4	2	2
5	2	2
6	2	2
7	2	2
8	2	2
9	2	2
10	2	2

See Also

Functions:

- “RAND Function” on page 806
- “RANTBL Function” on page 823

CALL RANTRI Routine

Returns a random variate from a triangular distribution.

Category: Random Number

Syntax

CALL RANTRI(*seed*,*h*,*x*);

Required Arguments

seed

is the seed value. A new value for *seed* is returned each time CALL RANTRI is executed.

Range $seed < 2^{31} - 1$

Note If $seed \leq 0$, the time of day is used to initialize the seed stream.

See “Seed Values” on page 11 and “Comparison of Seed Values in Random-Number Functions and CALL Routines” on page 15 for more information about seed values

h

is a numeric SAS value.

Range $0<h<1$

x

is a numeric SAS variable. A new value for the random variate *x* is returned each time CALL RANTRI is executed.

Details

The CALL RANTRI routine updates *seed* and returns a variate *x* generated from a triangular distribution on the interval (0,1) with parameter *h*, which is the modal value of the distribution.

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

The CALL RANTRI routine uses an inverse transform method applied to a RANUNI uniform variate.

For a discussion and example of an effective use of the random number CALL routines, see “Starting, Stopping, and Restarting a Stream” on page 26.

Comparisons

The CALL RANTRI routine gives greater control of the seed and random number streams than does the RANTRI function.

Example

This example uses the CALL RANTRI routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call rantri(seed, .5, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call rantri(seed, .5, x);
      output;
   end;
run;
data all;
   set u1 u2;
   z = rantri(104, .5);
run;
proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.11 *Output from the CALL RANTRI Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	0.34359	0.34359
2	0.76466	0.76466
3	0.54269	0.54269
4	0.89384	0.89384
5	0.72311	0.72311
6	0.68763	0.68763
7	0.48468	0.48468
8	0.38467	0.38467
9	0.29881	0.29881
10	0.80369	0.80369

See Also

Functions:

- "RAND Function" on page 806
- "RANTRI Function" on page 825

CALL RANUNI Routine

Returns a random variate from a uniform distribution.

Category: Random Number

Syntax

CALL RANUNI(*seed*,*x*);

Required Arguments

seed
is the seed value. A new value for *seed* is returned each time CALL RANUNI is executed.

Range $seed < 2^{31} - 1$

Tip If $seed \leq 0$, the time of day is used to initialize the seed stream.

See "Seed Values" on page 11 and "Comparison of Seed Values in Random-Number Functions and CALL Routines" on page 15 for more information about seed values

x
is a numeric variable. A new value for the random variate *x* is returned each time CALL RANUNI is executed.

Details

The CALL RANUNI routine updates *seed* and returns a variate *x* that is generated from the uniform distribution on the interval (0,1), using a prime modulus multiplicative generator with modulus 2^{31}–1 and multiplier 397204094 (Fishman and Moore 1982) (See "References" on page 1001.).

By adjusting the seeds, you can force streams of variates to agree or disagree for some or all of the observations in the same, or in subsequent, DATA steps.

For a discussion and example of an effective use of the random number CALL routines, see "Starting, Stopping, and Restarting a Stream" on page 26.

Comparisons

The CALL RANUNI routine gives greater control of the seed and random number streams than does the RANUNI function.

Example: Using the CALL RANUNI Routine

This example uses the CALL RANUNI routine:

```
data u1(keep=x);
   seed = 104;
   do i = 1 to 5;
      call ranuni(seed, x);
      output;
   end;
   call symputx('seed', seed);
run;
data u2(keep=x);
   seed = &seed;
   do i = 1 to 5;
      call ranuni(seed, x);
      output;
   end;
run;
data all;
   set u1 u2;
   z = ranuni(104);
run;
proc print label;
   label x = 'Separate Streams' z = 'Single Stream';
run;
```

Display 2.12 *Output from the CALL RANUNI Routine*

The SAS System

Obs	Separate Streams	Single Stream
1	0.23611	0.23611
2	0.88923	0.88923
3	0.58173	0.58173
4	0.97746	0.97746
5	0.84667	0.84667
6	0.80484	0.80484
7	0.46983	0.46983
8	0.29594	0.29594
9	0.17858	0.17858
10	0.92292	0.92292

See Also

Functions:

- "RAND Function" on page 806
- "RANUNI Function" on page 826

CALL SCAN Routine

Returns the position and length of the *n*th word from a character string.

Category: Character

Interaction: When invoked by the %SYSCALL macro statement, CALL SCAN removes the quotation marks from its arguments. For more information, see "Using CALL Routines and the %SYSCALL Macro Statement" on page 9 .

Syntax

CALL SCAN(<*string*> , *count*, *position*, *length* <, <*charlist*> <, <*modifier(s)*> > >);

Required Arguments

count
is a nonzero numeric constant, variable, or expression that has an integer value that specifies the number of the word in the character string that you want the CALL SCAN routine to select. For example, a value of 1 indicates the first word, a value of 2 indicates the second word, and so on. The following rules apply:

- If *count* is positive, then CALL SCAN counts words from left to right in the character string.
- If *count* is negative, then CALL SCAN counts words from right to left in the character string.

position
specifies a numeric variable in which the position of the word is returned. If *count* exceeds the number of words in the string, then the value that is returned in *position* is zero. If *count* is zero or missing, then the value that is returned in *position* is missing.

length
specifies a numeric variable in which the length of the word is returned. If *count* exceeds the number of words in the string, then the value that is returned in *length* is zero. If *count* is zero or missing, then the value that is returned in *length* is missing.

Optional Arguments

string
specifies a character constant, variable, or expression.

charlist
specifies an optional character constant, variable, or expression that initializes a list of characters. This list determines which characters are used as the delimiters that separate words. The following rules apply:

- By default, all characters in *charlist* are used as delimiters.
- If you specify the K modifier in the *modifier* argument, then all characters that are not in *charlist* are used as delimiters.

Tip You can add more characters to *charlist* by using other modifiers.

modifier
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the CALL SCAN routine. Blanks are ignored. You can use the following characters as modifiers:

a or A adds alphabetic characters to the list of characters.

b or B scans backwards, from right to left instead of from left to right, regardless of the sign of the *count* argument.

c or C adds control characters to the list of characters.

d or D adds digits to the list of characters.

f or F adds an underscore and English letters (that is, valid first characters in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

g or G adds graphic characters to the list of characters. Graphic characters are those that, when printed, produce an image on paper.

h or H adds a horizontal tab to the list of characters.

i or I ignores the case of the characters.

k or K causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, then characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.

l or L adds lower case letters to the list of characters.

m or M specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.

n or N adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O processes the *charlist* and *modifier* arguments only once, rather than every time the CALL SCAN routine is called. Using the O modifier in the DATA step can make CALL SCAN run faster when you call it in a loop where the *charlist* and *modifier* arguments do not change. The O modifier applies separately to each instance of the CALL SCAN routine in your SAS code, and does not cause all instances of the CALL SCAN routine to use the same delimiters and modifiers.

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

q or Q ignores delimiters that are inside of 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.

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 just one character argument instead of both character arguments, then use the TRIM function instead of the CALL SCAN routine with the T modifier.
u or U	adds upper case letters to the list of characters.
w or W	adds printable (writable) characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

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

Details

Definition of ”Delimiter” and “Word”

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

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

In the CALL SCAN routine, “word” refers to a substring that has all of the following characteristics:

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

A word can have a length of 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 CALL SCAN routine ignores words that have a length of zero unless you specify the M modifier.

Using Default Delimiters in ASCII and EBCDIC Environments

If you use the CALL SCAN routine with only four arguments, then 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 CALL SCAN routine 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 will be used are those 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 *charlist* argument. Modifiers do not add to the list of default modifiers.

Using the CALL SCAN Routine with the M Modifier

If you specify the M modifier, then the number of words in a string is defined as one plus the number of delimiters in the string. However, if you specify the Q modifier, delimiters that are inside quotation marks are ignored.

If you specify the M modifier, the CALL SCAN routine returns a positive position and a length of zero if one of the following 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.

In you specify a count that is greater in absolute value than the number of words in the string, then the CALL SCAN routine returns a position and length of zero.

Using the CALL SCAN Routine without the M Modifier

If you do not specify the M modifier, then 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, then the CALL SCAN routine does the following:

- ignores delimiters at the beginning or end of the string
- treats two or more consecutive delimiters as if they were a single delimiter

If the string contains no characters other than delimiters, or if you specify a count that is greater in absolute value than the number of words in the string, then the CALL SCAN routine returns a position and length of zero.

Finding the Word as a Character String

To find the designated word as a character string after calling the CALL SCAN routine, use the SUBSTRN function with the *string*, *position*, and *length* arguments:

```
substrn(string, position, length);
```

Because CALL SCAN can return a length of zero, using the SUBSTR function can cause an error.

Using Null Arguments

The CALL SCAN routine allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Examples

Example 1: Scanning for a Word in a String

The following example shows how you can use the CALL SCAN routine to find the position and length of a word in a string.

```
data artists;
   input string $60.;
   drop string;
   do i=1 to 99;
      call scan(string, i, position, length);
      if not position then leave;
      Name=substrn(string, position, length);
```

```
      output;
   end;
   datalines;
Picasso Toulouse-Lautrec Turner "Van Gogh" Velazquez
;
proc print data=artists;
run;
```

Display 2.13 *SAS Output: Scanning for a Word in a String*

The SAS System

Obs	i	position	length	Name
1	1	1	7	Picasso
2	2	9	8	Toulouse
3	3	18	7	Lautrec
4	4	26	6	Turner
5	5	33	4	"Van
6	6	38	5	Gogh"
7	7	44	9	Velazquez

Example 2: Finding the First and Last Words in a String

The following example scans a string for the first and last words. Note the following:

- A negative count instructs the CALL SCAN routine to scan from right to left.
- Leading and trailing delimiters are ignored because the M modifier is not used.
- In the last observation, all characters in the string are delimiters, so no words are found.

```
data firstlast;
   input String $60.;
   call scan(string, 1, First_Pos, First_Length);
   First_Word = substrn(string, First_Pos, First_Length);
   call scan(string, -1, Last_Pos, Last_Length);
   Last_Word = substrn(string, Last_Pos, Last_Length);
   datalines4;
Jack and Jill
& Bob & Carol & Ted & Alice &
Leonardo
! $ % & ( ) * + , - . / ;
;;;;
proc print data=firstlast;
   var First: Last:;
run;
```

Display 2.14 *Results of Finding the First and Last Words in a String*

The SAS System

Obs	First_Pos	First_Length	First_Word	Last_Pos	Last_Length	Last_Word
1	1	4	Jack	10	4	Jill
2	3	3	Bob	23	5	Alice
3	1	8	Leonardo	1	8	Leonardo
4	0	0		0	0	

Example 3: Finding All Words in a String without Using the M Modifier

The following example scans a string from left to right until no more words are found. Because the M modifier is not used, the CALL SCAN routine does not return any words that have a length of zero. Because blanks are included among the default delimiters, the CALL SCAN routine returns a position or length of zero only when the count exceeds the number of words in the string. The loop can be stopped when the returned position is less than or equal to zero. It is safer to use an inequality comparison to end the loop, rather than to use a strict equality comparison with zero, in case an error causes the position to be missing. (In SAS, a missing value is considered to have a lesser value than any nonmissing value.)

```
data all;
   length word $20;
   drop string;
   string = ' The quick brown fox jumps over the lazy dog.   ';
   do until(position <= 0);
      count+1;
      call scan(string, count, position, length);
      word = substrn(string, position, length);
      output;
   end;
run;
proc print data=all noobs;
   var count position length word;
run;
```

Display 2.15 *Results of Finding All Words in a String without Using the M Modifier*

The SAS System

count	position	length	word
1	2	3	The
2	6	5	quick
3	12	5	brown
4	18	3	fox
5	22	5	jumps
6	28	4	over
7	33	3	the
8	37	4	lazy
9	42	3	dog
10	0	0	

Example 4: Finding All Words in a String by Using the M and O Modifiers

The following example shows the results of using the M modifier with a comma as a delimiter. With the M modifier, leading, trailing, and multiple consecutive delimiters cause the CALL SCAN routine to return words that have a length of zero.

The O modifier is used for efficiency because the delimiters and modifiers are the same in every call to the CALL SCAN routine.

```
data comma;
   length word $30;
   string = ',leading,  trailing,and multiple,,delimiters,,';
   do until(position <= 0);
      count + 1;
      call scan(string, count, position, length, ',', 'mo');
      word = substrn(string, position, length);
      output;
   end;
run;
proc print data=comma noobs;
   var count position length word;
run;
```

Display 2.16 *Results of Finding All Words in a String by Using the M and O Modifiers*

The SAS System

count	position	length	word
1	1	0	
2	2	7	leading
3	10	10	trailing
4	21	12	and multiple
5	34	0	
6	35	10	delimiters
7	46	0	
8	47	0	
9	0	0	

Example 5: Using Comma-Separated Values, Substrings in Quotation Marks, and the O Modifier

The following example uses the CALL SCAN routine with the O modifier and a comma as a delimiter.

The O modifier is used for efficiency because in each call of the CALL SCAN routine, the delimiters and modifiers do not change.

```
data test;
   length word word_r $30;
   string = 'He said, "She said, ""No!""", not "Yes!"';
   do until(position <= 0);
      count + 1;
      call scan(string, count, position, length, ',', 'oq');
      word = substrn(string, position, length);
      output;
   end;
run;
proc print data=test noobs;
   var count position length word;
run;
```

Display 2.17 *Results of Comma-Separated Values and Substrings in Quotation Marks*

The SAS System

count	position	length	word
1	1	7	He said
2	9	20	"She said, ""No!"""
3	30	11	not "Yes!"
4	0	0	

Example 6: Finding Substrings of Digits by Using the D and K Modifiers

The following example finds substrings of digits. The *charlist* argument is null, and consequently the list of characters is initially empty. The D modifier adds digits to the list of characters. The K modifier treats all characters that are not in the list as delimiters. Therefore, all characters except digits are delimiters.

```
data digits;
   length digits $20;
   string = 'Call (800) 555-1234 now!';
   do until(position <= 0);
      count+1;
      call scan(string, count, position, length, , 'dko');
      digits = substrn(string, position, length);
      output;
   end;
run;
proc print data=digits noobs;
   var count position length digits;
run;
```

Display 2.18 *Results of Finding Substrings of Digits by Using the D and K Modifiers*

The SAS System

count	position	length	digits
1	7	3	800
2	12	3	555
3	16	4	1234
4	0	0	

See Also

Functions:

- "SCAN Function" on page 848
- "FINDW Function" on page 467
- "COUNTW Function" on page 343

CALL SET Routine

Links SAS data set variables to DATA step or macro variables that have the same name and data type.

Category: Variable Control

Syntax

CALL SET(*data-set-id*);

Required Argument

data-set-id
is the identifier that is assigned by the OPEN function when the data set is opened.

Details

Using SET can significantly reduce the coding that is required for accessing variable values for modification or verification when you use functions to read or to manipulate a SAS file. After a CALL SET, whenever a read is performed from the SAS data set, the values of the corresponding macro or DATA step variables are set to the values of the matching SAS data set variables. If the variable lengths do not match, the values are truncated or padded according to need. If you do not use SET, then you must use the GETVARC and GETVARN functions to move values explicitly between data set variables and macro or DATA step variables.

As a general rule, use CALL SET immediately following OPEN if you want to link the data set and the macro and DATA step variables.

Example: Using the CALL SET Routine

This example uses the CALL SET routine:

- The following statements automatically set the values of the macro variables PRICE and STYLE when an observation is fetched:

```
%macro setvar;
   %let dsid=%sysfunc(open(sasuser.houses,i));
      /* No leading ampersand with %SYSCALL */
   %syscall set(dsid);
   %let rc=%sysfunc(fetchobs(&dsid,10));
   %let rc=%sysfunc(close(&dsid));
%mend setvar;
%global price style;
%setvar
%put _global_;
```

- The %PUT statement writes these lines to the SAS log:

```
GLOBAL PRICE 127150
GLOBAL STYLE CONDO
```

- The following statements obtain the values for the first 10 observations in SASUSER.HOUSES and store them in MYDATA:

```
data mydata;
      /* create variables for assignment */
      /*by CALL SET */
   length style $8 sqfeet bedrooms baths 8
      street $16 price 8;
   drop rc dsid;
   dsid=open("sasuser.houses","i");
   call set (dsid);
   do i=1 to 10;
      rc=fetchobs(dsid,i);
      output;
   end;
run;
```

See Also

Functions:

- “FETCH Function” on page 405
- “FETCHOBS Function” on page 406
- “GETVARC Function” on page 521
- “GETVARN Function” on page 522

CALL SLEEP Routine

For a specified period of time, suspends the execution of a program that invokes this CALL routine.

Category: Special

See: “CALL SLEEP Routine: UNIX” in *SAS Companion for UNIX Environments*
“CALL SLEEP Routine: z/OS” in *SAS Companion for z/OS*

Syntax

CALL SLEEP(*n*<, *unit*>);

Required Argument

n
is a numeric constant that specifies the number of units of time for which you want to suspend execution of a program.

Range $n \geq 0$

Optional Argument

unit

specifies the unit of time in seconds, which is applied to *n*. For example, 1 corresponds to 1 second, .001 corresponds to 1 millisecond, and 5 corresponds to 5 seconds.

Default 1 in a Windows PC environment, .001 in other environments

Details

The CALL SLEEP routine suspends the execution of a program that invokes this call routine for a period of time that you specify. The program can be a DATA step, macro, IML, SCL, or anything that can invoke a call routine. The maximum sleep period for the CALL SLEEP routine is 46 days.

Examples

Example 1: Suspending Execution for a Specified Period of Time

The following example tells SAS to suspend the execution of the DATA step PAYROLL for 1 minute and 10 seconds:

```
data payroll;
   call sleep(7000,.01);
   ...more SAS statements...
run;
```

Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following example tells SAS to suspend the execution of the DATA step BUDGET until March 1, 2013, at 3:00 AM. SAS calculates the length of the suspension based on the target date and the date and time that the DATA step begins to execute.

```
data budget;
   sleeptime='01mar2013:03:00'dt-datetime();
   call sleep(sleeptime,1);
   ...more SAS statements...;
run;
```

See Also

Functions:

- “SLEEP Function” on page 863

CALL SOFTMAX Routine

Returns the softmax value.

Category: Mathematical

Syntax

CALL SOFTMAX(*argument*<,*argument*,…>);

Required Argument

argument
is numeric.

Restriction The CALL SOFTMAX routine only accepts variables as valid arguments. Do not use a constant or a SAS expression because the CALL routine is unable to update these arguments.

Details

The CALL SOFTMAX routine replaces each argument with the softmax value of that argument. For example x_j is replaced by

$$\frac{\varepsilon^{x_j}}{\sum_{i=1}^{i=n} \varepsilon^{x_i}}$$

If any argument contains a missing value, then CALL SOFTMAX returns missing values for all the arguments. Upon a successful return, the sum of all the values is equal to 1.

Example

The following SAS statements produce these results.

SAS Statement	Result

`x=0.5;`
`y=-0.5;`
`z=1;`
`call softmax(x,y,z);`
`put x= y= z=;`
	`x=0.3314989604 y=0.1219516523 z=0.5465493873`

CALL SORTC Routine

Sorts the values of character arguments.

Category: Sort

Interaction: When invoked by the %SYSCALL macro statement, CALL SORTC removes the quotation marks from its arguments. For more information, see "Using CALL Routines and the %SYSCALL Macro Statement" on page 9.

Syntax

CALL SORTC(*variable-1*<, …, *variable-n*>);

Required Argument

variable
specifies a character variable.

Details

The values of *variable* are sorted in ascending order by the CALL SORTC routine.

Comparisons

The CALL SORTC routine is used with character variables, and the CALL SORTN routine is used with numeric variables.

Example

The following example sorts the character variables in the array in ascending order.

```
data _null_;
   array x(8) $10
      ('tweedledum' 'tweedledee' 'baboon' 'baby'
       'humpty' 'dumpty' 'banana' 'babylon');
   call sortc(of x(*));
   put +3 x(*);
run;
```

SAS writes the following output to the log:

```
   baboon baby babylon banana dumpty humpty tweedledee tweedledum
```

See Also

CALL Routines:

- "CALL SORTN Routine" on page 250

CALL SORTN Routine

Sorts the values of numeric arguments.

Category: Sort

Syntax

CALL SORTN(*variable-1*<, …, *variable-n*>);

Required Argument

variable
specifies a numeric variable.

Details

The values of *variable* are sorted in ascending order by the CALL SORTN routine.

Comparisons

The CALL SORTN routine is used with numeric variables, and the CALL SORTC routine is used with character variables.

Example

The following example sorts the numeric variables in the array in ascending order.

```
data _null_;
   array x(10) (0, ., .a, 1e-12, -1e-8, .z, -37, 123456789, 1e20, 42);
   call sortn(of x(*));
   put +3 x(*);
run;
```

SAS writes the following output to the log:

```
   . A Z -37 -1E-8 0 1E-12 42 123456789 1E20
```

See Also

CALL Routines:

- "CALL SORTC Routine" on page 249

CALL STDIZE Routine

Standardizes the values of one or more variables.

Category: Mathematical

Interaction: When invoked by the %SYSCALL macro statement, CALL STDIZE removes the quotation marks from its arguments. For more information, see "Using CALL Routines and the %SYSCALL Macro Statement" on page 9.

Syntax

CALL STDIZE(<*option-1*,*option-2, …*,> *variable-1*<, *variable-2, …*>);

Required Argument

variable
is numeric. These values will be standardized according to the method that you use.

Optional Arguments

option
specifies a character expression whose values can be uppercase, lowercase, or mixed case letters. Leading and trailing blanks are ignored. *option* includes the following three categories:

- standardization-options
- VARDEF-options
- miscellaneous-options

Restriction Use a separate argument for each option because you cannot specify more than one option in a single argument.

Tip Character expressions can end with an equal sign that is followed by another argument that is a numeric constant, variable, or expression.

See PROC STDIZE in *SAS/STAT 9.3 User's Guide* for information about formulas and other details. The options that are used in CALL STDIZE are the same as those used in PROC STDIZE.

standardization-options
specify how to compute the location and scale measures that are used to standardize the variables. The following standardization options are available:

ABW= must be followed by an argument that is a numeric expression specifying the tuning constant.

AGK= must be followed by an argument that is a numeric expression that specifies the proportion of pairs to be included in the estimation of the within-cluster variances.

AHUBER= must be followed by an argument that is a numeric expression specifying the tuning constant.

AWAVE= must be followed by an argument that is a numeric expression specifying the tuning constant.

EUCLEN specifies the Euclidean length.

IQR specifies the interquartile range.

L= must be followed by an argument that is a numeric expression with a value greater than or equal to 1 specifying the power to which differences are to be raised in computing an L(p) or Minkowski metric.

MAD specifies the median absolute deviation from the median.

MAXABS specifies the maximum absolute values.

MEAN specifies the arithmetic mean (average).

MEDIAN specifies the middle number in a set of data that is ordered according to rank.

MIDRANGE specifies the midpoint of the range.

RANGE specifies a range of values.

SPACING= must be followed by an argument that is a numeric expression that specifies the proportion of data to be contained in the spacing.

STD specifies the standard deviation.

SUM specifies the result that you obtain when you add numbers.

USTD specifies the standard deviation about the origin, based on the uncorrected sum of squares.

VARDEF-options
specify the divisor to be used in the calculation of variances. VARDEF options can have the following values:

DF specifies degrees of freedom.

N specifies the number of observations. The default is DF.

miscellaneous-options

Miscellaneous options can have the following values:

ADD= is followed by a numeric argument that specifies a number to add to each value after standardizing and multiplying by the value from the MULT= option. The default value is 0.

FUZZ= is followed by a numeric argument that specifies the relative fuzz factor.

MISSING= is followed by a numeric argument that specifies a value to be assigned to variables that have a missing value.

MULT= is followed by a numeric argument that specifies a number by which to multiply each value after standardizing. The default value is 1.

NORM normalizes the scale estimator to be consistent for the standard deviation of a normal distribution. This option affects only the methods AGK=, IQR, MAD, and SPACING=.

PSTAT writes the values of the location and scale measures in the log.

REPLACE replaces missing values with the value 0 in the standardized data (this value corresponds to the location measure before standardizing). To replace missing values by other values, see the MISSING= option.

SNORM normalizes the scale estimator to have an expectation of approximately 1 for a standard normal distribution. This option affects only the SPACING= method.

Details

The CALL STDIZE routine transforms one or more arguments that are numeric variables by subtracting a location measure and dividing by a scale measure. You can use a variety of location and scale measures. The default location option is MEAN, and the default scale option is STD.

In addition, you can multiply each standardized value by a constant, and you can add a constant. The final output value would be $result = add + mult * \left(\frac{(original - location)}{scale} \right)$.

These are the descriptions of the variables:

result
specifies the final value that is returned for each variable.

add
specifies the constant to add (ADD= option).

mult
specifies the constant to multiply by (MULT= option).

original
specifies the original input value.

location
specifies the location measure.

scale
specifies the scale measure.

You can replace missing values by any constant. If you do not specify the MISSING= or the REPLACE option, variables that have missing values are not altered. The initial estimation method for the ABW=, AHUBER=, and AWAVE= methods is MAD. Percentiles are computed using definition 5. For more information about percentile calculations, see SAS Elementary Statistics Procedures SAS Elementary Statistics Procedures in *Base SAS Procedures Guide*.

Comparisons

The CALL STDIZE routine is similar to the STDIZE procedure in the SAS/STAT product. However, the CALL STDIZE routine is primarily useful for standardizing the rows of a SAS data set, whereas the STDIZE procedure can standardize only the columns of a SAS data set. For more information, see PROC STDIZE in *SAS/STAT User's Guide*.

Example

The following SAS statements produce these results.

SAS Statement	Result

`retain x 1 y 2 z 3;`
`call stdize(x,y,z);`
`put x= y= z=;`
	`x=-1 y=0 z=1`

`retain w 10 x 11 y 12 z 13;`
`call stdize('iqr',w,x,y,z);`
`put w= x= y= z=;`
	`w=-0.75 x=-0.25 y=0.25 z=0.75`

`retain w . x 1 y 2 z 3;`
`call stdize('range',w,x,y,z);`
`put w= x= y= z=;`
	`w=. x=0 y=0.5 z=1`

`retain w . x 1 y 2 z 3;`
`call stdize('mult=',10,'missing=',`
`-1,'range',w,x,y,z);`
`put w= x= y= z=;`
	`w=-1 x=0 y=5 z=10`

CALL STREAMINIT Routine

Specifies a seed value to use for subsequent random number generation by the RAND function.

Category: Random Number

Syntax

CALL STREAMINIT(*seed*);

Required Argument

seed
is an integer seed value.

Range $seed < 2^{31} - 1$

Tip If you specify a nonpositive seed, then CALL STREAMINIT is ignored. Any subsequent random number generation seeds itself from the system clock.

Details

If you want to create reproducible streams of random numbers, then specify CALL STREAMINIT before any calls to the RAND random number function. If you call the RAND function before you specify a seed with the CALL STREAMINIT routine (or if you specify a nonpositive seed value in the CALL STREAMINIT routine), then the RAND function uses a call to the system clock to seed itself. Each DATA step honors one CALL STREAMINIT seed. The prevailing seed value is the one that is specified prior to the first RAND function call. For more information about seed values see "Seed Values" on page 11.

Example: Creating a Reproducible Stream of Random Numbers

The following example shows how to specify a seed value with CALL STREAMINIT to create a reproducible stream of random numbers with the RAND function.

```
data random;
   call streaminit(123);
   do i=1 to 10;
      x1=rand('cauchy');
      output;
   end;
proc print data=random;
   id i;
run;
```

Display 2.19 *Number String Seeded with CALL STREAMINIT*

The SAS System

i	x1
1	-0.17593
2	3.76106
3	1.23427
4	0.49095
5	-0.05094
6	0.72496
7	-0.51646
8	7.61304
9	0.89784
10	1.69348

See Also

Functions:

- "RAND Function" on page 806

CALL SYMPUT Routine

Assigns DATA step information to a macro variable.

Category: Macro

Syntax

CALL SYMPUT(*argument-1*,*argument-2*);

Required Arguments

argument-1
specifies a character expression that identifies the macro variable that is assigned a value. If the macro variable does not exist, the routine creates it.

argument-2
specifies a character constant, variable, or expression that contains the value that is assigned.

Details

The CALL SYMPUT routine either creates a macro variable whose value is information from the DATA step or assigns a DATA step value to an existing macro variable. CALL SYMPUT is fully documented in "SYMPUT Routine" in *SAS Macro Language: Reference*.

See Also

Functions:

- "SYMGET Function" on page 901

CALL SYMPUTX Routine

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

Category: Macro

Syntax

CALL SYMPUTX(*macro-variable*, *value* <,*symbol-table*>);

Required Arguments

macro-variable
can be one of the following:

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

a character constant, variable, or expression. Leading and trailing blanks are removed from the value of *name*, and the result is then used as the name of the macro variable.

value
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 and does not issue a note to the SAS log. Leading and trailing blanks are removed, and the resulting character string is assigned to the macro variable.

Optional Argument

symbol-table
specifies a character constant, variable, or expression. The value of *symbol-table* is not case sensitive. The first non-blank character in *symbol-table* specifies the symbol table in which to store the macro variable. The following values are valid as the first non-blank character in *symbol-table*:

G specifies that the macro variable is stored in the global symbol table, even if the local symbol table exists.

L specifies that the macro variable is stored in the most local symbol table that exists, which will be the global symbol table, if used outside a macro.

F specifies that if the macro variable exists in any symbol table, CALL SYMPUTX uses the version in the most local symbol table in which it exists. If the macro variable does not exist, CALL SYMPUTX stores the variable in the most local symbol table.

Note: If you omit *symbol-table* or if *symbol-table* is blank, CALL SYMPUTX stores the macro variable in the same symbol table as does the CALL SYMPUT routine.

Details

CALL SYMPUTX is similar to CALL SYMPUT except that

- CALL SYMPUTX does not write a note to the SAS log when the second argument is numeric. CALL SYMPUT, however, writes a note to the log stating that numeric values were converted to character values.
- CALL SYMPUTX uses a field width of up to 32 characters when it converts a numeric second argument to a character value. CALL SYMPUT uses a field width of up to 12 characters.
- CALL SYMPUTX left-justifies both arguments and trims trailing blanks. CALL SYMPUT does not left-justify the arguments, and trims trailing blanks from the first argument only. Leading blanks in the value of *name* cause an error.
- CALL SYMPUTX enables you to specify the symbol table in which to store the macro variable, whereas CALL SYMPUT does not.

Example: Using CALL SYMPUTX

The following example shows the results of using CALL SYMPUTX.

```
data _null_;
   call symputx('  items  ', '   leading and trailing blanks removed   ',
                'lplace');
   call symputx('  x   ', 123.456);
run;
%put items=!&items!;
%put x=!&x!;
```

The following lines are written to the SAS log:

```
----+----1----+----2----+----3----+----4----+----5
items=!leading and trailing blanks removed!
x=!123.456!
```

See Also

Functions:

- “SYMGET Function” on page 901

CALL Routines:

- “CALL SYMPUT Routine” on page 256

CALL SYSTEM Routine

Submits an operating environment command for execution.

Category: Special

Interaction: When invoked by the %SYSCALL macro statement, CALL SYSTEM removes quotation marks from its arguments. For more information, see "Using CALL Routines and the %SYSCALL Macro Statement" on page 9.

See: "CALL SYSTEM Routine: Windows" in *SAS Companion for Windows*

"CALL SYSTEM Routine: UNIX" in *SAS Companion for UNIX Environments*

"CALL SYSTEM Routine: z/OS" in *SAS Companion for z/OS*

Syntax

CALL SYSTEM(*command*);

Required Argument

command

specifies any of the following: a system command that is enclosed in quotation marks (character string), an expression whose value is a system command, or the name of a character variable whose value is a system command that is executed.

Operating Environment Information

See the SAS documentation for your operating environment for information about what you can specify.

Restriction The length of the command cannot be greater than 1024 characters, including trailing blanks.

Details

The behavior of the CALL SYSTEM routine is similar to that of the X command, the X statement, and the SYSTEM function. It is useful in certain situations because it can be conditionally executed, it accepts an expression as an argument, and it is executed at run time.

See Also

Functions:

- "SYSTEM Function" on page 910

CALL TANH Routine

Returns the hyperbolic tangent.

Category: Mathematical

Syntax

CALL TANH(*argument*<, *argument,...*>);

Required Argument

argument
is numeric.

Restriction The CALL TANH routine only accepts variables as valid arguments. Do not use a constant or a SAS expression, because the CALL routine is unable to update these arguments.

Details

The subroutine TANH replaces each argument by the tanh of that argument. For example x_j is replaced by

$$tanh(x_j) = \frac{\varepsilon^{x_j} - \varepsilon^{-x_j}}{\varepsilon^{x_j} + \varepsilon^{-x_j}}$$

If any argument contains a missing value, then CALL TANH returns missing values for all the arguments.

Example: Examples

The following SAS statements produce these results.

SAS Statement	Result

`x=0.5;`
`y=-0.5;`
`call tanh(x,y);`
`put x= y=;`
	`x=0.4621171573 y=-0.462117157`

See Also

Functions:

- "TANH Function" on page 911

CALL VNAME Routine

Assigns a variable name as the value of a specified variable.

Category: Variable Control

Syntax

CALL VNAME(*variable-1*,*variable-2*);

Required Arguments

variable-1
specifies any SAS variable.

variable-2
specifies any SAS character variable. Because SAS variable names can contain up to 32 characters, the length of *variable-2* should be at least 32.

Details

The CALL VNAME routine assigns the name of the *variable-1* variable as the value of the *variable-2* variable.

Example: Using the CALL VNAME Routine

This example uses the CALL VNAME routine with array references to return the names of all variables in the data set OLD:

```
data new(keep=name);
   set old;
      /* all character variables in old */
   array abc{*} _character_;
      /* all numeric variables in old */
   array def{*} _numeric_;
      /* name is not in either array */
   length name $32;
   do i=1 to dim(abc);
         /* get name of character variable */
      call vname(abc{i},name);
         /* write name to an observation */
      output;
   end;
   do j=1 to dim(def);
         /* get name of numeric variable */
      call vname(def{j},name);
         /* write name to an observation */
      output;
   end;
   stop;
run;
```

See Also

Functions:

- “VNAME Function” on page 969
- “VNAMEX Function” on page 970

CALL VNEXT Routine

Returns the name, type, and length of a variable that is used in a DATA step.

Category: Variable Information

Syntax

CALL VNEXT(*varname* <,*vartype* <, *varlength*> >);

Required Argument

varname
is a character variable that is updated by the CALL VNEXT routine. The following rules apply:

- If the input value of *varname* is blank, the value that is returned in *varname* is the name of the first variable in the DATA step's list of variables.
- If the CALL VNEXT routine is executed for the first time in the DATA step, the value that is returned in *varname* is the name of the first variable in the DATA step's list of variables.

If neither of the above conditions exists, the input value of *varname* is ignored. Each time the CALL VNEXT routine is executed, the value that is returned in *varname* is the name of the next variable in the list.

After the names of all the variables in the DATA step are returned, the value that is returned in *varname* is blank.

Optional Arguments

vartype
is a character variable whose input value is ignored. The value that is returned is "N" or "C." The following rules apply:

- If the value that is returned in *varname* is the name of a numeric variable, the value that is returned in *vartype* is "N."
- If the value that is returned in *varname* is the name of a character variable, the value that is returned in *vartype* is "C."
- If the value that is returned in *varname* is blank, the value that is returned in *vartype* is also blank.

varlength
is a numeric variable. The input value of *varlength* is ignored.

The value that is returned is the length of the variable whose name is returned in *varname*. If the value that is returned in *varname* is blank, the value that is returned in *varlength* is zero.

Details

The variable names that are returned by the CALL VNEXT routine include automatic variables such as _N_ and _ERROR_. If the DATA step contains a BY statement, the variable names that are returned by CALL VNEXT include the FIRST.*variable* and LAST.*variable* names. CALL VNEXT also returns the names of the variables that are used as arguments to CALL VNEXT.

Note: The order in which variable names are returned by CALL VNEXT can vary in different releases of SAS and in different operating environments.

Example: Using the CALL VNEXT Routine

The following example shows the results from using the CALL VNEXT routine.

```
data test;
   x=1;
   y='abc';
   z=.;
   length z 5;
run;
data attributes;
   set test;
   by x;
   input a b $ c;
   length name $32 type $3;
   name=' ';
   length=666;
   do i=1 to 99 until(name=' ');
      call vnext(name,type,length);
      put i= name @40 type= length=;
   end;
   this_is_a_long_variable_name=0;
   datalines;
1 q 3
;
```

Log 2.7 *Partial SAS Log Output for the CALL VNEXT Routine*

```
i=1 x                                type=N length=8
i=2 y                                type=C length=3
i=3 z                                type=N length=5
i=4 FIRST.x                          type=N length=8
i=5 LAST.x                           type=N length=8
i=6 a                                type=N length=8
i=7 b                                type=C length=8
i=8 c                                type=N length=8
i=9 name                             type=C length=32
i=10 type                            type=C length=3
i=11 length                          type=N length=8
i=12 i                               type=N length=8
i=13 this_is_a_long_variable_name    type=N length=8
i=14 _ERROR_                         type=N length=8
i=15 _N_                             type=N length=8
i=16                                 type=  length=0
```

CAT Function

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

Category: Character

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

Tip: DBCS equivalent function is KSTRCAT in *SAS National Language Support (NLS): Reference Guide*.

Syntax

CAT(*item-1* <, …, *item-n*>)

Required Argument

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, 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

In a DATA step, 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 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 which 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 the DATA step except in WHERE clauses
- up to 65534 characters when CAT is called from the macro processor

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

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

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

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

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operator (||) and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operator. For more information, see “Length of Returned Variable” on page 264 .

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter, such as a blank or comma.

Function	Equivalent Code
CAT(OF X1-X4)	X1||X2||X3||X4
CATS(OF X1-X4)	TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))
CATT(OF X1-X4)	TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4)
CATX(SP, OF X1-X4)	TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4))

Example

The following example shows how the CAT function concatenates strings.

```
data _null_;
   x='  The 2012 Olym';
   y='pic Arts Festi';
   z='  val included works by D  ';
   a='ale Chihuly.';
   result=cat(x,y,z,a);
   put result $char.;
run;
```

SAS writes the following line to the log:

```
   ----+----1----+----2----+----3----+----4----+----5----+----6----+----7
     The 2012 Olympic Arts Festi  val included works by D  ale Chihuly.
```

See Also

Functions:

- “CATQ Function” on page 266
- “CATS Function” on page 270
- “CATT Function” on page 272
- “CATX Function” on page 274

CALL Routines:

- “CALL CATS Routine” on page 159
- “CALL CATT Routine” on page 161
- “CALL CATX Routine” on page 163

CATQ Function

Concatenates character or numeric values by using a delimiter to separate items and by adding quotation marks to strings that contain the delimiter.

Category: Character

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

The following example shows how the CATQ function concatenates strings.

```
options ls=110;
data _null_;
   result1=CATQ(' ',
                'noblanks',
                'one blank',
                12345,
                '  lots of blanks    ');
   result2=CATQ('CS',
                'Period (.)                ',
                'Ampersand (&)             ',
                'Comma (,)                 ',
                'Double quotation marks (") ',
                '   Leading Blanks');
   result3=CATQ('BCQT',
                'Period (.)                ',
                'Ampersand (&)             ',
                'Comma (,)                 ',
                'Double quotation marks (") ',
                '   Leading Blanks');
   result4=CATQ('ADT',
                '#=#',
                'Period (.)                ',
                'Ampersand (&)             ',
                'Comma (,)                 ',
                'Double quotation marks (") ',
                '   Leading Blanks');
   result5=CATQ('N',
                'ABC_123   ',
                '123    ',
                'ABC 123');
   put (result1-result5) (=/);
run;
```

SAS writes the following output to the log.

```
result1=noblanks "one blank" 12345 "  lots of blanks    "
result2=Period (.),Ampersand (&),"Comma (,)",Double quotation marks (),Leading
Blanks
result3=Period (.),Ampersand (&),"Comma (,)",'Double quotation marks (')',"
Leading Blanks"
result4="Period (.)"#=#'Ampersand (&)'#=#"Comma (,)"#=#'Double quotation marks
(")'#=#"   Leading Blanks"
result5=ABC_123 "123"n "ABC 123"n
```

See Also

Functions:

- "CAT Function" on page 263
- "CATS Function" on page 270
- "CATT Function" on page 272
- "CATX Function" on page 274

CALL Routines:

- "CALL CATS Routine" on page 159
- "CALL CATT Routine" on page 161
- "CALL CATX Routine" on page 163

CATS Function

Removes leading and trailing blanks, and returns a concatenated character string.

Category: Character

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

Syntax

CATS(*item-1* <, …, *item-n*>)

Required Argument

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

Length of Returned Variable

In a DATA step, if the CATS function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. If the concatenation operator (||) returns a value to a variable that has not previously been assigned a length, then that variable is given a length that is the sum of the lengths of the values which 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 the DATA step except in WHERE clauses

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

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

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

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

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

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operator (||) and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operator. For more information, see "Length of Returned Variable" on page 270.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter, such as a blank or comma.

Function	Equivalent Code
`CAT(OF X1-X4)`	`X1||X2||X3||X4`
`CATS(OF X1-X4)`	`TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))`
`CATT(OF X1-X4)`	`TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4)`
`CATX(SP, OF X1-X4)`	`TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4))`

Example

The following example shows how the CATS function concatenates strings.

```
data _null_;
   x='  The   Olym';
   y='pic Arts Festi';
   z='  val includes works by D  ';
```

```
   a='ale Chihuly.';
   result=cats(x,y,z,a);
   put result $char.;
run;
```

The following line is written to the SAS log:

```
   ----+----1----+----2----+----3----+----4----+----5----+----6
   The   Olympic Arts Festival includes works by Dale Chihuly.
```

See Also

Functions:

- "CAT Function" on page 263
- "CATQ Function" on page 266
- "CATT Function" on page 272
- "CATX Function" on page 274

CALL Routines:

- "CALL CATS Routine" on page 159
- "CALL CATT Routine" on page 161
- "CALL CATX Routine" on page 163

CATT Function

Removes trailing blanks, and returns a concatenated character string.

Category: Character

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

Syntax

CATT(*item-1* <, … *item-n*>)

Required Argument

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

In a DATA step, 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 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 which 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 the DATA step except in WHERE clauses
- up to 65534 characters when CATT is called from the macro processor

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

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

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

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

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operator (||) and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operator. For more information, see “Length of Returned Variable” on page 272.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter, such as a blank or comma.

Function	Equivalent Code
`CAT(OF X1-X4)`	`X1||X2||X3||X4`
`CATS(OF X1-X4)`	`TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))`
`CATT(OF X1-X4)`	`TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4)`
`CATX(SP, OF X1-X4)`	`TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4))`

Example

The following example shows how the CATT function concatenates strings.

```
data _null_;
   x='  The   Olym';
   y='pic Arts Festi';
   z='  val includes works by D  ';
   a='ale Chihuly.';
   result=catt(x,y,z,a);
   put result $char.;
run;
```

The following line is written to the SAS log:

```
   ----+----1----+----2----+----3----+----4----+----5----+----6----+----7
     The   Olympic Arts Festi  val includes works by Dale Chihuly.
```

See Also

Functions:

- “CAT Function” on page 263
- “CATQ Function” on page 266
- “CATS Function” on page 270
- “CATX Function” on page 274

CALL Routines:

- “CALL CATS Routine” on page 159
- “CALL CATT Routine” on page 161
- “CALL CATX Routine” on page 163

CATX Function

Removes leading and trailing blanks, inserts delimiters, and returns a concatenated character string.

Category: Character

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

Syntax

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

Required Arguments

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

item
specifies a constant, variable, or expression, either character or numeric. If *item* is numeric, then its value is converted to a character string by using the BEST*w*. format. In this case, SAS does not write a note to the log. For more information, see "The Basics" on page 275.

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

In a DATA step, if the CATX 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 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 which are being concatenated.

Length of Returned Variable: Special Cases

The CATX function returns a value to a variable, or returns a value in a temporary buffer. The value that is returned from the CATX function has 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 CATX is called from the macro processor

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

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

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

Comparisons

The results of the CAT, CATS, CATT, and CATX functions are *usually* equivalent to results that are produced by certain combinations of the concatenation operator (||) and the TRIM and LEFT functions. However, the default length for the CAT, CATS, CATT, and CATX functions is different from the length that is obtained when you use the concatenation operator. For more information, see "Length of Returned Variable" on page 275.

Using the CAT, CATS, CATT, and CATX functions is faster than using TRIM and LEFT, and you can use them with the OF syntax for variable lists in calling environments that support variable lists.

Note: In the case of variables that have missing values, the concatenation produces different results. See"Example 2: Concatenating Strings That Have Missing Values" on page 276.

The following table shows equivalents of the CAT, CATS, CATT, and CATX functions. The variables X1 through X4 specify character variables, and SP specifies a delimiter, such as a blank or comma.

Function	Equivalent Code
CAT(OF X1-X4)	X1||X2||X3||X4
CATS(OF X1-X4)	TRIM(LEFT(X1))||TRIM(LEFT(X2))||TRIM(LEFT(X3))|| TRIM(LEFT(X4))
CATT(OF X1-X4)	TRIM(X1)||TRIM(X2)||TRIM(X3)||TRIM(X4)
CATX(SP, OF X1-X4)	TRIM(LEFT(X1))||SP||TRIM(LEFT(X2))||SP|| TRIM(LEFT(X3))||SP||TRIM(LEFT(X4))

Examples

Example 1: Concatenating Strings That Have No Missing Values

The following example shows how the CATX function concatenates strings the have no missing values.

```
data _null_;
   separator='%%$%%';
   x='The Olympic  ';
   y='   Arts Festival ';
   z='   includes works by ';
   a='Dale Chihuly.';
   result=catx(separator,x,y,z,a);
   put result $char.;
run;
```

The following line is written to the SAS log:

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7
The Olympic%%$%%Arts Festival%%$%%includes works by%%$%%Dale Chihuly.
```

Example 2: Concatenating Strings That Have Missing Values

The following example shows how the CATX function concatenates strings that contain missing values.

```
data one;
   length x1-x4 $1;
   input x1-x4;
   datalines;
A B C D
E . F G
```

```
H . . J
;
run;
data two;
   set one;
   SP='^';
   test1=catx(sp, of x1-x4);
   test2=trim(left(x1)) || sp || trim(left(x2)) || sp || trim(left(x3)) || sp ||
      trim(left(x4));
run;

proc print data=two;
run;
```

Display 2.20 *Using CATX with Missing Values*

The SAS System

Obs	x1	x2	x3	x4	SP	test1	test2
1	A	B	C	D	^	A^B^C^D	A^B^C^D
2	E		F	G	^	E^F^G	E^ ^F^G
3	H			J	^	H^J	H^ ^ ^J

See Also

Functions:

- “CAT Function” on page 263
- “CATQ Function” on page 266
- “CATS Function” on page 270
- “CATT Function” on page 272

CALL Routines:

- “CALL CATS Routine” on page 159
- “CALL CATT Routine” on page 161
- “CALL CATX Routine” on page 163

CDF Function

Returns a value from a cumulative probability distribution.

Category: Probability

Note: The QUANTILE function returns the quantile from a distribution that you specify. The QUANTILE function is the inverse of the CDF function. For more information, see “QUANTILE Function” on page 799.

Syntax

CDF (*distribution*,*quantile*<,*parm-1, ... ,parm-k*>)

Required Arguments

distribution

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	`BERNOULLI`
Beta	`BETA`
Binomial	`BINOMIAL`
Cauchy	`CAUCHY`
Chi-Square	`CHISQUARE`
Exponential	`EXPONENTIAL`
F	`F`
Gamma	`GAMMA`
Generalized Poisson	`GENPOISSON`
Geometric	`GEOMETRIC`
Hypergeometric	`HYPERGEOMETRIC`
Laplace	`LAPLACE`
Logistic	`LOGISTIC`
Lognormal	`LOGNORMAL`
Negative binomial	`NEGBINOMIAL`
Normal	`NORMAL` | `GAUSS`
Normal mixture	`NORMALMIX`
Pareto	`PARETO`
Poisson	`POISSON`
T	`T`

Distribution	Argument
Tweedie (p. 290)	**TWEEDIE**
Uniform	**UNIFORM**
Wald (inverse Gaussian)	**WALD** | **IGAUSS**
Weibull	**WEIBULL**

Note Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of the random variable.

Optional Argument

parm-1, … ,parm-k
are optional constants, variables, or expressions that specify *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

See "Details" on page 279 for complete information about these parameters.

Details

The CDF function computes the left cumulative distribution function from various continuous and discrete probability distributions.

Bernoulli Distribution

CDF('BERNOULLI',*x*,*p*)

Arguments

x
is a numeric random variable.

p
is a numeric probability of success.

Range $0 \leq p \leq 1$

Details

The CDF function for the Bernoulli distribution returns the probability that an observation from a Bernoulli distribution, with probability of success equal to *p*, is less than or equal to *x*. The equation follows:

$$CDF('BERN', x, p) = \begin{cases} 0 & x < 0 \\ 1 - p & 0 \leq x < 1 \\ 1 & x \geq 1 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Beta Distribution

CDF('BETA',*x*,*a*,*b*<,*l*,*r*>)

Arguments

x

is a numeric random variable.

a

is a numeric shape parameter.

Range $a > 0$

b

is a numeric shape parameter.

Range $b > 0$

l

is the numeric left location parameter.

Default 0

r

is the right location parameter.

Default 1

Range $r > l$

Details

The CDF function for the beta distribution returns the probability that an observation from a beta distribution, with shape parameters *a* and *b*, is less than or equal to *v*. The following equation describes the CDF function of the beta distribution:

$$CDF('BETA', x, a, b, l, r) = \begin{cases} 0 & x \le l \\ \dfrac{1}{\beta(a, b)} \displaystyle\int_{l}^{x} \frac{(v - l)^{a-1}(r - v)^{b-1}}{(r - l)^{a+b-1}} dv & l < x \le r \\ 1 & x > r \end{cases}$$

The following relationship applies to the preceding equation:

$$\beta(a, b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a + b)}$$

The following relationship applies to the preceding equation:

$$\Gamma(a) = \int_{0}^{\infty} x^{a-1} e^{-x} dx$$

Binomial Distribution

CDF('BINOMIAL',*m*,*p*,*n*)

Arguments

m

is an integer random variable that counts the number of successes.

Range $m = 0, 1, ...$

p

is a numeric probability of success.

Range $0 \le p \le 1$

n

is an integer parameter that counts the number of independent Bernoulli trials.

Range $n = 0, 1, ...$

Details

The CDF function for the binomial distribution returns the probability that an observation from a binomial distribution, with parameters p and n, is less than or equal to m. The equation follows:

$$CDF('BINOM', m, p, n) = \begin{cases} 0 & m < 0 \\ \sum_{j=0}^{m} \binom{N}{j} p^{j}(1-p)^{n-j} & 0 \le m \le n \\ 1 & m > n \end{cases}$$

Note: There are no *location* or *scale* parameters for the binomial distribution.

Cauchy Distribution

CDF('CAUCHY',*x*<,*θ*><,*λ*>)

Arguments

x

is a numeric random variable.

θ

is a numeric location parameter.

Default 0

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the Cauchy distribution returns the probability that an observation from a Cauchy distribution, with the location parameter θ and the scale parameter λ, is less than or equal to x. The equation follows:

$$CDF('CAUCHY', x, \theta, \lambda) = \frac{1}{2} + \frac{1}{\pi} tan^{-1}\left(\frac{x-\theta}{\lambda}\right)$$

Chi-Square Distribution

CDF('CHISQUARE',*x*,*df* <,*nc*>)

Arguments

x
is a numeric random variable.

df
is a numeric degrees of freedom parameter.

Range $df > 0$

nc
is an optional numeric non-centrality parameter.

Range $nc \geq 0$

Details

The CDF function for the chi-square distribution returns the probability that an observation from a chi-square distribution, with *df* degrees of freedom and non-centrality parameter *nc*, is less than or equal to *x*. This function accepts non-integer degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central chi-square distribution. In the following equation, let $v = df$ and let $\lambda = nc$. The following equation describes the CDF function of the chi-square distribution:

$$CDF('CHISQ', x, v, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} P_c(x, v+2j) & x \geq 0 \end{cases}$$

In the equation, $P_c(.,.)$ denotes the probability from the central chi-square distribution:

$$P_c(x, a) = P_g\left(\frac{x}{2}, \frac{a}{2}\right)$$

In the equation, $P_g(y,b)$ is the probability from the gamma distribution given by the equation:

$$P_g(y, b) = \frac{1}{\Gamma(b)} \int_0^y e^{-v} v^{b-1} dv$$

Exponential Distribution

CDF('EXPONENTIAL',*x* <,*λ*>)

Arguments

x
is a numeric random variable.

λ
is a scale parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the exponential distribution returns the probability that an observation from an exponential distribution, with the scale parameter λ, is less than or equal to *x*. The equation follows:

$$CDF('EXPO', x, \lambda) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\frac{x}{\lambda}} & x \geq 0 \end{cases}$$

F Distribution

CDF('F',*x*,*ndf*,*ddf* <,*nc*>)

Arguments

x

is a numeric random variable.

ndf

is a numeric numerator degrees of freedom parameter.

Range $ndf > 0$

ddf

is a numeric denominator degrees of freedom parameter.

Range $ddf > 0$

nc

is a numeric non-centrality parameter.

Range $nc \geq 0$

Details

The CDF function for the *F* distribution returns the probability that an observation from an *F* distribution, with *ndf* numerator degrees of freedom, *ddf* denominator degrees of freedom, and non-centrality parameter *nc*, is less than or equal to *x*. This function accepts non-integer degrees of freedom for *ndf* and *ddf*. If *nc* is omitted or equal to zero, the value returned is from a central *F* distribution. In the following equation, let $v_1 = ndf$, let $v_2 = ddf$, and let $\lambda = nc$. The following equation describes the CDF function of the *F* distribution:

$$CDF('F', x, v_1, v_2, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} P_F(x, v_1 + 2j, v_2) & x \geq 0 \end{cases}$$

In the equation, $P_f(f,u_1,u_2)$ is the probability from the central *F* distribution with

$$P_F(x, u_1, u_2) = P_B\left(\frac{u_1 x}{u_1 x + u_2}, \frac{u_1}{2}, \frac{u_2}{2}\right)$$

and $P_B(x,a,b)$ is the probability from the standard beta distribution.

Note: There are no *location* or *scale* parameters for the *F* distribution.

Gamma Distribution

CDF('GAMMA',*x*,*a*<,*λ*>)

Arguments

x
is a numeric random variable.

a
is a numeric shape parameter.

Range $a > 0$

λ
is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the gamma distribution returns the probability that an observation from a gamma distribution, with shape parameter *a* and scale parameter λ, is less than or equal to *x*. The equation follows:

$$CDF('GAMMA', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ \dfrac{1}{\lambda^a \Gamma(a)} \displaystyle\int_0^x v^{a-1} e^{-\frac{v}{\lambda}} dv & x \geq 0 \end{cases}$$

Generalized Poisson Distribution

CDF ('GENPOISSON', *x*, *θ*, *η*)

Arguments

x
is an integer random variable.

θ
specifies a shape parameter.

Range ≤5 and >0

η
specifies a shape parameter.

Range ≥0 and <0.95

Tip When η =0, the distribution is the Poisson distribution with a mean and variance of θ. When η>0, the mean is $\theta \div (1 - \eta)$ and the variance is $\theta \div (1 - \eta)^3$.

Details

The probability mass function for the generalized Poisson distribution follows:

$$f(x; \theta, \eta) = \theta(\theta + \eta x)^{x-1} e^{-\theta - \eta x} / x!, \quad x = 0, 1, 2, \ldots, \quad \theta > 0, \ 0 \leq \eta < 1$$

If η =0, then the generalized Poisson distribution becomes the standard Poisson distribution with shape parameter θ.

Geometric Distribution

CDF('GEOMETRIC',*m*,*p*)

Arguments

m

is a numeric random variable that denotes the number of failures.

Range $m = 0, 1, ...$

p

is a numeric probability of success.

Range $0 \leq p \leq 1$

Details

The CDF function for the geometric distribution returns the probability that an observation from a geometric distribution, with parameter p, is less than or equal to m. The equation follows:

$$CDF('GEOM', m, p) = \begin{cases} 0 & m < 0 \\ 1 - (1 - p)^{(m+1)} & m \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Hypergeometric Distribution

CDF('HYPER',*x*,*N*,*R*,*n*<,*o*>)

Arguments

x

is an integer random variable.

N

is an integer population size parameter.

Range $N = 1, 2, ...$

R

is an integer number of items in the category of interest.

Range $R = 0, 1, ..., N$

n

is an integer sample size parameter.

Range $n = 1, 2, ..., N$

o

is an optional numeric odds ratio parameter.

Range $o > 0$

Details

The CDF function for the hypergeometric distribution returns the probability that an observation from an extended hypergeometric distribution, with population size N, number of items R, sample size n, and odds ratio o, is less than or equal to x. If o is omitted or equal to 1, the value returned is from the usual hypergeometric distribution. The equation follows:

$$CDF('HYPER', x, N, R, n, o) = \begin{cases} 0 & x < max(0, R + n - N) \\ \dfrac{\sum_{i=0}^{x} \binom{R}{i}\binom{N-R}{n-i} o^i}{\sum_{j=max(0,R+n-N)}^{min(R,n)} \binom{R}{j}\binom{N-R}{n-j} o^j} & max(0, R + n - N) \le x \le min(R, n) \\ 1 & x > min(R, n) \end{cases}$$

Laplace Distribution

CDF('LAPLACE',*x*<,*θ*,*λ*>)

Arguments

x

is a numeric random variable.

θ

is a numeric location parameter.

Default 0

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the Laplace distribution returns the probability that an observation from the Laplace distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*. The equation follows:

$$CDF('LAPLACE', x, \theta, \lambda) = \begin{cases} \frac{1}{2} e^{\frac{(x-\theta)}{\lambda}} & x < \theta \\ 1 - \frac{1}{2} e^{\left(-\frac{(x-\theta)}{\lambda}\right)} & x \ge \theta \end{cases}$$

Logistic Distribution

CDF('LOGISTIC',*x*<,*θ*,*λ*>)

Arguments

x

is a numeric random variable.

θ

is a numeric location parameter

Default 0

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the Logistic distribution returns the probability that an observation from a Logistic distribution, with a location parameter θ and a scale parameter λ, is less than or equal to *x*. The equation follows:

$$CDF('LOGISTIC', x, \theta, \lambda) = \frac{1}{1 + e^{\left(-\frac{x-\theta}{\lambda}\right)}}$$

Lognormal Distribution

CDF('LOGNORMAL',*x*<,*θ*,*λ*>)

Arguments

x

is a numeric random variable.

θ

specifies a numeric log scale parameter. (e(θ) is a scale parameter.)

Default 0

λ

specifies a numeric shape parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the lognormal distribution returns the probability that an observation from a lognormal distribution, with the log scale parameter θ and the shape parameter λ, is less than or equal to *x*. The equation follows:

$$CDF('LOGN', x, \theta, \lambda) = \begin{cases} 0 & x \le 0 \\ \frac{1}{\lambda\sqrt{2\pi}} \int_{-\infty}^{\log(x)} e^{\left(-\frac{(v-\theta)^2}{2\lambda^2}\right)} dv & x > 0 \end{cases}$$

Negative Binomial Distribution

CDF('NEGBINOMIAL',*m*,*p*,*n*)

Arguments

m

is a positive integer random variable that counts the number of failures.

Range $m = 0, 1, \ldots$

p

is a numeric probability of success.

Range $0 \le p \le 1$

n
is a numeric value that counts the number of successes.

Range $n > 0$

Details

The CDF function for the negative binomial distribution returns the probability that an observation from a negative binomial distribution, with probability of success *p* and number of successes *n*, is less than or equal to *m*. The equation follows:

$$CDF('NEGB', m, p, n) = \begin{cases} 0 & m < 0 \\ p^n \sum_{j=0}^{m} \binom{n+j-1}{n-1} (1-p)^j & m \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the negative binomial distribution.

Normal Distribution

CDF('NORMAL',*x*<,*θ*,*λ*>)

Arguments

x
is a numeric random variable.

θ
is a numeric location parameter.

Default 0

λ
is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the Normal distribution returns the probability that an observation from the Normal distribution, with the location parameter θ and the scale parameter λ, is less than or equal to *x*. The equation follows:

$$CDF('NORMAL', x, \theta, \lambda) = \frac{1}{\lambda\sqrt{2\pi}} \int_{-\infty}^{x} e^{\left(-\frac{(v-\theta)^2}{2\lambda^2}\right)} dv$$

Normal Mixture Distribution

CDF('NORMALMIX',*x*,*n*,*p*,*m*,*s*)

Arguments

x
is a numeric random variable.

n
is the integer number of mixtures.

Range $n = 1, 2, \ldots$

p

is the n proportions, $p_1, p_2, \ldots, p_n$, where $\sum_{i=1}^{i=n} p_i = 1$.

Range $p = 0, 1, \ldots$

m

is the n means $m_1, m_2, \ldots, m_n$.

s

is the n standard deviations $s_1, s_2, \ldots, s_n$.

Range $s > 0$

Details

The CDF function for the normal mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to x. The equation follows:

$$CDF('NORMALMIX', x, n, p, m, s) = \sum_{i=1}^{i=n} p_i \, CDF('NORMAL', x, m_i, s_i)$$

Note: There are no *location* or *scale* parameters for the normal mixture distribution.

Pareto Distribution

CDF('PARETO',*x*,*a*<,*k*>)

Arguments

x

is a numeric random variable.

a

is a numeric shape parameter.

Range $a > 0$

k

is a numeric scale parameter.

Default 1

Range $k > 0$

Details

The CDF function for the Pareto distribution returns the probability that an observation from a Pareto distribution, with the shape parameter a and the scale parameter k, is less than or equal to x. The equation follows:

$$CDF('PARETO', x, a, k) = \begin{cases} 0 & x < k \\ 1 - \left(\frac{k}{x}\right)^a & x \geq k \end{cases}$$

Poisson Distribution

CDF('POISSON',*n*,*m*)

Arguments

n
is an integer random variable.

Range $n = 0, 1, ...$

m
is a numeric mean parameter.

Range $m > 0$

Details

The CDF function for the Poisson distribution returns the probability that an observation from a Poisson distribution, with mean *m*, is less than or equal to *n*. The equation follows:

$$CDF('POISSON', n, m) = \begin{cases} 0 & n < 0 \\ \sum_{i=0}^{n} e^{-m}\frac{m^i}{i!} & n \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the Poisson distribution.

T Distribution

CDF('T',*t*,*df*<,*nc*>)

Arguments

t
is a numeric random variable.

df
is a numeric degrees of freedom parameter.

Range $df > 0$

nc
is an optional numeric non-centrality parameter.

Details

The CDF function for the *T* distribution returns the probability that an observation from a *T* distribution, with degrees of freedom *df* and non-centrality parameter *nc*, is less than or equal to *x*. This function accepts non-integer degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central *T* distribution. In the following equation, let $\nu = df$ and let $\delta = nc$. The equation follows:

$$CDF('T', t, \nu, \delta) = \frac{1}{2^{(\nu/2-1)}\Gamma\left(\frac{\nu}{2}\right)}\int_0^{\infty} x^{\nu-1} e^{-\frac{1}{2}x^2} \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\frac{tx}{\sqrt{\nu}}} e^{-\frac{1}{2}(u-\delta)^2} du dx$$

Note: There are no *location* or *scale* parameters for the *T* distribution.

Tweedie Distribution

CDF ('TWEEDIE', *y*, *p*<,μ, φ>)

Arguments

y
is a random variable.

Range $y \geq 0$

Notes This argument is required.

When $p>1$, y is numeric. When $p=1$, y is an integer.

p

is the power parameter.

Range $p \geq 1$

Note This argument is required.

μ

is the mean.

Default 1

Range $\mu>0$

φ

is the dispersion parameter.

Default 1

Range $\varphi>0$

Details

The CDF function for the Tweedie distribution returns an exponential dispersion model with variance and mean related by the equation *variance* = $\phi \times \mu^{p}$.

The equation follows:

$$\int_0^y \frac{1}{y} \sum_{j=1}^{\infty} \left(\frac{y^{-j\alpha}(p-1)^{j\alpha}}{\phi^{j(1-\alpha)}(2-p)^{j} j!\,\Gamma(-j\alpha)} \right) e^{\left(\frac{1}{\phi} \left(y \frac{\mu^{1-p}-1}{1-p} - \frac{\mu^{2-p}-1}{2-p} \right) \right)} dy$$

The following relationship applies to the preceding equation:

$$\alpha = \frac{2-p}{1-p}$$

Note: The accuracy of computed Tweedie probabilities is highly dependent on the location in parameter space. Ten digits of accuracy are usually available except when p is near 2 or phi is near 0, in which case the accuracy might be as low as six digits.

Uniform Distribution

CDF('UNIFORM',*x*<,*l*,*r*>)

Arguments

x

is a numeric random variable.

l

is the numeric left location parameter.

Default 0

r
is the numeric right location parameter.

Default 1

Range $r > l$

Details

The CDF function for the uniform distribution returns the probability that an observation from a uniform distribution, with the left location parameter *l* and the right location parameter *r*, is less than or equal to *x*. The equation follows:

$$CDF('UNIFORM', x, l, r) = \begin{cases} 0 & x < l \\ \frac{x - l}{r - l} & l \le x < r \\ 1 & x \ge r \end{cases}$$

Note: The default values for *l* and *r* are 0 and 1, respectively.

Wald (Inverse Gaussian) Distribution

CDF('WALD',*x*,*λ*<,*μ*>)

CDF('IGAUSS',*x*,*λ*<,*μ*>)

Arguments

x
is a numeric random variable.

λ
is a numeric shape parameter.

Range $\lambda > 0$

μ
is the mean.

Default 1

Range $\mu > 0$

Details

The CDF function for the Wald distribution returns the probability that an observation from a Wald distribution, with shape parameter λ, is less than or equal to *x*. The equation follows:

$$Fx(x) = \Phi\left\{\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} - 1\right)\right\} + e^{2\lambda/\mu}\Phi\left\{-\sqrt{\frac{\lambda}{x}}\left(\frac{x}{\mu} + 1\right)\right\}$$

In the equation, $\Phi(.)$ is the standard normal cumulative distribution function. When $x \le 0$, the CDF is 0.

Weibull Distribution

CDF('WEIBULL',*x*,*a*<,*λ*>)

Arguments

x
is a numeric random variable.

a

is a numeric shape parameter.

Range $a > 0$

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The CDF function for the Weibull distribution returns the probability that an observation from a Weibull distribution, with the shape parameter *a* and the scale parameter λ, is less than or equal to *x*. The equation follows:

$$CDF('WEIBULL', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\left(\frac{x}{\lambda}\right)^a} & x \geq 0 \end{cases}$$

Example

The following SAS statements produce these results.

SAS Statement	Result
`y=cdf('BERN',0,.25);`	0.75
`y=cdf('BETA',0.2,3,4);`	0.09888
`y=cdf('BINOM',4,.5,10);`	0.37695
`y=cdf('CAUCHY',2);`	0.85242
`y=cdf('CHISQ',11.264,11);`	0.57858
`y=cdf('EXPO',1);`	0.63212
`y=cdf('F',3.32,2,3);`	0.82639
`y=cdf('GAMMA',1,3);`	0.080301
`y=cdf('GENPOISSON',9,1,.7);`	0.906162963
`y=cdf('HYPER',2,200,50,10);`	0.52367
`y=cdf('LAPLACE',1);`	0.81606
`y=cdf('LOGISTIC',1);`	0.73106
`y=cdf('LOGNORMAL',1);`	0.5

SAS Statement	Result
`y=cdf('NEGB',1,.5,2);`	0.5
`y=cdf('NORMAL',1.96);`	0.97500
`y=cdf('NORMALMIX',2.3,3,.33,.33,.34, .5,1.5,2.5,.79,1.6,4.3);`	0.7181
`y=cdf('PARETO',1,1);`	0
`y=cdf('POISSON',2,1);`	0.91970
`y=cdf('T',.9,5);`	0.79531
`y=cdf('TWEEDIE',.8,5);`	0.5917629164
`y=cdf('UNIFORM',0.25);`	0.25
`y=cdf('WALD',1,2);`	0.62770
`y=cdf('WEIBULL',1,2);`	0.63212

See Also

Functions:

- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “QUANTILE Function” on page 799
- “SDF Function” on page 856
- “SQUANTILE Function” on page 881

CEIL Function

Returns the smallest integer that is greater than or equal to the argument, fuzzed to avoid unexpected floating-point results.

Category: Truncation

Syntax

CEIL (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

If the argument is within 1E-12 of an integer, the function returns that integer.

Comparisons

Unlike the CEILZ function, the CEIL function fuzzes the result. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function you might get unexpected results.

Example

The following SAS statements produce these results.

SAS Statement	Result

`var1=2.1;`
`a=ceil(var1);`
`put a;`
	`3`

`b=ceil(-2.4);` `put b;`	`-2`
`c=ceil(1+1.e-11);` `put c;`	`2`
`d=ceil(-1+1.e-11);` `put d;`	`0`
`e=ceil(1+1.e-13);` `put e;`	`1`
`f=ceil(223.456);` `put f;`	`224`
`g=ceil(763);` `put g;`	`763`
`h=ceil(-223.456);` `put h;`	`-223`

See Also

Functions:

- “CEILZ Function” on page 296

CEILZ Function

Returns the smallest integer that is greater than or equal to the argument, using zero fuzzing.

Category: Truncation

Syntax

CEILZ (*argument*)

Required Argument

argument
is a numeric constant, variable, or expression.

Details

Unlike the CEIL function, the CEILZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the CEIL function fuzzes the result to be equal to that integer. The CEILZ function does not fuzz the result. Therefore, with the CEILZ function you might get unexpected results.

Example

The following SAS statements produce these results.

SAS Statement	Result
`a=ceilz(2.1);` `put a;`	`3`
`b=ceilz(-2.4);` `put b;`	`-2`
`c=ceilz(1+1.e-11);` `put c;`	`2`
`d=ceilz(-1+1.e-11);` `put d;`	`0`
`e=ceilz(1+1.e-13);` `put e;`	`2`
`f=ceilz(223.456);` `put f;`	`224`
`g=ceilz(763);` `put g;`	`763`
`h=ceilz(-223.456);` `put h;`	`-223`

See Also

Functions:

- "CEIL Function" on page 294
- "FLOOR Function" on page 480
- "FLOORZ Function" on page 481
- "INT Function" on page 555
- "INTZ Function" on page 596
- "ROUND Function" on page 833
- "ROUNDE Function" on page 840
- "ROUNDZ Function" on page 843

CEXIST Function

Verifies the existence of a SAS catalog or SAS catalog entry.

Category: SAS File I/O

Syntax

CEXIST(*entry*<,'U'>)

Required Argument

entry
is a character constant, variable, or expression that specifies a SAS catalog, or the name of an entry in a catalog. If the *entry* value is a one- or two-level name, then it is assumed to be the name of a catalog. Use a three- or four-level name to test for the existence of an entry within a catalog.

Optional Argument

'U'
tests whether the catalog can be opened for updating.

Details

CEXIST returns 1 if the SAS catalog or catalog entry exists, or 0 if the SAS catalog or catalog entry does not exist.

Examples

Example 1: Verifying the Existence of an Entry in a Catalog

This example verifies the existence of the entry X.PROGRAM in LIB.CAT1:

```
data _null_;
   if cexist("lib.cat1.x.program") then
   put "Entry X.PROGRAM exists";
```

```
run;
```

Example 2: Determining if a Catalog Can Be Opened for Update

This example tests whether the catalog LIB.CAT1 exists and can be opened for update. If the catalog does not exist, a message is written to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%if %sysfunc(cexist(lib.cat1,u)) %then
   %put The catalog LIB.CAT1 exists and can be opened for update.;
%else
   %put %sysfunc(sysmsg());
```

See Also

Functions:

- "EXIST Function" on page 396

CHAR Function

Returns a single character from a specified position in a character string.

Category: Character

Syntax

CHAR(*string*, *position*)

Required Arguments

string
specifies a character constant, variable, or expression.

position
is an integer that specifies the position of the character to be returned.

Details

In a DATA step, the default length of the target variable for the CHAR function is 1.

If *position* has a missing value, then CHAR returns a string with a length of 0. Otherwise, CHAR returns a string with a length of 1.

If *position* is less than or equal to 0, or greater than the length of the string, then CHAR returns a blank. Otherwise, CHAR returns the character at the specified position in the string.

Comparisons

The CHAR function returns the same result as SUBPAD(*string*, *position*, 1). While the results are the same, the default length of the target variable is different.

Example

The following example shows the results of using the CHAR function.

```
data test;
   retain string "abc";
   do position = -1 to 4;
      result=char(string, position);
      output;
   end;
run;

proc print noobs data=test;
run;
```

Display 2.21 *Output from the CHAR Function*

The SAS System

string	position	result
abc	-1	
abc	0	
abc	1	a
abc	2	b
abc	3	c
abc	4	

See Also

Functions:

- "FIRST Function" on page 479

CHOOSEC Function

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

Category: Character

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

Syntax

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

Required Arguments

index-expression
specifies a numeric constant, variable, or expression.

selection
specifies a character constant, variable, or expression. The value of this argument is returned by the CHOOSEC function.

Details

Length of Returned Variable

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

The Basics

The CHOOSEC function uses the value of *index-expression* to select from the arguments that follow. For example, if *index-expression* is three, CHOOSEC returns the value of *selection-3*. If the first argument is negative, the function counts backwards 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:

```
data _null_;
   Fruit=choosec(1,'apple','orange','pear','fig');
   Color=choosec(3,'red','blue','green','yellow');
   Planet=choosec(2,'Mars','Mercury','Uranus');
   Sport=choosec(-3,'soccer','baseball','gymnastics','skiing');
   put Fruit= Color= Planet= Sport=;
run;
```

SAS writes the following line to the log:

```
Fruit=apple Color=green Planet=Mercury Sport=baseball
```

See Also

Functions:

- “CHOOSEN Function” on page 300

CHOOSEN Function

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

Category: Character

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

Syntax

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

Required Arguments

index-expression
specifies a numeric constant, variable, or expression.

selection
specifies a numeric constant, variable, or expression. The value of this argument is returned by the CHOOSEN function.

Details

The CHOOSEN function uses the value of *index-expression* to select from the arguments that follow. For example, if *index-expression* is 3, CHOOSEN returns the value of *selection-3*. If the first argument is negative, the function counts backwards from the list of arguments, and returns that value.

Comparisons

The CHOOSEN function is similar to the CHOOSEC function except that CHOOSEN returns a numeric value while CHOOSEC returns a character value.

Example

The following example shows how CHOOSEN chooses from a series of values:

```
data _null_;
   ItemNumber=choosen(5,100,50,3784,498,679);
   Rank=choosen(-2,1,2,3,4,5);
   Score=choosen(3,193,627,33,290,5);
   Value=choosen(-5,-37,82985,-991,3,1014,-325,3,54,-618);
   put ItemNumber= Rank= Score= Value=;
run;
```

SAS writes the following line to the log:

```
ItemNumber=679 Rank=4 Score=33 Value=1014
```

See Also

Functions:

- “CHOOSEC Function” on page 299

CINV Function

Returns a quantile from the chi-square distribution.

Category: Quantile

Syntax

CINV (*p*,*df*<,*nc*>)

Required Arguments

p
is a numeric probability.

Range $0 \leq p < 1$

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Optional Argument

nc
is a numeric noncentrality parameter.

Range $nc \geq 0$

Details

The CINV function returns the p^{th} quantile from the chi-square distribution with degrees of freedom *df* and a noncentrality parameter *nc*. The probability that an observation from a chi-square distribution is less than or equal to the returned quantile is *p*. This function accepts a noninteger degrees of freedom parameter *df*.

If the optional parameter *nc* is not specified or has the value 0, the quantile from the central chi-square distribution is returned. The noncentrality parameter *nc* is defined such that if X is a normal random variable with mean μ and variance 1, X^2 has a noncentral chi-square distribution with *df*=1 and $nc = \mu^2$.

CAUTION:
For large values of nc, the algorithm could fail. In that case, a missing value is returned.

Note: CINV is the inverse of the PROBCHI function.

Example

The first statement following shows how to find the 95th percentile from a central chi-square distribution with 3 degrees of freedom. The second statement shows how to find the 95th percentile from a noncentral chi-square distribution with 3.5 degrees of freedom and a noncentrality parameter equal to 4.5.

SAS Statement	Result
`q1=cinv(.95,3);`	`7.8147279033`
`a2=cinv(.95,3.5,4.5);`	`7.504582117`

See Also

Functions:

- "QUANTILE Function" on page 799

CLOSE Function

Closes a SAS data set.

Category: SAS File I/O

Syntax

CLOSE(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

Details

CLOSE returns zero if the operation was successful, or returns a non-zero value if it was not successful. Close all SAS data sets as soon as they are no longer needed by the application.

Note: All data sets opened within a DATA step are closed automatically at the end of the DATA step.

Example

This example uses OPEN to open the SAS data set PAYROLL. If the data set opens successfully, indicated by a positive value for the variable PAYID, the example uses CLOSE to close the data set.

```
%let payid=%sysfunc(open(payroll,is));
   macro statements
%if &payid > 0 %then
   %let rc=%sysfunc(close(&payid));
```

See Also

Functions:

- "OPEN Function" on page 716

CMISS Function

Counts the number of missing arguments.

Category: Descriptive Statistics

Syntax

CMISS(*argument-1* <, *argument-2,*…>)

Required Argument

argument
specifies a constant, variable, or expression. *Argument* can be either a character value or a numeric value.

Details

A character expression is counted as missing if it evaluates to a string that contains all blanks or has a length of zero.

A numeric expression is counted as missing if it evaluates to a numeric missing value: ., ._, .A, … , .Z.

Comparisons

The CMISS function does not convert any argument. The NMISS function converts all arguments to numeric values.

See Also

Functions:

- "NMISS Function" on page 683
- "MISSING Function" on page 662

CNONCT Function

Returns the noncentrality parameter from a chi-square distribution.

Category: Mathematical

Syntax

CNONCT(*x*,*df*,*prob*)

Required Arguments

x
is a numeric random variable.

Range $x \geq 0$

df
is a numeric degrees of freedom parameter.

Range $df > 0$

prob
is a probability.

Range $0 < prob < 1$

Details

The CNONCT function returns the nonnegative noncentrality parameter from a noncentral chi-square distribution whose parameters are *x*, *df*, and *nc*. If *prob* 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

$$P_c(x \mid \mathrm{df},\ nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_c(x \mid \mathrm{df},\ \mathrm{nc}) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} P_g\left(\frac{x}{2} \,\middle|\, \frac{df}{2} + j\right)$$

The following relationship applies to the preceding equation:

$$P_g(x \mid a)$$

is the probability from the gamma distribution given by

$$P_g(x \mid a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} \varepsilon^{-t} dt$$

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

Example

```
data work;
   x=2;
   df=4;
   do nc=1 to 3 by .5;
      prob=probchi(x,df,nc);
      ncc=cnonct(x,df,prob);
      output;
   end;
run;
proc print;
run;
```

Display 2.22 *Computations of the Noncentrality Parameters from the Chi-squared Distribution*

The SAS System

Obs	x	df	nc	prob	ncc
1	2	4	1.0	0.18611	1.0
2	2	4	1.5	0.15592	1.5
3	2	4	2.0	0.13048	2.0
4	2	4	2.5	0.10907	2.5
5	2	4	3.0	0.09109	3.0

COALESCE Function

Returns the first non-missing value from a list of numeric arguments.

Category: Mathematical

Syntax

COALESCE(*argument-1*<..., *argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The Basics

COALESCE accepts one or more numeric arguments. The COALESCE function checks the value of each argument in the order in which they are listed and returns the first non-missing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all arguments are missing, then the COALESCE function returns a missing value.

Comparisons

The COALESCE function searches numeric arguments, whereas the COALESCEC function searches character arguments.

Example

The following statements produce these results.

SAS Statement	Result
`x = COALESCE(42, .);`	`42`
`y = COALESCE(.A, .B, .C);`	`.`
`z = COALESCE(., 7, ., ., 42);`	`7`

See Also

Functions:

- "COALESCEC Function" on page 307

COALESCEC Function

Returns the first non-missing value from a list of character arguments.

Category: Character

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

Syntax

COALESCEC(*argument-1*<..., *argument-n*>)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

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

The Basics

COALESCEC accepts one or more character arguments. The COALESCEC function checks the value of each argument in the order in which they are listed and returns the first non-missing value. If only one value is listed, then the COALESCEC function returns the value of that argument. A character value is considered missing if it has a length of zero or if all the characters are blank. If all the values of all arguments are missing, then the COALESCEC function returns a string with a length of zero.

Comparisons

The COALESCEC function searches character arguments, whereas the COALESCE function searches numeric arguments.

Example

The following statements produce these results.

SAS Statement	Result
`COALESCEC('', 'Hello')`	`Hello`
`COALESCEC ('', 'Goodbye', 'Hello')`	`Goodbye`

See Also

Functions:

- "COALESCE Function" on page 306

COLLATE Function

Returns a character string in ASCII or EBCDIC collating sequence.

Category: Character

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

See: "COLLATE Function: Windows" in *SAS Companion for Windows*
"COLLATE Function: UNIX" in *SAS Companion for UNIX Environments*

Syntax

COLLATE (*start-position*<,*end-position*>) | (*start-position*<,,*length*>)

Required Argument

start-position
specifies the numeric position in the collating sequence of the first character to be returned.

Interaction If you specify only *start-position*, COLLATE returns consecutive characters from that position to the end of the collating sequence or up to 255 characters, whichever comes first.

Optional Arguments

end-position
specifies the numeric position in the collating sequence of the last character to be returned.

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

Tips *end-position* must be larger than *start-position*

If you specify *end-position*, COLLATE returns all character values in the collating sequence between *start-position* and *end-position,* inclusive.

If you omit *end-position* and use *length*, mark the *end-position* place with a comma.

length
specifies the number of characters in the collating sequence.

Default 200

Tip If you omit *end-position*, use *length* to specify the length of the result explicitly.

Details

Length of Returned Variable

In a DATA step, if the COLLATE 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

If you specify both *end-position* and *length*, COLLATE ignores *length*. If you request a string longer than the remainder of the sequence, COLLATE returns a string through the end of the sequence.

Example

The following SAS statements produce these results.

SAS Statement	Result
ASCII	`----+----1----+---2--`

`x=collate(48,,10);`
`y=collate(48,57);`
`put @1 x @14 y;`
	`0123456789  0123456789`

EBCDIC	

`x=collate(240,,10);`
`y=collate(240,249);`
`put @1 x @14 y;`
	`0123456789  0123456789`

See Also

Functions:

- "BYTE Function" on page 149
- "RANK Function" on page 820

COMB Function

Computes the number of combinations of *n* elements taken *r* at a time.

Category: Combinatorial

Syntax

COMB(*n*,*r*)

Required Arguments

n
is a nonnegative integer that represents the total number of elements from which the sample is chosen.

r
is a nonnegative integer that represents the number of chosen elements.

Restriction $r \leq n$

Details

The mathematical representation of the COMB function is given by the following equation:

$$COMB(n,r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

In the preceding equation, $n \geq 0$, $r \geq 0$, 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 produces this result.

SAS Statement	Result
`x=comb(5,1);`	5

See Also

Functions:

- “FACT Function” on page 399
- “PERM Function” on page 743
- “LCOMB Function” on page 615

COMPARE Function

Returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference.

Category: Character

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

Tip: DBCS equivalent function is KCOMPARE in *SAS National Language Support (NLS): Reference Guide*. See also “DBCS Compatibility” on page 312 .

Syntax

COMPARE(*string–1*, *string–2*<,*modifiers*>)

Required Arguments

string–1
specifies a character constant, variable, or expression.

string–2
specifies a character constant, variable, or expression.

Optional Argument

modifier
specifies a character string that can modify the action of the COMPARE function. You can use one or more of the following characters as a valid modifier:

i or I	ignores the case in *string–1* and *string–2*.
l or L	removes leading blanks in *string–1* and *string–2* before comparing the values.
n or N	removes quotation marks from any argument that is a name literal and ignores the case of *string–1* and *string–2*. A name literal is a name token that is expressed as a string within quotation marks, followed by the uppercase or lowercase letter *n*. Name literals enable you to use special characters (including blanks) that are not otherwise allowed in SAS data set or variable names. For COMPARE to recognize a string as a name literal, the first character must be a quotation mark.
: (colon)	truncates the longer of *string–1* or *string–2* to the length of the shorter string, or to 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.

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, which is documented in *SAS National Language Support (NLS): Reference Guide*. There are minor differences between the COMPARE and KCOMPARE functions. While both functions accept varying numbers of arguments, 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*)

Examples

Example 1: Understanding the Order of Comparisons When Comparing Two Strings

The following example compares two strings by using the COMPARE function.

```
data test;
   infile datalines missover;
   input string1 $char8. string2 $char8. modifiers $char8.;
   result=compare(string1, string2, modifiers);
   datalines;
1234567812345678
123     abc
abc     abx
xyz     abcdef
aBc     abc
aBc     AbC     i
   abc  abc
   abc  abc     l
 abc       abx
 abc       abx  l
```

```
ABC     'abc'n
ABC     'abc'n  n
 '$12'n $12     n
 '$12'n $12     nl
 '$12'n $12     ln
;

proc print data=test;
run;
```

Display 2.23 *Results of Comparing Two Strings by Using the COMPARE Function*

The SAS System

Obs	string1	string2	modifiers	result
1	12345678	12345678		0
2	123	abc		-1
3	abc	abx		-3
4	xyz	abcdef		1
5	aBc	abc		-2
6	aBc	AbC	i	0
7	abc	abc		-1
8	abc	abc	l	0
9	abc	abx		2
10	abc	abx	l	-3
11	ABC	'abc'n		1
12	ABC	'abc'n	n	0
13	'$12'n	$12	n	-1
14	'$12'n	$12	nl	1
15	'$12'n	$12	ln	0

Example 2: Truncating Strings Using the COMPARE Function

The following example uses the : (colon) modifier to truncate strings.

```
data test2;
   pad1=compare('abc','abc            ');
   pad2=compare('abc','abcdef         ');
   truncate1=compare('abc','abcdef',':');
   truncate2=compare('abcdef','abc',':');
   blank=compare('','abc',          ':');
run;

proc print data=test2 noobs;
run;
```

Display 2.24 *Results of Using the Truncation Modifier*

The SAS System

pad1	pad2	truncate1	truncate2	blank
0	-4	0	0	-1

See Also

Functions:

- "COMPGED Function" on page 317
- "COMPLEV Function" on page 323

CALL Routines:

- "CALL COMPCOST Routine" on page 165

COMPBL Function

Removes multiple blanks from a character string.

Category: Character

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

Syntax

COMPBL(*source*)

Required Argument

source
specifies a character constant, variable, or expression to compress.

Details

Length of Returned Variable

In a DATA step, if the COMPBL function returns a value to a variable that has not previously been assigned a length, then the length of that variable defaults to the length of the first argument.

The Basics

The COMPBL function removes multiple blanks in a character string by translating each occurrence of two or more consecutive blanks into a single blank.

Comparisons

The COMPRESS function removes every occurrence of the specific character from a string. If you specify a blank as the character to remove from the source string, the COMPRESS function removes all blanks from the source string, while the COMPBL function compresses multiple blanks to a single blank and has no effect on a single blank.

Example

The following SAS statements produce these results.

SAS Statement	Result
`----+----1----+---2--`	

`string='Hey`
`  Diddle  Diddle';`
`string=compbl(string);`
`put string;`
	`Hey Diddle Diddle`

`string='125    E Main St';`
`length address $10;`
`address=compbl(string);`
`put address;`
	`125 E Main`

See Also

Functions:

- "COMPRESS Function" on page 327

COMPFUZZ Function

Performs a fuzzy comparison of two numeric values.

Category: Mathematical

Syntax

COMPFUZZ(*value1*, *value2* <, *fuzz* <, *scale*> >)

Required Arguments

value1
specifies the first of two numeric values to be compared.

value2
specifies the second numeric value to be compared.

Optional Arguments

fuzz
is a nonnegative numeric value that specifies the relative threshold for comparisons. Values greater than or equal to one are treated as multiples of the machine precision.

Default 1024

scale
specifies the scale factor.

Default MAX (ABS (value1), ABS (value2))

Details

The COMPFUZZ function returns the following values if you specify all four arguments:

- -1 if *value1* < *value2* - threshold
- 0 if ABS(*value1* - *value2*) ≤ threshold
- 1 if *value1* > *value2* + threshold

The following relationships exist:

- threshold = *fuzz* * ABS(*scale*) if 0 ≤ *fuzz* < 1
- threshold = *fuzz* * ABS(*scale*) * CONSTANT('MACEPS') if 1 ≤ *fuzz* < 1 / CONSTANT('MACEPS')

COMPFUZZ avoids floating point underflow or overflow.

Comparisons

The COMPFUZZ function compares two floating point numbers and returns a value based on the comparison. The ROUND function rounds an argument to a value that is very close to a multiple of a second argument. The result might not be an exact multiple of the second argument.

Example

In floating point arithmetic, the value of a sum sometimes depends on the order in which the numbers are added. One approximate bound for the floating point error in the computation of a sum of *n* numbers, x1 through x*n* is expressed by the following formula:

```
n * machine_precision * sum (abs(x1) + ... + abs(xn))
```

To compare sums of *n* floating point numbers with the COMPFUZZ function, you can therefore use *n* as the fuzz value and the sum of the absolute values as the scale factor, as shown in the following DATA step:

```
data _null_;
   x1 = -1/3;
   x2 = 22/7;
   x3 = -1234567891;
   x4 = 1234567890;
      /* Add the numbers in two different orders. */
   sum1 = x1 + x2 + x3 + x4;
   sum2 = x4 + x3 + x2 + x1;
```

```
   diff = abs (sum1 - sum2);
   put sum1= / sum2= / diff=;
      /* Using only a fuzz value gives the wrong result. The fuzz value  */
      /* is 8 because there are four numbers in each sum, for a total of */
      /* eight numbers.                                                  */
   compfuzz = compfuzz (sum1, sum2, 8);
   put "fuzz only (wrong):         " compfuzz=;
      /* Using a fuzz factor and a scale value gives the correct result. */
   scale = abs(x1) + abs(x2) + abs(x3) + abs(x4);
   compfuzz = compfuzz (sum1, sum2, 8, scale);
   put "fuzz and scale (correct):  " compfuzz=;
run;
```

SAS writes the following output to the log:

Log 2.8 *Partial SAS Log for the COMPFUZZ Function*

```
sum1=1.8095238095
sum2=1.8095238095
diff=5.543588E-11
fuzz only (wrong):         compfuzz=-1
fuzz and scale (correct):  compfuzz=0
```

See Also

Functions:

- "FUZZ Function" on page 500
- "ROUND Function" on page 833

COMPGED Function

Returns the generalized edit distance between two strings.

Category: Character

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

Syntax

COMPGED(*string-1*, *string-2* <,*cutoff*> <,*modifiers*>)

Required Arguments

string–1
specifies a character constant, variable, or expression.

string-2
specifies a character constant, variable, or expression.

Optional Arguments

cutoff
is a numeric constant, variable, or expression. If the actual generalized edit distance is greater than the value of *cutoff*, the value that is returned is equal to the value of *cutoff*.

TIP Using a small value of *cutoff* improves the efficiency of COMPGED if the values of *string–1* and *string–2* are long

modifiers
specifies a character string that can modify the action of the COMPGED function. You can use one or more of the following characters as a valid modifier:

i or I	ignores the case in *string–1* and *string–2*.
l or L	removes leading blanks in *string–1* and *string–2* before comparing the values.
n or N	removes quotation marks from any argument that is an n-literal and ignores the case of *string–1* and *string–2*.
: (colon)	truncates the longer of *string–1* or *string–2* to the length of the shorter string, or to one, whichever is greater.

TIP COMPGED ignores blanks that are used as modifiers.

Details

The Order in Which Modifiers Appear

The order in which the modifiers appear in the COMPGED function is relevant.

- "LN" first removes leading blanks from each string and then removes quotation marks from n-literals.
- "NL" first removes quotation marks from n-literals and then removes leading blanks from each string.

Definition of Generalized Edit Distance

Generalized edit distance is a generalization of Levenshtein edit distance, which is a measure of dissimilarity between two strings. The Levenshtein edit distance is the number of deletions, insertions, or replacements of single characters that are required to transform *string-1* into *string-2*.

Computing the Generalized Edit Distance

The COMPGED function returns the generalized edit distance between *string-1* and *string-2*. The generalized edit distance is the minimum-cost sequence of operations for constructing *string-1* from *string-2*.

The algorithm for computing the sum of the costs involves a pointer that points to a character in *string-2* (the input string). An output string is constructed by a sequence of operations that might advance the pointer, add one or more characters to the output string, or both. Initially, the pointer points to the first character in the input string, and the output string is empty.

The operations and their costs are described in the following table.

Operation	Default Cost in Units	Description of Operation
APPEND	50	When the output string is longer than the input string, add any one character to the end of the output string without moving the pointer.
BLANK	10	Do any of the following: • Add one space character to the end of the output string without moving the pointer. • When the character at the pointer is a space character, advance the pointer by one position without changing the output string. • When the character at the pointer is a space character, add one space character to the end of the output string, and advance the pointer by one position. If the cost for BLANK is set to zero by the COMPCOST function, the COMPGED function removes all space characters from both strings before doing the comparison.
DELETE	100	Advance the pointer by one position without changing the output string.
DOUBLE	20	Add the character at the pointer to the end of the output string without moving the pointer.
FDELETE	200	When the output string is empty, advance the pointer by one position without changing the output string.
FINSERT	200	When the pointer is in position one, add any one character to the end of the output string without moving the pointer.
FREPLACE	200	When the pointer is in position one and the output string is empty, add any one character to the end of the output string, and advance the pointer by one position.
INSERT	100	Add any one character to the end of the output string without moving the pointer.

Operation	Default Cost in Units	Description of Operation
MATCH	0	Copy the character at the pointer from the input string to the end of the output string, and advance the pointer by one position.
PUNCTUATION	30	Do any of the following: • Add one punctuation character to the end of the output string without moving the pointer. • When the character at the pointer is a punctuation character, advance the pointer by one position without changing the output string. • When the character at the pointer is a punctuation character, add one punctuation character to the end of the output string, and advance the pointer by one position. If the cost for PUNCTUATION is set to zero by the COMPCOST function, the COMPGED function removes all punctuation characters from both strings before doing the comparison.
REPLACE	100	Add any one character to the end of the output string, and advance the pointer by one position.
SINGLE	20	When the character at the pointer is the same as the character that follows in the input string, advance the pointer by one position without changing the output string.
SWAP	20	Copy the character that follows the pointer from the input string to the output string. Then copy the character at the pointer from the input string to the output string. Advance the pointer two positions.
TRUNCATE	10	When the output string is shorter than the input string, advance the pointer by one position without changing the output string.

To set the cost of the string operations, you can use the CALL COMPCOST routine or use default costs. If you use the default costs, the values that are returned by COMPGED are approximately 100 times greater than the values that are returned by COMPLEV.

Examples of Errors

The rationale for determining the generalized edit distance is based on the number and types of typographical errors that can occur. COMPGED assigns a cost to each error and determines the minimum sum of these costs that could be incurred. Some types of errors can be more serious than others. For example, inserting an extra letter at the beginning of a string might be more serious than omitting a letter from the end of a string. For another example, if you type a word or phrase that exists in *string-2* and introduce a typographical error, you might produce *string-1* instead of *string-2*.

Making the Generalized Edit Distance Symmetric

Generalized edit distance is not necessarily symmetric. That is, the value that is returned by **COMPGED(string1, string2)** is not always equal to the value that is returned by **COMPGED(string2, string1)**. To make the generalized edit distance symmetric, use the CALL COMPCOST routine to assign equal costs to the operations within each of the following pairs:

- INSERT, DELETE
- FINSERT, FDELETE
- APPEND, TRUNCATE
- DOUBLE, SINGLE

Comparisons

You can compute the Levenshtein edit distance by using the COMPLEV function. You can compute the generalized edit distance by using the CALL COMPCOST routine and the COMPGED function. Computing generalized edit distance requires considerably more computer time than does computing Levenshtein edit distance. But generalized edit distance usually provides a more useful measure than Levenshtein edit distance for applications such as fuzzy file merging and text mining.

Example

The following example uses the default costs to calculate the generalized edit distance.

```
data test;
   infile datalines missover;
   input String1 $char8. +1 String2 $char8. +1 Operation $40.;
   GED=compged(string1, string2);
   datalines;
baboon   baboon   match
baXboon  baboon   insert
baoon    baboon   delete
baXoon   baboon   replace
baboonX  baboon   append
baboo    baboon   truncate
babboon  baboon   double
babon    baboon   single
baobon   baboon   swap
bab oon  baboon   blank
bab,oon  baboon   punctuation
bXaoon   baboon   insert+delete
bXaYoon  baboon   insert+replace
bXoon    baboon   delete+replace
```

```
Xbaboon  baboon   finsert
aboon    baboon   trick question: swap+delete
Xaboon   baboon   freplace
axoon    baboon   fdelete+replace
axoo     baboon   fdelete+replace+truncate
axon     baboon   fdelete+replace+single
baby     baboon   replace+truncate*2
balloon  baboon   replace+insert
;

proc print data=test label;
   label GED='Generalized Edit Distance';
   var String1 String2 GED Operation;
run;
```

Display 2.25 *Generalized Edit Distance Based on Operation*

The SAS System

Obs	String1	String2	Generalized Edit Distance	Operation
1	baboon	baboon	0	match
2	baXboon	baboon	100	insert
3	baoon	baboon	100	delete
4	baXoon	baboon	100	replace
5	baboonX	baboon	50	append
6	baboo	baboon	10	truncate
7	babboon	baboon	20	double
8	babon	baboon	20	single
9	baobon	baboon	20	swap
10	bab oon	baboon	10	blank
11	bab,oon	baboon	30	punctuation
12	bXaoon	baboon	200	insert+delete
13	bXaYoon	baboon	200	insert+replace
14	bXoon	baboon	200	delete+replace
15	Xbaboon	baboon	200	finsert
16	aboon	baboon	120	trick question: swap+delete
17	Xaboon	baboon	200	freplace
18	axoon	baboon	300	fdelete+replace
19	axoo	baboon	310	fdelete+replace+truncate
20	axon	baboon	320	fdelete+replace+single
21	baby	baboon	120	replace+truncate*2
22	balloon	baboon	200	replace+insert

See Also

Functions:

- "COMPARE Function" on page 311
- "COMPLEV Function" on page 323

CALL Routines:

- "CALL COMPCOST Routine" on page 165

COMPLEV Function

Returns the Levenshtein edit distance between two strings.

Category: Character

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

Syntax

COMPLEV(*string-1*, *string-2* <,*cutoff*> <,*modifiers*>)

Required Arguments

string–1
specifies a character constant, variable, or expression.

string–2
specifies a character constant, variable, or expression.

Optional Arguments

cutoff
specifies a numeric constant, variable, or expression. If the actual Levenshtein edit distance is greater than the value of *cutoff*, the value that is returned is equal to the value of *cutoff*.

TIP Using a small value of *cutoff* improves the efficiency of COMPLEV if the values of *string–1* and *string–2* are long.

modifiers
specifies a character string that can modify the action of the COMPLEV function. You can use one or more of the following characters as a valid modifier:

i or I	ignores the case in *string–1* and *string–2*.
l or L	removes leading blanks in *string–1* and *string–2* before comparing the values.
n or N	removes quotation marks from any argument that is an n-literal and ignores the case of *string–1* and *string–2*.
: (colon)	truncates the longer of *string–1* or *string–2* to the length of the shorter string, or to one, whichever is greater.

TIP COMPLEV ignores blanks that are used as modifiers.

Details

The order in which the modifiers appear in the COMPLEV function is relevant.

- "LN" first removes leading blanks from each string and then removes quotation marks from n-literals.
- "NL" first removes quotation marks from n-literals and then removes leading blanks from each string.

The COMPLEV function ignores trailing blanks.

COMPLEV returns the Levenshtein edit distance between *string-1* and *string-2*. Levenshtein edit distance is the number of insertions, deletions, or replacements of single characters that are required to convert one string to the other. Levenshtein edit distance is symmetric. That is, **COMPLEV(string-1,string-2)** is the same as **COMPLEV(string-2,string-1)**.

Comparisons

The Levenshtein edit distance that is computed by COMPLEV is a special case of the generalized edit distance that is computed by COMPGED.

COMPLEV executes much more quickly than COMPGED.

Example

The following example compares two strings by computing the Levenshtein edit distance.

```
data test;
   infile datalines missover;
   input string1 $char8. string2 $char8. modifiers $char8.;
   result=complev(string1, string2, modifiers);
   datalines;
1234567812345678
abc     abxc
ac      abc
aXc     abc
aXbZc   abc
aXYZc   abc
WaXbYcZ abc
XYZ     abcdef
aBc     abc
aBc     AbC      i
  abc   abc
  abc   abc      l
AxC     'abc'n
AxC     'abc'n   n
;

proc print data=test;
run;
```

Display 2.26 *Results of Comparing Two Strings by Computing the Levenshtein Edit Distance*

The SAS System

Obs	string1	string2	modifiers	result
1	12345678	12345678		0
2	abc	abxc		1
3	ac	abc		1
4	aXc	abc		1
5	aXbZc	abc		2
6	aXYZc	abc		3
7	WaXbYcZ	abc		4
8	XYZ	abcdef		6
9	aBc	abc		1
10	aBc	AbC	i	0
11	abc	abc		2
12	abc	abc	l	0
13	AxC	'abc'n		6
14	AxC	'abc'n	n	1

See Also

Functions:

- “COMPARE Function” on page 311
- “COMPGED Function” on page 317

CALL Routines:

- “CALL COMPCOST Routine” on page 165

COMPOUND Function

Returns compound interest parameters.

Category: Financial

Syntax

COMPOUND(*a*,*f*,*r*,*n*)

Required Arguments

a

is numeric, and specifies the initial amount.

Range $a \geq 0$

f

is numeric, and specifies the future amount (at the end of *n* periods).

Range $f \geq 0$

r

is numeric, and specifies the periodic interest rate expressed as a fraction.

Range $r \geq 0$

n

is an integer, and specifies the number of compounding periods.

Range $n \geq 0$

Details

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

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

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

If n=0, then

$$f = a$$

and

$$(1 + r)^n$$

are equal to 1.

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

Example

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

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

The value returned is 2502.54. The second argument has been set to missing, indicating that the future amount is to be calculated. The 9 percent 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.

Category: Character

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

Tip: DBCS equivalent function is KCOMPRESS .

Syntax

COMPRESS(<*source*> <, *chars*> <, *modifiers*>)

Optional Arguments

source
specifies a character constant, variable, or expression from which specified characters will be removed.

chars
specifies a character constant, variable, or expression that initializes a list of characters.

By default, the characters in this list are removed from the *source* argument. If you specify the K modifier in the third argument, then only the characters in this list are kept in the result.

TIP You can add more characters to this list by using other modifiers in the third argument.

TIP Enclose a literal string of characters in quotation marks.

modifier
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COMPRESS function. Blanks are ignored. The following characters can be used as modifiers:

a or A	adds alphabetic characters to the list of characters.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds the underscore character and English letters to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters to be kept or removed.
k or K	keeps the characters in the list instead of removing them.
l or L	adds lowercase letters to the list of characters.

n or N	adds digits, the underscore character, and English letters to the list of characters.
o or O	processes the second and third arguments once rather than every time the COMPRESS function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make COMPRESS run much faster when you call it in a loop where the second and third arguments do not change.
p or P	adds punctuation marks to the list of characters.
s or S	adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.
t or T	trims trailing blanks from the first and second arguments.
u or U	adds uppercase letters to the list of characters.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

TIP If the *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

Details

Length of Returned Variable

In a DATA step, if the COMPRESS function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The Basics

The COMPRESS function allows null arguments. A null argument is treated as a string that has a length of zero.

Based on the number of arguments, the COMPRESS functions works as follows:

Number of Arguments	Result
only the first argument, *source*	The argument has all blanks 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.
the first two arguments, *source* and *chars*	All characters that appear in the second argument are removed from the result.
three arguments, *source*, *chars*, and *modifier(s)*	The K modifier (specified in the third argument) determines whether the characters in the second argument are kept or removed from the result.

The COMPRESS function compiles a list of characters to keep or remove, comprising the characters in the second argument plus any types of characters that are specified by

the modifiers. For example, the D modifier specifies digits. Both of the following function calls remove digits from the result:

```
COMPRESS(source, "1234567890");
COMPRESS(source, , "d");
```

To remove digits and plus or minus signs, you can use either of the following function calls:

```
COMPRESS(source, "1234567890+-");
COMPRESS(source, "+-", "d");
```

Examples

Example 1: Compressing Blanks

SAS Statement	Result
	`----+----1`

`a='AB C D ';`
`b=compress(a);`
`put b;`
	`ABCD`

Example 2: Compressing Lowercase Letters

SAS Statement	Result
	`----+----1----+----2----+----3`

`x='123-4567-8901 B 234-5678-9012 c';`
`y=compress(x,'ABCD','l');`
`put y;`
	`123-4567-8901 234-5678-9012`

Example 3: Compressing Space Characters

SAS Statement	Result
	`----+----1`

`x='1  2  3  4  5';`
`y=compress(x,,'s');`
`put y;`
	`12345`

Example 4: Keeping Characters in the List

SAS Statement	Result
	`----+----1`

SAS Statement	Result
x='Math A English B Physics A'; y=compress(x,'ABCD','k'); put y;	ABA

Example 5: Compressing a String and Returning a Length of 0

SAS Statement	Result
	----+----1
x=' '; l=lengthn(compress(x)); put l;	0

See Also

Functions:

- “COMPBL Function” on page 314
- “LEFT Function” on page 616
- “TRIM Function” on page 924

CONSTANT Function

Computes machine and mathematical constants.

Category: Mathematical

Syntax

CONSTANT(*constant*<, *parameter*>)

Required Argument

constant
is a character constant, variable, or expression that identifies the constant to be returned. Valid constants are as follows:

Description	Constant
The natural base	'E'
Euler constant	'EULER'
Pi	'PI'

Description	Constant
Exact integer	'EXACTINT' <,nbytes>
The largest double-precision number	'BIG'
The log with respect to *base* of BIG	'LOGBIG' <,base>
The square root of BIG	'SQRTBIG'
The smallest double-precision number	'SMALL'
The log with respect to *base* of SMALL	'LOGSMALL' <,base>
The square root of SMALL	'SQRTSMALL'
Machine precision constant	'MACEPS'
The log with respect to *base* of MACEPS	'LOGMACEPS' <,base>
The square root of MACEPS	'SQRTMACEPS'

Optional Argument

parameter
is an optional numeric parameter. Some of the constants specified in *constant* have an optional argument that alters the functionality of the CONSTANT function.

Details

Overview

CAUTION:
In some operating environments, the run-time library might have limitations that prevent the use of the full range of floating-point numbers that the hardware provides. In such cases, the CONSTANT function attempts to return values that are compatible with the limitations of the run-time library. For example, if the run-time library cannot compute **EXP(LOG(CONSTANT('BIG')))**, then **CONSTANT('LOGBIG')** 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.

Restriction The *base* that you specify must be greater than the value of 1+SQRTMACEPS.

Default the natural base, E.

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.

Restriction	The *base* that you specify must be greater than the value of 1+SQRTMACEPS.
Default	the natural base, E.

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.

Restriction	The *base* that you specify must be greater than the value of 1+SQRTMACEPS.
Default	the natural base, E.

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').

CONVX Function

Returns the convexity for an enumerated cash flow.

Category: Financial

Syntax

CONVX(*y*,*f*,*c(1), ... ,c(k)*)

Required Arguments

y
specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range $0 < y < 1$

f
specifies the frequency of cash flows per period.

Range $f > 0$

c(1), ... ,c(k)
specifies a list of cash flows.

Details

The CONVX function returns the value

$$C = \sum_{k=1}^{K} \frac{k(k+f)\dfrac{c(k)}{(1+y)^{\frac{k}{f}}}}{P\left((1+y)^2\right) f^2}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example

```
data _null_;
   c=convx(1/20,1,.33,.44,.55,.49,.50,.22,.4,.8,.01,.36,.2,.4);
   put c;
run;
```

The value that is returned is 42.3778.

CONVXP Function

Returns the convexity for a periodic cash flow stream, such as a bond.

Category: Financial

Syntax

CONVXP(*A*,*c*,*n*,*K*,k_0,*y*)

Required Arguments

A

specifies the par value.

Range $A > 0$

c

specifies the nominal per-period coupon rate, expressed as a fraction.

Range $0 \le c < 1$

n

specifies the number of coupons per period.

Range $n > 0$ and is an integer

K

specifies the number of remaining coupons.

Range $K > 0$ and is an integer

k_0

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

Range $0 < k_0 \le \frac{1}{n}$

y

specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Details

The CONVXP function returns the value

$$C = \frac{1}{n^2}\left(\frac{\sum_{k=1}^{K} t_k(t_k+1)\frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}}{P\left(1+\frac{y}{n}\right)^2}\right)$$

The following relationships apply to the preceding equation:

$$t_k = nk_0 + k - 1$$

$$c(k) = \frac{c}{n}A \quad for\ k = 1, \ \ldots, K-1$$

$$c(K) = \left(1+\frac{c}{n}\right)A$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1+\frac{y}{n}\right)^{t_k}}$$

Example

In the following example, the CONVXP function returns the convexity of a bond that has a face value of 1000, an annual coupon rate of 0.01, 4 coupons per year, and 14 remaining coupons. The time from settlement date to next coupon date is 0.165, and the annual yield-to-maturity is 0.08.

```
data _null_;
   y=convxp(1000,.01,4,14,.33/2,.08);
   put y;
run;
```

The value that is returned is 11.729001987.

COS Function

Returns the cosine.

Category: Trigonometric

Syntax

COS (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression and is expressed in radians. If the magnitude of *argument* is so great that **mod(argument,pi)** is accurate to less than about three decimal places, COS returns a missing value.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=cos(0.5);`	`0.8775825619`
`x=cos(0);`	`1`
`x=cos(3.14159/3);`	`0.500000766`

COSH Function

Returns the hyperbolic cosine.

Category: Hyperbolic

Syntax

COSH(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The COSH function returns the hyperbolic cosine of the argument, given by

$$\left(\varepsilon^{argument} + \varepsilon^{-argument}\right) / 2$$

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=cosh(0);`	`1`
`x=cosh(-5.0);`	`74.209948525`

SAS Statement	Result
`x=cosh(0.5);`	`1.1276259652`

COUNT Function

Counts the number of times that a specified substring appears within a character string.

Category: Character

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

Tip: You can use the KCOUNT function for DBCS processing, but the functionality is different. See DBCS Compatibility on page 339 .

Syntax

COUNT(*string*,*substring* <,*modifiers*>)

Required Arguments

string
specifies a character constant, variable, or expression in which substrings are to be counted.

Tip Enclose a literal string of characters in quotation marks.

substring
is a character constant, variable, or expression that specifies the substring of characters to count in *string*.

Tip Enclose a literal string of characters in quotation marks.

Optional Argument

modifiers
is a character constant, variable, or expression that specifies one or more modifiers. The following *modifiers* can be in uppercase or lowercase:

i ignores character case during the count. If this modifier is not specified, COUNT only counts character substrings with the same case as the characters in *substring*.

t trims trailing blanks from *string* and *substring*.

Tip If the *modifier* is a constant, enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

Details

The Basics

The COUNT function searches *string*, from left to right, for the number of occurrences of the specified *substring*, and returns that number of occurrences. If the substring is not found in *string*, COUNT returns a value of 0.

CAUTION:

If two occurrences of the specified substring overlap in the string, the result is undefined. For example, COUNT('boobooboo', 'booboo') might return either a 1 or a 2.

DBCS Compatibility

You can use the KCOUNT function, which is documented in *SAS National Language Support (NLS): Reference Guide*, for DBCS processing, but the functionality is different.

If the value of *substring* in the COUNT function is longer than two bytes, then the COUNT function can handle DBCS strings. The following examples show the differences in syntax:

COUNT(*string*, *substring <,modifiers>*

KCOUNT(*string)*

Comparisons

The COUNT function counts substrings of characters in a character string, whereas the COUNTC function counts individual characters in a character string.

Example

The following SAS statements produce these results:

SAS Statement	Result

`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`

SAS Statement	Result

`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

See Also

Functions:

- “COUNTC Function” on page 340
- “COUNTW Function” on page 343
- “FIND Function” on page 457
- “INDEX Function” on page 543

COUNTC Function

Counts the number of characters in a string that appear or do not appear in a list of characters.

Category: Character

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

Syntax

COUNTC(*string*, *charlist* <,*modifiers*>)

Required Arguments

string
specifies a character constant, variable, or expression in which characters are counted.

Tip Enclose a literal string of characters in quotation marks.

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 *modifier* 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.

Tips Enclose a literal string of characters in quotation marks.

If there are no characters in the list after processing the modifiers, COUNTC returns 0.

Optional Argument

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTC function. Blanks are ignored. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
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.

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.

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.

```
data test;
   string  = 'Baboons Eat Bananas     ';
   a       = countc(string, 'a');
   b       = countc(string,'b');
   b_i     = countc(string,'b','i');
   abc_i   = countc(string,'abc','i');
      /* Scan string for characters that are not "a", "b", */
      /* and "c", ignore case, (and include blanks).       */
   abc_iv  = countc(string,'abc','iv');
      /* Scan string for characters that are not "a", "b", */
      /* and "c", ignore case, and trim trailing blanks.   */
   abc_ivt = countc(string,'abc','ivt');
run;

options pageno=1 ls=80 nodate;
proc print data=test noobs;
run;
```

Display 2.27 *Output from Using the COUNTC Functions with and without Modifiers*

The SAS System

string	a	b	b_i	abc_i	abc_iv	abc_ivt
Baboons Eat Bananas	5	1	3	8	16	11

See Also

Functions:

- “ANYALNUM Function” on page 99
- “ANYALPHA Function” on page 101
- “ANYCNTRL Function” on page 103
- “ANYDIGIT Function” on page 105
- “ANYGRAPH Function” on page 108

- "ANYLOWER Function" on page 110
- "ANYPRINT Function" on page 113
- "ANYPUNCT Function" on page 116
- "ANYSPACE Function" on page 118
- "ANYUPPER Function" on page 120
- "ANYXDIGIT Function" on page 121
- "NOTALNUM Function" on page 685
- "NOTALPHA Function" on page 687
- "NOTCNTRL Function" on page 689
- "NOTDIGIT Function" on page 690
- "NOTGRAPH Function" on page 695
- "NOTLOWER Function" on page 697
- "NOTPRINT Function" on page 701
- "NOTPUNCT Function" on page 702
- "NOTSPACE Function" on page 704
- "NOTUPPER Function" on page 707
- "NOTXDIGIT Function" on page 708
- "FINDC Function" on page 459
- "INDEXC Function" on page 545
- "VERIFY Function" on page 944

COUNTW Function

Counts the number of words in a character string.

Category: Character

Syntax

COUNTW(<*string*> <, *chars*> <, *modifiers*>)

Optional Arguments

string
specifies a character constant, variable, or expression in which words are counted.

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.

modifier

specifies a character constant, variable, or expression in which each non-blank character modifies the action of the COUNTW function. The following characters, in uppercase or lowercase, can be used as modifiers:

blank	is ignored.
a or A	adds alphabetic characters to the list of characters.
b or B	counts from right to left instead of from left to right. Right-to-left counting makes a difference only when you use the Q modifier and the string contains unbalanced quotation marks.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.
f or F	adds an underscore and English letters (that is, the characters that can begin a SAS variable name using VALIDVARNAME=V7) to the list of characters.
g or G	adds graphic characters to the list of characters.
h or H	adds a horizontal tab to the list of characters.
i or I	ignores the case of the characters.
k or K	causes all characters that are not in the list of characters to be treated as delimiters. If K is not specified, 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 of substrings that are enclosed in quotation marks. If the value of *string* contains unmatched quotation marks, then scanning from left to right will produce different words than scanning from right to left.
s or S	adds space characters (blank, horizontal tab, vertical tab, carriage return, line feed, and form feed) to the list of characters.
t or T	trims trailing blanks from the *string* and *chars* arguments.
u or U	adds uppercase letters to the list of characters.
w or W	adds printable characters to the list of characters.
x or X	adds hexadecimal characters to the list of characters.

Details

Definition of "Word"

In the COUNTW function, "word" refers to a substring that has one of the following characteristics:

- is bounded on the left by a delimiter or the beginning of the string
- is bounded on the right by a delimiter or the end of the string
- contains no delimiters, except if you use the Q modifier and the delimiters are within substrings that have quotation marks

Note: The definition of "word" is the same in both the SCAN function and the COUNTW.sgml 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 of 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.

```
data test;
   length default blanks mp 8;
   input string $char60.;
   default = countw(string);
   blanks = countw(string, ' ');
   mp = countw(string, 'mp');
   datalines;
The quick brown fox jumps over the lazy dog.
        Leading blanks
```

```
2+2=4
/unix/path/names/use/slashes
\Windows\Path\Names\Use\Backslashes
;
run;

proc print noobs data=test;
run;
```

Display 2.28 *Output from the COUNTW Function*

The SAS System

default	blanks	mp	string
9	9	2	The quick brown fox jumps over the lazy dog.
2	2	1	Leading blanks
2	1	1	2+2=4
5	1	3	/unix/path/names/use/slashes
1	1	2	\Windows\Path\Names\Use\Backslashes

See Also

Functions:

- “COUNT Function” on page 338
- “COUNTC Function” on page 340
- “FINDW Function” on page 467
- “SCAN Function” on page 848

CALL Routines:

- “CALL SCAN Routine” on page 237

CSS Function

Returns the corrected sum of squares.

Category: Descriptive Statistics

Syntax

CSS(*argument-1*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one nonmissing argument is required. Otherwise, the function returns a missing value. If you have more than one argument, the argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=css(5,9,3,6);`	`18.75`
`x2=css(5,8,9,6,.);`	`10`
`x3=css(8,9,6,.);`	`4.6666666667`
`x4=css(of x1-x3);`	`101.11574074`

CUMIPMT Function

Returns the cumulative interest paid on a loan between the start and end period.

Category: Financial

Syntax

CUMIPMT (*rate*, *number-of-periods*, *principal-amount*, <*start-period*>, <*end-period*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

number-of-periods
specifies the number of payment periods. *number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

start-period
specifies the start period for the calculation.

end-period
specifies the end period for the calculation.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

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:

```
TotalInterest = CUMIPMT(0.09/12, 360, 125000, 13, 24, 0);
```

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:

```
first_period_interest = CUMIPMT(0.09/12, 360, 125000, 1, 1, 0);
```

This computation returns a value of 937.50.

CUMPRINC Function

Returns the cumulative principal paid on a loan between the start and end period.

Category: Financial

Syntax

CUMPRINC (*rate*, *number-of-periods*, *principal-amount*, <*start-period*>, <*end-period*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

number-of-periods
specifies the number of payment periods. *number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

start-period
specifies the start period for the calculation.

end-period
specifies the end period for the calculation.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

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:

```
PrincipalYear2=CUMPRINC(0.09/12, 360, 125000, 12, 24, 0);
```

  This computation returns a value of 934.107.

- The principal that is paid on the second year of the same loan with beginning-of-period payments is computed as follows:

```
PrincipalYear2b = CUMPRINC(0.09/12, 360, 125000, 12, 24, 1);
```

  This computation returns a value of 927.153.

CUROBS Function

Returns the observation number of the current observation.

Category: SAS File I/O

Requirement: Use this function only with an uncompressed SAS data set that is accessed using a native library engine.

Syntax

CUROBS(*data-set-id*)

Required Argument

data-set-id
is a numeric value that specifies the data set identifier that the OPEN function returns.

Details

If the engine being used does not support observation numbers, the function returns a missing value.

With a SAS view, the function returns the relative observation number, that is, the number of the observation within the SAS view (as opposed to the number of the observation within any related SAS data set).

Example

This example uses the FETCHOBS function to fetch the tenth observation in the data set MYDATA. The value of OBSNUM returned by CUROBS is 10.

```
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetchobs(&dsid,10));
%let obsnum=%sysfunc(curobs(&dsid));
```

See Also

Functions:

- "FETCHOBS Function" on page 406
- "OPEN Function" on page 716

CV Function

Returns the coefficient of variation.

Category: Descriptive Statistics

Syntax

CV(*argument-1*,*argument-2*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two arguments are required. The argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=cv(5,9,3,6);`	`43.47826087`
`x2=cv(5,8,9,6,.);`	`26.082026548`
`x3=cv(8,9,6,.);`	`19.924242152`
`x4=cv(of x1-x3);`	`40.953539216`

DACCDB Function

Returns the accumulated declining balance depreciation.

Category: Financial

Syntax

DACCDB(*p*,*v*,*y*,*r*)

Required Arguments

p

is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v

is numeric, the depreciable initial value of the asset.

y

is numeric, the lifetime of the asset.

Range $y > 0$

r

is numeric, the rate of depreciation expressed as a decimal.

Range $r > 0$

Details

The DACCDB function returns the accumulated depreciation by using a declining balance method. The formula is

$$DACCDB(p,\ v,\ y,\ r) = \begin{cases} 0 & p \le 0 \\ v\left(1 - \left(1 - \frac{r}{y}\right)^{int(p)}\right)\left(1 - (p - int(p))\frac{r}{y}\right) & p > 0 \end{cases}$$

Note that int(*p*) is the integer part of *p*. The *p* and *y* arguments must be expressed by using the same units of time. A double-declining balance is obtained by setting *r* equal to 2.

Example

An asset has a depreciable initial value of $1000 and a fifteen-year lifetime. Using a 200 percent declining balance, the depreciation throughout the first 10 years can be expressed as

```
a=daccdb(10,1000,15,2);
```

The value returned is 760.93. The first and the third arguments are expressed in years.

DACCDBSL Function

Returns the accumulated declining balance with conversion to a straight-line depreciation.

Category: Financial

Syntax

DACCDBSL(*p*,*v*,*y*,*r*)

Required Arguments

p

is numeric, the period for which the calculation is to be done.

v
is numeric, the depreciable initial value of the asset.

y
is an integer, the lifetime of the asset.

Range $y > 0$

r
is numeric, the rate of depreciation that is expressed as a fraction.

Range $r > 0$

Details

The DACCDBSL function returns the accumulated depreciation by using a declining balance method, with conversion to a straight-line depreciation function that is defined by

$$DACCDBSL\ (p,\ v,\ y,\ r) = \sum_{i=1}^{p} DEPDBSL\ (i,\ v,\ y,\ r)$$

The declining balance with conversion to a straight-line depreciation chooses for each time period the method of depreciation (declining balance or straight-line on the remaining balance) that gives the larger depreciation. The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset has a depreciable initial value of $1,000 and a ten-year lifetime. Using a declining balance rate of 150%, the accumulated depreciation of that asset in its fifth year can be expressed as

```
y5=daccdbsl(5,1000,10,1.5);
```

The value returned is 564.99. The first and the third arguments are expressed in years.

DACCSL Function

Returns the accumulated straight-line depreciation.

Category: Financial

Syntax

DACCSL(*p*,*v*,*y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For fractional *p*, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

Details

The DACCSL function returns the accumulated depreciation by using the straight-line method, which is given by

$$DACCSL\ (p,\ v,\ y) = \begin{cases} 0 & p < 0 \\ v\left(\frac{p}{y}\right) & 0 \le p \le y \\ v & p > y \end{cases}$$

The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01APR86, has a depreciable initial value of $1000 and a ten-year lifetime. The accumulated depreciation in the value of the asset through 31DEC87 can be expressed as

```
a=daccsl(1.75,1000,10);
```

The value returned is 175.00. The first and the third arguments are expressed in years.

DACCSYD Function

Returns the accumulated sum-of-years-digits depreciation.

Category: Financial

Syntax

DACCSYD(*p*,*v*,*y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

Details

The DACCSYD function returns the accumulated depreciation by using the sum-of-years-digits method. The formula is

$$DACCSYD(p,\ v,\ y) = \begin{cases} 0 & p < 0 \\ v\dfrac{int(p)\left(y - \dfrac{int(p) - 1}{2}\right) + (p - int(p))(y - int(p))}{int(y)\left(y - \dfrac{int(y) - 1}{2}\right) + (y - int(y))^2} & 0 \le p \le y \\ v & p > y \end{cases}$$

Note that int(*y*) indicates the integer part of *y*. The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01OCT86, has a depreciable initial value of $1,000 and a five-year lifetime. The accumulated depreciation of the asset throughout 01JAN88 can be expressed as

```
y2=daccsyd(15/12,1000,5);
```

The value returned is 400.00. The first and the third arguments are expressed in years.

DACCTAB Function

Returns the accumulated depreciation from specified tables.

Category: Financial

Syntax

DACCTAB(*p*,*v*,*t1, . . . ,tn*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

t1,t2, . . . ,tn
are numeric, the fractions of depreciation for each time period with *t1+t2+...tn* ≤ 1.

Details

The DACCTAB function returns the accumulated depreciation by using user-specified tables. The formula for this function is

$$DACCTAB(p,\ v,\ t_1,\ t_2,\ ...,\ t_n) = \begin{cases} 0 & p \le 0 \\ v\left(t_1 + t_2 + ... + t_{int(p)} + (p - int(p))t_{int(p)+1}\right) & 0 < p < n \\ v & p \ge n \end{cases}$$

For a given *p*, only the arguments $t_1,\ t_2,\ \ldots,\ t_k$ need to be specified with **k=ceil(p)**.

Example

An asset has a depreciable initial value of $1000 and a five-year lifetime. Using a table of the annual depreciation rates of .15, .22, .21, .21, and .21 during the first, second, third, fourth, and fifth years, respectively, the accumulated depreciation throughout the third year can be expressed as

```
y3=dacctab(3,1000,.15,.22,.21,.21,.21);
```

The value that is returned is 580.00. The fourth rate, .21, and the fifth rate, .21, can be omitted because they are not needed in the calculation.

DAIRY Function

Returns the derivative of the AIRY function.

Category: Mathematical

Syntax

DAIRY(*x*)

Required Argument

x
specifies a numeric constant, variable, or expression.

Details

The DAIRY function returns the value of the derivative of the AIRY function. (See a list of References on page 1001.)

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=dairy(2.0);`	`-0.053090384`
`x=dairy(-2.0);`	`0.6182590207`

DATDIF Function

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

Category: Date and Time

Syntax

DATDIF(*sdate*,*edate*,*basis*)

Required Arguments

sdate
specifies a SAS date value that identifies the starting 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.

Tip If *edate* falls at the end of a month, then SAS treats the date as if it were the last day of a 30-day month.

basis
specifies a character string that represents the day count basis. The following values for *basis* are valid:

'30/360'
specifies a 30-day month and a 360-day year, regardless of the actual number of calendar days in a month or year.

A security that pays interest on the last day of a month will either always make its interest payments on the last day of the month, or it will always make its payments on the numerically same day of a month, unless that day is not a valid day of the month, such as February 30. For more information, see "Method of Calculation for Day Count Basis (30/360)" on page 357 .

Alias '360'

'ACT/ACT'
uses the actual number of days between dates. Each month is considered to have the actual number of calendar days in that month, and each year is considered to have the actual number of calendar days in that year.

Alias *'Actual'*

'ACT/360'
uses the actual number of calendar days in a particular month, and 360 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/360* is used for short-term securities.

'ACT/365'
uses the actual number of calendar days in a particular month, and 365 days as the number of days in a year, regardless of the actual number of days in a year.

Tip *ACT/365* is used for short-term securities.

Details

The Basics

The DATDIF function has a specific meaning in the securities industry, and the method of calculation is not the same as the actual day count method. Calculations can use months and years that contain the actual number of days. Calculations can also be based on a 30-day month or a 360-day year. For more information about standard securities calculation methods, see the References section at the bottom of this function.

Note: When counting the number of days in a month, DATDIF *always* includes the starting date and excludes the ending date.

Method of Calculation for Day Count Basis (30/360)

To calculate the number of days between two dates, use the following formula:

$$Numberofdays = [(Y2 - Y1) * 360] + [(M2 - M1) * 30] + (D2 - D1)$$

Arguments

Y2
specifies the year of the later date.

Y1
specifies the year of the earlier date.

M2
specifies the month of the later date.

M1
specifies the month of the earlier date.

D2
specifies the day of the later date.

D1
specifies the day of the earlier date.

Because all months can contain only 30 days, you must adjust for the months that do not contain 30 days. Do this before you calculate the number of days between the two dates.

The following rules apply:

- If the security follows the End-of-Month rule, and D2 is the last day of February (28 days in a non-leap year, 29 days in a leap year), and D1 is the last day of February, then change D2 to 30.
- If the security follows the End-of-Month rule, and D1 is the last day of February, then change D1 to 30.
- If the value of D2 is 31 and the value of D1 is 30 or 31, then change D2 to 30.
- If the value of D1 is 31, then change D1 to 30.

Example

In the following example, DATDIF returns the actual number of days between two dates, as well as the number of days based on a 30-day month and a 360-day year.

```
data _null;
   sdate='16oct78'd;
   edate='16feb96'd;
   actual=datdif(sdate, edate, 'act/act');
```

```
    days360=datdif(sdate, edate, '30/360');
    put actual= days360=;
run;
```

SAS Statement	Result
put actual=;	6332
put days360=;	6240

See Also

Functions:

- "YRDIF Function" on page 986

References

Securities Industry Association 1994. *Standard Securities Calculation Methods - Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. New York, USA: . Securities Industry Association.

DATE Function

Returns the current date as a SAS date value.

Category: Date and Time

Alias: TODAY

See: "TODAY Function" on page 917

Syntax

DATE()

DATEJUL Function

Converts a Julian date to a SAS date value.

Category: Date and Time

Syntax

DATEJUL(*julian-date*)

Required Argument

julian-date
specifies a SAS numeric expression that represents a Julian date. A Julian date in SAS is a date in the form *yyddd* or *yyyyddd*, where *yy* or *yyyy* is a two-digit or four-

digit integer that represents the year and *ddd* is the number of the day of the year. The value of *ddd* must be between 1 and 365 (or 366 for a leap year).

Example

The following SAS statements produce these results:

SAS Statement	Result
`Xstart=datejul(94365);` `put Xstart / Xstart date9.;`	`12783` `31DEC1994`
`Xend=datejul(2001001);` `put Xend / Xend date9.;`	`14976` `01JAN2001`

See Also

Functions:

- "JULDATE Function" on page 602

DATEPART Function

Extracts the date from a SAS datetime value.

Category: Date and Time

Syntax

DATEPART(*datetime*)

Required Argument

datetime
specifies a SAS expression that represents a SAS datetime value.

Example

The following SAS statements produce this result:

SAS Statement	Result

`conn='01feb94:8:45'dt;`
`servdate=datepart(conn);`
`put servdate worddate.;`
	`February 1, 1994`

See Also

Functions:

- "DATETIME Function" on page 360
- "TIMEPART Function" on page 912

DATETIME Function

Returns the current date and time of day as a SAS datetime value.

Category: Date and Time

Syntax

DATETIME()

Example

This example returns a SAS value that represents the number of seconds between January 1, 1960 and the current time:

```
when=datetime();
put when=;
```

See Also

Functions:

- "DATE Function" on page 358
- "TIME Function" on page 912

DAY Function

Returns the day of the month from a SAS date value.

Category: Date and Time

Syntax

DAY(*date*)

Required Argument

date
specifies a SAS expression that represents a SAS date value.

Details

The DAY function produces an integer from 1 to 31 that represents the day of the month.

Example

The following SAS statements produce this result:

SAS Statement	Result

`now='05may97'd;`
`d=day(now);`
`put d;`
	`5`

See Also

Functions:

- "MONTH Function" on page 669
- "YEAR Function" on page 984

DCLOSE Function

Closes a directory that was opened by the DOPEN function.

Category: External Files

Syntax

DCLOSE(*directory-id*)

Required Argument

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

Details

DCLOSE returns 0 if the operation was successful, ≠0 if it was not successful. The DCLOSE function closes a directory that was previously opened by the DOPEN function. DCLOSE also closes any open members.

Note: All directories or members opened within a DATA step are closed automatically when the DATA step ends.

Examples

Example 1: Using DCLOSE to Close a Directory

This example opens the directory to which the fileref MYDIR has previously been assigned, returns the number of members, and then closes the directory:

```
%macro memnum(filrf,path);
%let rc=%sysfunc(filename(filrf,&path));
%if %sysfunc(fileref(&filrf)) = 0 %then
   %do;
```

```
         /* Open the directory. */
      %let did=%sysfunc(dopen(&filrf));
      %put did=&did;
         /* Get the member count. */
      %let memcount=%sysfunc(dnum(&did));
      %put &memcount members in &filrf.;
         /* Close the directory. */
      %let rc= %sysfunc(dclose(&did));
   %end;
%else %put Invalid FILEREF;
%mend;
%memnum(MYDIR,physical-filename)
```

Example 2: Using DCLOSE within a DATA Step

This example uses the DCLOSE function within a DATA step:

```
%let filrf=MYDIR;
data _null_;
  rc=filename("&filrf","physical-filename");
  if fileref("&filrf") = 0 then
     do;
           /* Open the directory. */
        did=dopen("&filrf");
           /* Get the member count. */
        memcount=dnum(did);
        put memcount "members in &filrf";
           /* Close the directory. */
        rc=dclose(did);
     end;
  else put "Invalid FILEREF";
run;
```

See Also

Functions:

- “DOPEN Function” on page 382
- “FCLOSE Function” on page 402
- “FOPEN Function” on page 485
- “MOPEN Function” on page 670

DCREATE Function

Returns the complete pathname of a new, external directory.

Category: External Files

Syntax

DCREATE(*directory-name*<,*parent-directory*>)

Required Argument

directory-name
is a character constant, variable, or expression that specifies the name of the directory to create. This value cannot include a pathname.

Optional Argument

parent-directory
is a character constant, variable, or expression that contains the complete pathname of the directory in which to create the new directory. If you do not supply a value for *parent-directory*, then the current directory is the parent directory.

Details

The DCREATE function enables you to create a directory in your operating environment. If the directory cannot be created, then DCREATE returns an empty string.

Example

To create a new directory in the UNIX operating environment, using the name that is stored in the variable DirectoryName, follow this form:

```
NewDirectory=dcreate(DirectoryName,'/local/u/abcdef/');
```

To create a new directory in the Windows operating environment, using the name that is stored in the variable DirectoryName, follow this form:

```
NewDirectory=dcreate(DirectoryName,'d:\testdir\');
```

DEPDB Function

Returns the declining balance depreciation.

Category: Financial

Syntax

DEPDB(*p*,*v*,*y*,*r*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

r
is numeric, the rate of depreciation that is expressed as a fraction.

Range $r \geq 0$

Details

The DEPDB function returns the depreciation by using the declining balance method, which is given by

$$DEPDB(p, v, y, r) = DACCDB(p, v, y, r) - DACCDB(p - 1, v, y, r)$$

The *p* and *y* arguments must be expressed by using the same units of time. A double-declining balance is obtained by setting *r* equal to 2.

Example

An asset has an initial value of $1,000 and a fifteen-year lifetime. Using a declining balance rate of 200%, the depreciation of the value of the asset for the tenth year can be expressed as

```
y10=depdb(10,1000,15,2);
```

The value returned is 36.78. The first and the third arguments are expressed in years.

DEPDBSL Function

Returns the declining balance with conversion to a straight-line depreciation.

Category: Financial

Syntax

DEPDBSL(*p*,*v*,*y*,*r*)

Required Arguments

p
is an integer, the period for which the calculation is to be done.

v
is numeric, the depreciable initial value of the asset.

y
is an integer, the lifetime of the asset.

Range $y > 0$

r
is numeric, the rate of depreciation that is expressed as a fraction.

Range $r \geq 0$

Details

The DEPDBSL function returns the depreciation by using the declining balance method with conversion to a straight-line depreciation, which is given by the following equation:

$$DEPDBSL\ (p,\ v,\ y,\ r) = \begin{cases} 0 & p \le 0 \\ v\frac{r}{y}\left(1 - \frac{r}{y}\right)^{p-1} & 0 < p \le t \\ \frac{v\left(1 - \frac{r}{y}\right)^{t}}{(y - t)} & t < p \le y \\ 0 & p > y \end{cases}$$

The following relationship applies to the preceding equation:

$$t = int\left(y - \frac{y}{r} + 1\right)$$

and *int*( ) denotes the integer part of a numeric argument.

The *p* and *y* arguments must be expressed by using the same units of time. The declining balance that changes to a straight-line depreciation chooses for each time period the method of depreciation (declining balance or straight-line on the remaining balance) that gives the larger depreciation.

Example

An asset has a depreciable initial value of $1,000 and a ten-year lifetime. Using a declining balance rate of 150%, the depreciation of the value of the asset in the fifth year can be expressed as

```
y5=depdbsl(5,1000,10,1.5);
```

The value 87.001041667 is returned. The first and the third arguments are expressed in years.

DEPSL Function

Returns the straight-line depreciation.

Category: Financial

Syntax

DEPSL(*p*,*v*,*y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For fractional *p*, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset.

Range $y > 0$

Details

The DEPSL function returns the straight-line depreciation, which is given by

$$DEPSL(p, v, y) = DACCSL(p, v, y) - DACCSL(p - 1, v, y)$$

The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01APR86, has a depreciable initial value of $1,000 and a ten-year lifetime. The depreciation in the value of the asset for the year 1986 can be expressed as

```
d=depsl(9/12,1000,10);
```

The value returned is 75.00. The first and the third arguments are expressed in years.

DEPSYD Function

Returns the sum-of-years-digits depreciation.

Category: Financial

Syntax

DEPSYD(*p*,*v*,*y*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

y
is numeric, the lifetime of the asset in number of depreciation periods.

Range $y > 0$

Details

The DEPSYD function returns the sum-of-years-digits depreciation, which is given by

$$DEPSYD(p, v, y) = DACCSYD(p, v, y) - DACCSYD(p - 1, v, y)$$

The *p* and *y* arguments must be expressed by using the same units of time.

Example

An asset, acquired on 01OCT86, has a depreciable initial value of $1,000 and a five-year lifetime. The depreciations in the value of the asset for the years 1986 and 1987 can be expressed as

```
y1=depsyd(3/12,1000,5);
y2=depsyd(15/12,1000,5);
```

The values returned are 83.33 and 316.67, respectively. The first and the third arguments are expressed in years.

DEPTAB Function

Returns the depreciation from specified tables.

Category: Financial

Syntax

DEPTAB(*p*,*v*,*t1*,...,*tn*)

Required Arguments

p
is numeric, the period for which the calculation is to be done. For noninteger *p* arguments, the depreciation is prorated between the two consecutive time periods that precede and follow the fractional period.

v
is numeric, the depreciable initial value of the asset.

t1,t2, . . . ,tn
are numeric, the fractions of depreciation for each time period with *t1+t2+...tn* ≤ 1.

Details

The DEPTAB function returns the depreciation by using specified tables. The formula is

$$DEPTAB(p,\ v,\ t_1,\ t_2,\ ...,\ t_n) = DACCTAB(p,\ v,\ t_1,\ t_2,\ ...,\ t_n) - DACCTAB(p-1,\ v,\ t_1,\ t_2,\ ...,\ t_n)$$

For a given *p*, only the arguments $t_1,\ t_2,\ \ldots,\ t_k$ need to be specified with **k=ceil(p)**.

Example

An asset has a depreciable initial value of $1,000 and a five-year lifetime. Using a table of the annual depreciation rates of .15, .22, .21, .21, and .21 during the first, second, third, fourth, and fifth years, respectively, the depreciation in the third year can be expressed as

```
y3=deptab(3,1000,.15,.22,.21,.21,.21);
```

The value that is returned is 210.00. The fourth rate, .21, and the fifth rate, .21, can be omitted because they are not needed in the calculation.

DEQUOTE Function

Removes matching quotation marks from a character string that begins with a quotation mark, and deletes all characters to the right of the closing quotation mark.

Category: Character

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

Syntax

DEQUOTE(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the DEQUOTE function returns a value to a variable that has not been previously assigned a length, then that variable is given the length of the argument.

The Basics

The value that is returned by the DEQUOTE function is determined as follows:

- If the first character of *string* is not a single or double quotation mark, DEQUOTE returns *string* unchanged.
- If the first two characters of *string* are both single quotation marks or both double quotation marks, and the third character is not the same type of quotation mark, then DEQUOTE returns a result with a length of zero.
- If the first character of *string* is a single quotation mark, the DEQUOTE function removes that single quotation mark from the result. DEQUOTE then scans *string* from left to right, looking for more single quotation marks. Each pair of consecutive, single quotation marks is reduced to one single quotation mark. The first single quotation mark that does not have an ending quotation mark in *string* is removed and all characters to the right of that quotation mark are also removed.
- If the first character of *string* is a double quotation mark, the DEQUOTE function removes that double quotation mark from the result. DEQUOTE then scans *string* from left to right, looking for more double quotation marks. Each pair of consecutive, double quotation marks is reduced to one double quotation mark. The first double quotation mark that does not have an ending quotation mark in *string* is removed and all characters to the right of that quotation mark are also removed.

Note: If *string* is a constant enclosed by quotation marks, those quotation marks are not part of the value of *string*. Therefore, you do not need to use DEQUOTE to remove the quotation marks that denote a constant.

Example

This example demonstrates the use of DEQUOTE within a DATA step.

```
data test;
   input string $60.;
   result = dequote(string);
   datalines;
No quotation marks, no change
No "leading" quotation marks, no change
"Matching double quotation marks are removed"
'Matching single quotation marks are removed'
"Paired ""quotation marks"" are reduced"
'Paired '' quotation marks '' are reduced'
"Single 'quotation marks' inside '' double'' quotation marks are unchanged"
'Double "quotation marks" inside ""single"" quotation marks are unchanged'
"No matching quotation mark, no problem
Don't remove this apostrophe
"Text after the matching quotation mark" is "deleted"
;
proc print noobs;
title 'Input Strings and Output Results from DEQUOTE';
run;
```

Display 2.29 *Removing Matching Quotation Marks with the DEQUOTE Function*

Input Strings and Output Results from DEQUOTE

string	result
No quotation marks, no change	No quotation marks, no change
No "leading" quotation marks, no change	No "leading" quotation marks, no change
"Matching double quotation marks are removed"	Matching double quotation marks are removed
'Matching single quotation marks are removed'	Matching single quotation marks are removed
"Paired ""quotation marks"" are reduced"	Paired "quotation marks" are reduced
'Paired '' quotation marks '' are reduced'	Paired ' quotation marks ' are reduced
"Single 'quotation marks' inside '' double" quotation marks	Single 'quotation marks' inside '' double" quotation marks
'Double "quotation marks" inside ""single"" quotation marks	Double "quotation marks" inside ""single"" quotation marks
"No matching quotation mark, no problem	No matching quotation mark, no problem
Don't remove this apostrophe	Don't remove this apostrophe
"Text after the matching quotation mark" is "deleted"	Text after the matching quotation mark

DEVIANCE Function

Returns the deviance based on a probability distribution.

Category: Mathematical

Syntax

DEVIANCE(*distribution*, *variable*, *shape-parameters*<,*ε*>)

Required Arguments

distribution
is a character constant, variable, or expression that identifies the distribution. Valid distributions are listed in the following table:

Distribution	Argument
Bernoulli	`'BERNOULLI' | 'BERN'`
Binomial	`'BINOMIAL' | 'BINO'`
Gamma	`'GAMMA'`
Inverse Gauss (Wald)	`'IGAUSS' | 'WALD'`
Normal	`'NORMAL' | 'GAUSSIAN'`
Poisson	`'POISSON' | 'POIS'`

variable
is a numeric constant, variable, or expression.

shape-parameter
are one or more distribution-specific numeric parameters that characterize the shape of the distribution.

Optional Argument

ε
is an optional numeric small value used for all of the distributions, except for the normal distribution.

Details

The Bernoulli Distribution

DEVIANCE('BERNOULLI', *variable*, *p*<, *ε*>)

Arguments

variable
is a binary numeric random variable that has the value of 1 for success and 0 for failure.

p
is a numeric probability of success with $\varepsilon \le p \le 1-\varepsilon$.

ε
is an optional positive numeric value that is used to bound p. Any value of p in the interval $0 \le p \le \varepsilon$ is replaced by ε. Any value of p in the interval $1 - \varepsilon \le p \le 1$ is replaced by $1 - \varepsilon$.

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:

$$DEVIANCE('BERN', variable, p, \varepsilon) = \begin{cases} -2\log(1-p) & x = 0 \\ -2\log(p) & x = 1 \\ . & otherwise \end{cases}$$

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 an integer number of Bernoulli trials parameter

Range $n \ge 0$

ε
is an optional positive numeric value that is used to bound μ. Any value of μ in the interval $0 \le \mu \le n\varepsilon$ is replaced by $n\varepsilon$. Any value of μ in the interval $n(1-\varepsilon) \le \mu \le n$ is replaced by $n(1-\varepsilon)$.

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.

$$DEVIANCE('BINO', x, \mu, n) = \begin{cases} . & x < 0 \\ 2\left(x\log\left(\frac{x}{\mu}\right) + (n-x)\log\left(\frac{n-x}{n-\mu}\right)\right) & 0 \le x \le n \\ . & x > n \end{cases}$$

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:

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

The Inverse Gauss (Wald) Distribution

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

Arguments

variable
is a numeric random variable.

Range $variable \geq \varepsilon$

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:

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

The Normal Distribution

DEVIANCE('NORMAL' | 'GAUSSIAN', *variable*, *μ*)

Arguments

variable
is a numeric random variable.

μ
is a numeric mean parameter.

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

$$DEVIANCE('NORMAL', x, \mu) = (x - \mu)^2$$

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:

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

DHMS Function

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

Category: Date and Time

Syntax

DHMS(*date*,*hour*,*minute*,*second*)

Required Arguments

date
specifies a SAS expression that represents a SAS date value.

hour
is numeric.

minute
is numeric.

second
is numeric.

Details

The DHMS function returns a numeric value that represents a SAS datetime value. This numeric value can be either positive or negative.

Example

The following SAS statements produce these results:

SAS Statement	Result

`dtid=dhms('01jan03'd,15,30,15);`
`put dtid;`
`put dtid datetime.;`
	`1357054215`
	`01JAN03:15:30:15`

`dtid2=dhms('01jan03'd,15,30,61);`
`put dtid2;`
`put dtid2 datetime.;`
	`1357054261`
	`01JAN03:15:31:01`

`dtid3=dhms('01jan03'd,15,.5,15);`
`put dtid3;`
`put dtid3 datetime.;`
	`1357052445`
	`01JAN03:15:00:45`

The following SAS statements show how to combine a SAS date value with a SAS time value into a SAS datetime value. If you execute these statements on April 2, 2003 at the time of 15:05:02, it produces these results:

SAS Statement	Result

`day=date();`
`time=time();`
`sasdt=dhms(day,0,0,time);`
`put sasdt datetime.;`
	`02APR03:15:05:02`

See Also

Functions:

- "HMS Function" on page 530

DIF Function

Returns differences between an argument and its *n*th lag.

Category: Special

Syntax

DIF<*n*> (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Optional Argument

n
specifies the number of lags.

Details

The DIF functions, DIF1, DIF2, ..., DIF100, return the first differences between the argument and its *n*th lag. DIF1 can also be written as DIF. DIF*n* is defined as DIF*n*(*x*)=*x*-LAG*n*(*x*).

For details about storing and returning values from the LAG*n* queue, see the LAG function.

Comparisons

The function DIF2(X) is not equivalent to the second difference DIF(DIF(X)).

Example

This example demonstrates the difference between the LAG and DIF functions.

```
data two;
   input X @@;
   Z=lag(x);
   D=dif(x);
   datalines;
1 2 6 4 7
;
proc print data=two;
run;
```

Display 2.30 *Difference between the DIF and LAG Functions*

The SAS System

Obs	X	Z	D
1	1	.	.
2	2	1	1
3	6	2	4
4	4	6	-2
5	7	4	3

See Also

Functions:

- "LAG Function" on page 605

DIGAMMA Function

Returns the value of the digamma function.

Category: Mathematical

Syntax

DIGAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Restriction Nonpositive integers are invalid.

Details

The DIGAMMA function returns the ratio that is given by

$$\Psi(x) = \Gamma'(x) / \Gamma(x)$$

where $\Gamma(.)$ and $\Gamma'(.)$ denote the Gamma function and its derivative, respectively. For *argument*>0, the DIGAMMA function is the derivative of the LGAMMA function.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=digamma(1.0);`	`-0.577215665`

DIM Function

Returns the number of elements in an array.

Category: Array

Syntax

DIM<*n*> (*array-name*)

DIM(*array-name*,*bound-n*)

Required Arguments

array-name
specifies the name of an array that was previously defined in the same DATA step. This argument cannot be a constant, variable, or expression.

bound-n
is a numeric constant, variable, or expression that specifies the dimension, in a multidimensional array, for which you want to know the number of elements. Use *bound-n* only when *n* is not specified.

Optional Argument

n
specifies the dimension, in a multidimensional array, for which you want to know the number of elements. If no *n* value is specified, the DIM function returns the number of elements in the first dimension of the array.

Details

The DIM function returns the number of elements in a one-dimensional array or the number of elements in a specified dimension of a multidimensional array when the lower bound of the dimension is 1. Use DIM in array processing to avoid changing the upper bound of an iterative DO group each time you change the number of array elements.

Comparisons

- DIM always returns a total count of the number of elements in an array dimension.
- HBOUND returns the literal value of the upper bound of an array dimension.

Note: This distinction is important when the lower bound of an array dimension has a value other than 1 and the upper bound has a value other than the total number of elements in the array dimension.

Examples

Example 1: One-dimensional Array

In this example, DIM returns a value of 5. Therefore, SAS repeats the statements in the DO loop five times.

```
array big{5} weight sex height state city;
do i=1 to dim(big);
  more SAS statements;
end;
```

Example 2: Multidimensional Array

This example shows two ways of specifying the DIM function for multidimensional arrays. Both methods return the same value for DIM, as shown in the table that follows the SAS code example.

```
array mult{5,10,2} mult1-mult100;
```

Syntax	Alternative Syntax	Value
DIM(MULT)	DIM(MULT,1)	5
DIM2(MULT)	DIM(MULT,2)	10
DIM3(MULT)	DIM(MULT,3)	2

See Also

Functions:

- “HBOUND Function” on page 528
- “LBOUND Function” on page 612

Statements:

- “ARRAY Statement” in *SAS Statements: Reference*
- “Array Reference Statement” in *SAS Statements: Reference*

Other References:

- “Array Processing” in Chapter 23 of *SAS Language Reference: Concepts*

DINFO Function

Returns information about a directory.

Category: External Files

See: “DINFO Function: Windows” in *SAS Companion for Windows*
“DINFO Function: UNIX” in *SAS Companion for UNIX Environments*
“DINFO Function: z/OS” in *SAS Companion for z/OS*

Syntax

DINFO(*directory-id*,*info-item*)

Required Arguments

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

info-item
is a character constant, variable, or expression that specifies the information item to be retrieved. DINFO returns a blank if the value of the *info-item* argument is invalid. The information available varies according to the operating environment.

Details

Use the DOPTNAME function to determine the names of the available system-dependent directory information items. Use the DOPTNUM function to determine the number of directory information items that are available.

Operating Environment Information
DINFO returns the value of a system-dependent directory parameter. See the SAS documentation for your operating environment for information about system-dependent directory parameters.

Examples

Example 1: Using DINFO to Return Information about a Directory

This example opens the directory MYDIR, determines the number of directory information items available, and retrieves the value of the last one:

```
%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let numopts=%sysfunc(doptnum(&did));
%let foption=%sysfunc(doptname(&did,&numopts));
%let charval=%sysfunc(dinfo(&did,&foption));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DINFO within a DATA Step

This example creates a data set that contains the name and value of each directory information item:

```
data diropts;
   length foption $ 12 charval $ 40;
   keep foption charval;
   rc=filename("mydir","physical-name");
   did=dopen("mydir");
   numopts=doptnum(did);
   do i=1 to numopts;
      foption=doptname(did,i);
      charval=dinfo(did,foption);
      output;
   end;
run;
```

See Also

Functions:

- "DOPEN Function" on page 382
- "DOPTNAME Function" on page 384
- "DOPTNUM Function" on page 385
- "FINFO Function" on page 473
- "FOPTNAME Function" on page 488
- "FOPTNUM Function" on page 489

DIVIDE Function

Returns the result of a division that handles special missing values for ODS output.

Category: Arithmetic

Syntax

DIVIDE(*x*, *y*)

Required Arguments

x
is a numeric constant, variable, or expression.

y
is a numeric constant, variable, or expression.

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*.

Display 2.31 *Values That Are Returned by the DIVIDE Function*

		x						
		positive	zero	negative	.I	.M	._	other
y	positive	x/y or .I	0	x/y or .M	.I	.M	._	x
	zero	.I	.	.M	.I	.M	._	x
	negative	x/y or .M	0	x/y or .I	.M	.I	._	x
	.I	0	0	0	.	.	._	x
	.M	0	0	0	.	.	._	x
	._	._	._	._	._	._	._	._
	other	y	y	y	y	y	._	x

Note: The DIVIDE function never writes a note to the SAS log regarding missing values, division by zero, or overflow.

Example

The following example shows the results of using the DIVIDE function.

```
data _null_;
   a = divide(1, 0);
   put +3 a= '(infinity)';
   b = divide(2, .I);
   put +3 b=;
   c = divide(.I, -1);
   put +3 c= '(minus infinity)';
   d = divide(constant('big'), constant('small'));
   put +3 d= '(infinity because of overflow)';
run;
```

SAS writes the following output to the log:

```
   a=I (infinity)
   b=0
   c=M (minus infinity)
   d=I (infinity because of overflow)
```

DNUM Function

Returns the number of members in a directory.

Category: External Files

Syntax

DNUM(*directory-id*)

Required Argument

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

Details

You can use DNUM to determine the highest possible member number that can be passed to DREAD.

Examples

Example 1: Using DNUM to Return the Number of Members

This example opens the directory MYDIR, determines the number of members, and closes the directory:

```
%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let memcount=%sysfunc(dnum(&did));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DNUM within a DATA Step

This example creates a DATA step that returns the number of members in a directory called MYDIR:

```
data _null_;
   rc=filename("mydir","physical-name");
   did=dopen("mydir");
   memcount=dnum(did);
   rc=dclose(did);
run;
```

See Also

Functions:

- “DOPEN Function” on page 382
- “DREAD Function” on page 386

DOPEN Function

Opens a directory, and returns a directory identifier value.

Category: External Files

See: “DOPEN Function: Windows” in *SAS Companion for Windows*
“DOPEN Function: UNIX” in *SAS Companion for UNIX Environments*
“DOPEN Function: z/OS” in *SAS Companion for z/OS*

Syntax

DOPEN(*fileref*)

Required Argument

fileref
is a character constant, variable, or expression that specifies the fileref assigned to the directory.

Restriction You must associate a fileref with the directory before calling DOPEN.

Details

DOPEN opens a directory and returns a directory identifier value (a number greater than 0) that is used to identify the open directory in other SAS external file access functions. If the directory could not be opened, DOPEN returns 0, and you can obtain the error message by calling the SYSMSG function. The directory to be opened must be identified by a fileref. You can assign filerefs using the FILENAME statement or the FILENAME external file access function. Under some operating environments, you can also assign filerefs using system commands.

If you call the DOPEN function from a macro, then the result of the call is valid only when the result is passed to functions in a macro. If you call the DOPEN function from the DATA step, then the result is valid only when the result is passed to functions in the same DATA step.

Operating Environment Information
The term *directory* that is used in the description of this function and related SAS external file access functions refers to an aggregate grouping of files managed by the operating environment. Different operating environments identify such groupings with different names, such as directory, subdirectory, folder, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Using DOPEN to Open a Directory

This example assigns the fileref MYDIR to a directory. It uses DOPEN to open the directory. DOPTNUM determines the number of system-dependent directory information items available, and DCLOSE closes the directory:

```
%let filrf=MYDIR;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DOPEN within a DATA Step

This example opens a directory for processing within a DATA step.

```
data _null_;
   drop rc did;
   rc=filename("mydir","physical-name");
   did=dopen("mydir");
   if did > 0 then do;
      ...more statements...
   end;
   else do;
      msg=sysmsg();
```

```
      put msg;
   end;
run;
```

See Also

Functions:

- “DCLOSE Function” on page 361
- “DOPTNUM Function” on page 385
- “FOPEN Function” on page 485
- “MOPEN Function” on page 670
- “SYSMSG Function” on page 905

DOPTNAME Function

Returns directory attribute information.

Category: External Files

See: “DOPTNAME Function: Windows” in *SAS Companion for Windows*
“DOPTNAME Function: UNIX” in *SAS Companion for UNIX Environments*
“DOPTNAME Function: z/OS” in *SAS Companion for z/OS*

Syntax

DOPTNAME(*directory-id*,*nval*)

Required Arguments

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

nval
is a numeric constant, variable, or expression that specifies the sequence number of the option.

Details

Operating Environment Information
The number, names, and nature of the directory information varies between operating environments. The number of options that are available for a directory varies depending on the operating environment. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Using DOPTNAME to Retrieve Directory Attribute Information

This example opens the directory with the fileref MYDIR, retrieves all system-dependent directory information items, writes them to the SAS log, and closes the directory:

```
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%do j=1 %to &infocnt;
   %let opt=%sysfunc(doptname(&did,&j));
   %put Directory information=&opt;
%end;
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DOPTNAME within a DATA Step

This example creates a data set that contains the name and value of each directory information item:

```
data diropts;
   length optname $ 12 optval $ 40;
   keep optname optval;
   rc=filename("mydir","physical-name");
   did=dopen("mydir");
   numopts=doptnum(did);
   do i=1 to numopts;
      optname=doptname(did,i);
      optval=dinfo(did,optname);
      output;
   end;
   run;
```

See Also

Functions:

- “DINFO Function” on page 378
- “DOPEN Function” on page 382
- “DOPTNUM Function” on page 385

DOPTNUM Function

Returns the number of information items that are available for a directory.

Category: External Files

See: “DOPTNUM Function: Windows” in *SAS Companion for Windows*
“DOPTNUM Function: UNIX” in *SAS Companion for UNIX Environments*
“DOPTNUM Function: z/OS” in *SAS Companion for z/OS*

Syntax

DOPTNUM(*directory-id*)

Required Argument

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

Details

Operating Environment Information
The number, names, and nature of the directory information varies between operating environments. The number of options that are available for a directory varies depending on the operating environment. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Retrieving the Number of Information Items

This example retrieves the number of system-dependent directory information items that are available for the directory MYDIR and closes the directory:

```
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let infocnt=%sysfunc(doptnum(&did));
%let rc=%sysfunc(dclose(&did));
```

Example 2: Using DOPTNUM within a DATA Step

This example creates a data set that retrieves the number of system-dependent information items that are available for the MYDIR directory:

```
data _null_;
   rc=filename("mydir","physical-name");
   did=dopen("mydir");
   infocnt=doptnum(did);
   rc=dclose(did);
run;
```

See Also

Functions:

- “DINFO Function” on page 378
- “DOPEN Function” on page 382
- “DOPTNAME Function” on page 384

DREAD Function

Returns the name of a directory member.

Category: External Files

Syntax

DREAD(*directory-id*,*nval*)

Required Arguments

directory-id
is a numeric value that specifies the identifier that was assigned when the directory was opened by the DOPEN function.

nval
is a numeric constant, variable, or expression that specifies the sequence number of the member within the directory.

Details

DREAD returns a blank if an error occurs (such as when *nval* is out-of-range). Use DNUM to determine the highest possible member number that can be passed to DREAD.

Example

This example opens the directory identified by the fileref MYDIR, retrieves the number of members, and places the number in the variable MEMCOUNT. It then retrieves the name of the last member, places the name in the variable LSTNAME , and closes the directory:

```
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let lstname=;
%let memcount=%sysfunc(dnum(&did));
%if &memcount > 0 %then
   %let lstname=%sysfunc(dread(&did,&memcount));
%let rc=%sysfunc(dclose(&did));
```

See Also

Functions:

- “DNUM Function” on page 381
- “DOPEN Function” on page 382

DROPNOTE Function

Deletes a note marker from a SAS data set or an external file.

Categories: SAS File I/O
External Files

Syntax

DROPNOTE(*data-set-id | file-id*,*note-id*)

Required Arguments

data-set-id* | *file-id
is a numeric variable that specifies the identifier that was assigned when the data set or external file was opened, generally by the OPEN function or the FOPEN function.

note-id
is a numeric value that specifies the identifier that was assigned by the NOTE or FNOTE function.

Details

DROPNOTE deletes a marker set by NOTE or FNOTE. It returns a 0 if successful and ≠0 if not successful.

Example

This example opens the SAS data set MYDATA, fetches the first observation, and sets a note ID at the beginning of the data set. It uses POINT to return to the first observation, and then uses DROPNOTE to delete the note ID:

```
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetch(&dsid));
%let noteid=%sysfunc(note(&dsid));
   more macro statements
%let rc=%sysfunc(point(&dsid,&noteid));
%let rc=%sysfunc(fetch(&dsid));
%let rc=%sysfunc(dropnote(&dsid,&noteid));
```

See Also

Functions:

- “FETCH Function” on page 405
- “FNOTE Function” on page 484
- “FOPEN Function” on page 485
- “FPOINT Function” on page 490
- “NOTE Function” on page 692
- “OPEN Function” on page 716
- “POINT Function” on page 746

DSNAME Function

Returns the SAS data set name that is associated with a data set identifier.

Category: SAS File I/O

Syntax

DSNAME(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that is returned by the OPEN function.

Details

DSNAME returns the data set name that is associated with a data set identifier, or a blank if the data set identifier is not valid.

Example

This example determines the name of the SAS data set that is associated with the variable DSID and displays the name in the SAS log.

```
%let dsid=%sysfunc(open(sasuser.houses,i));
%put The current open data set
is %sysfunc(dsname(&dsid)).;
```

See Also

Functions:

- "OPEN Function" on page 716

DUR Function

Returns the modified duration for an enumerated cash flow.

Category: Financial

Syntax

DUR(*y,f,c(1), ... ,c(k)*)

Required Arguments

y
specifies the effective per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

f
specifies the frequency of cash flows per period.

Range $f > 0$

c(1), ... ,c(k)
specifies a list of cash flows.

Details

The DUR function returns the value

$$C = \sum_{k=1}^{K} \frac{k\left(\frac{c(k)}{(1+y)^{\frac{k}{f}}}\right)}{(P(1+y)f)}$$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{(1+y)^{\frac{k}{f}}}$$

Example

```
data _null_;
   d=dur(1/20,1,.33,.44,.55,.49,.50,.22,.4,.8,.01,.36,.2,.4);
   put d;
run;
```

The value that is returned is 5.28402.

DURP Function

Returns the modified duration for a periodic cash flow stream, such as a bond.

Category: Financial

Syntax

DURP(*A*,*c*,*n*,*K*,k_0,*y*)

Required Arguments

A

specifies the par value.

Range $A > 0$

c

specifies the nominal per-period coupon rate, expressed as a fraction.

Range $0 \leq c < 1$

n

specifies the number of coupons per period.

Range $n > 0$ and is an integer

K

specifies the number of remaining coupons.

Range $K > 0$ and is an integer

***k*$_0$**

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

Range $0 < k_0 \leq 1/n$

y

specifies the nominal per-period yield-to-maturity, expressed as a fraction.

Range $y > 0$

Details

The DURP function returns the value

$$D = \frac{1}{n} \frac{\sum_{k=1}^{K} t_k \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{t_k}}}{P\left(1 + \frac{y}{n}\right)}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ \ldots, \ K - 1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

The following relationship applies to the preceding equation:

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

Example

```
data _null_;
d=durp(1000,1/100,4,14,.33/2,.10);
put d;
run;
```

The value returned is 3.26496.

EFFRATE Function

Returns the effective annual interest rate.

Category: Financial

Syntax

EFFRATE(*compounding-interval*, *rate*)

Required Arguments

compounding-interval
is a SAS interval. This value represents how often *rate* compounds.

rate
is numeric. *rate* is a nominal annual interest rate (expressed as a percentage) that is compounded at each compounding interval.

Details

The EFFRATE function returns the effective annual interest rate. The function computes the effective annual interest rate that corresponds to a nominal annual interest rate.

The following details apply to the EFFRATE function:

- The values for rates must be at least –99.
- In considering a nominal interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by EFFRATE equals $e^{rate/100}-1$.

  If *compounding-interval* is not 'CONTINUOUS', and *m* compounding intervals occur in a year, the value that is returned by EFFRATE equals $(1+[rate/100\ m])^{m-1}$.
- The following values are valid for *compounding-interval*:
  - 'CONTINUOUS'
  - 'DAY'
  - 'SEMIMONTH'
  - 'MONTH'
  - 'QUARTER'
  - 'SEMIYEAR'
  - 'YEAR'
- If the interval is 'DAY', then *m*=365.

Example

- 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);
```

ENVLEN Function

Returns the length of an environment variable.

Category: SAS File I/O

Syntax

ENVLEN(*argument*)

Required Argument

argument
specifies a character variable that is the name of an operating system environment variable. Enclose *argument* in quotation marks.

Details

The ENVLEN function returns the length of the value of an operating system environment variable. If the environment variable does not exist, SAS returns –1.

Operating Environment Information
The value of *argument* is specific to your operating environment.

Example

The following examples are for illustration purposes only. The actual value that is returned depends on where SAS is installed on your computer.

SAS Statement	Result

`/* Windows operating environment */`
`x=envlen("PATH");`
`put x;`
	`309`

`/* UNIX operating environment */`
`y=envlen("PATH");`
`put y;`
	`365`

`z=envlen("THIS IS NOT DEFINED");` `put z;`	`-1`

ERF Function

Returns the value of the (normal) error function.

Category: Mathematical

Syntax

ERF(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The ERF function returns the integral, given by

$$ERF(x) = \frac{2}{\sqrt{\pi}} \int_0^x \varepsilon^{-z^2} dz$$

Example

You can use the ERF function to find the probability (p) that a normally distributed random variable with mean 0 and standard deviation will take on a value less than X. For example, the quantity that is given by the following statement is equivalent to PROBNORM(X):

```
p=.5+.5*erf(x/sqrt(2));
```

The following SAS statements produce these results.

SAS Statement	Result
`y=erf(1.0);`	`0.8427007929`
`y=erf(-1.0);`	`-0.842700793`

ERFC Function

Returns the value of the complementary (normal) error function.

Category: Mathematical

Syntax

ERFC(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The ERFC function returns the complement to the ERF function (that is, 1 − ERF(*argument*)).

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=erfc(1.0);`	`0.1572992071`

SAS Statement	Result
`x=erfc(-1.0);`	`.8427007929`

EUCLID Function

Returns the Euclidean norm of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

EUCLID(*value-1* <,*value-2 ...*>)

Required Argument

value
specifies a numeric constant, variable, or expression.

Details

If all arguments have missing values, then the result is a missing value. Otherwise, the result is the Euclidean norm of the nonmissing values.

In the following example, *x*1, *x*2, ..., *xn* are the values of the nonmissing arguments.

$$EUCLID(x1,\ x2,\ ...,\ xn) = \sqrt{x1^2 + x2^2 + ... + xn^2}$$

Examples

Example 1: Calculating the Euclidean Norm of Nonmissing Arguments

The following example returns the Euclidean norm of the nonmissing arguments.

```
data _null_;
   x=euclid(.,3,0,.q,-4);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=5
```

Example 2: Calculating the Euclidean Norm When You Use a Variable List

The following example uses a variable list to calculate the Euclidean norm.

```
data _null_;
   x1 = 1;
   x2 = 3;
   x3 = 4;
   x4 = 3;
```

```
   x5 = 1;
   x = euclid(of x1-x5);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=6
```

See Also

Functions:

- "RMS Function" on page 833
- "LPNORM Function" on page 648

EXIST Function

Verifies the existence of a SAS library member.

Category: SAS File I/O

Syntax

EXIST(*member-name*<,*member-type*<, *generation*> >)

Required Argument

member-name
is a character constant, variable, or expression that specifies the SAS library member. If *member-name* is blank or a null string, then EXIST uses the value of the _LAST_ system variable as the member name.

Optional Arguments

member-type
is a character constant, variable, or expression that specifies the type of SAS library member. A few common member types include ACCESS, CATALOG, DATA, and VIEW. If you do not specify a *member-type*, then the member type DATA is assumed.

generation
is a numeric constant, variable, or expression that specifies the generation number of the SAS data set whose existence you are checking. If *member-type* is not DATA, *generation* is ignored.

Positive numbers are absolute references to a historical version by its generation number. Negative numbers are relative references to a historical version in relation to the base version, from the youngest predecessor to the oldest. For example, –1 refers to the youngest version or, one version back from the base version. Zero is treated as a relative generation number.

Details

If you use a sequential library, then the results of the EXIST function are undefined. If you do *not* use a sequential library, then EXIST returns 1 if the library member exists, or 0 if *member-name* does not exist or *member-type* is invalid.

Use the CEXIST function to verify the existence of an entry in a catalog.

Examples

Example 1: Verifying the Existence of a Data Set

This example verifies the existence of a data set. If the data set does not exist, then the example displays a message in the log:

```
%let dsname=sasuser.houses;
%macro opends(name);
%if %sysfunc(exist(&name)) %then
   %let dsid=%sysfunc(open(&name,i));
%else %put Data set &name does not exist.;
%mend opends;
%opends(&dsname);
```

Example 2: Verifying the Existence of a Data View

This example verifies the existence of the SAS view TEST.MYVIEW. If the view does not exist, then the example displays a message in the log:

```
data _null_;
dsname="test.myview";
   if (exist(dsname,"VIEW")) then
      dsid=open(dsname,"i");
   else put dsname 'does not exist.';
run;
```

Example 3: Determining If a Generation Data Set Exists

This example verifies the existence of a generation data set by using positive generation numbers (absolute reference):

```
data new(genmax=3);
   x=1;
run;
data new;
   x=99;
run;
data new;
   x=100;
run;
data new;
   x=101;
run;
data _null_;
   test=exist('new', 'DATA', 4);
   put test=;
   test=exist('new', 'DATA', 3);
   put test=;
   test=exist('new', 'DATA', 2);
   put test=;
```

```
   test=exist('new', 'DATA', 1);
   put test=;
run;
```

These lines are written to the SAS log:

```
test=1
test=1
test=1
test=0
```

You can change this example to verify the existence of the generation data set by using negative numbers (relative reference):

```
data new2(genmax=3);
   x=1;
run;
data new2;
   x=99;
run;
data new2;
   x=100;
run;
data new2;
   x=101;
run;
data _null_;
   test=exist('new2', 'DATA', 0);
   put test=;
   test=exist('new2', 'DATA', -1);
   put test=;
   test=exist('new2', 'DATA', -2);
   put test=;
   test=exist('new2', 'DATA', -3);
   put test=;
   test=exist('new2', 'DATA', -4);
   put test=;
run;
```

These lines are written to the SAS log:

```
test=1
test=1
test=1
test=0
test=0
```

See Also

Functions:

- “CEXIST Function” on page 297
- “FEXIST Function” on page 408
- “FILEEXIST Function” on page 410

EXP Function

Returns the value of the exponential function.

Category: Mathematical

Syntax

EXP(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression. For more information, see "Definitions for SAS Expressions" in Chapter 6 of *SAS Language Reference: Concepts*.

Details

The EXP function raises the constant *e*, which is approximately 2.71828, to the power that is supplied by the argument. The result is limited by the maximum value of a floating-point value on the computer.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=exp(1.0);`	`2.7182818285`
`x=exp(0);`	`1`

See Also

SAS Language Reference: Concepts

- "Arithmetic Operators" in Chapter 6 of *SAS Language Reference: Concepts*

FACT Function

Computes a factorial.

Category: Mathematical

Syntax

FACT(*n*)

Required Argument

n
is a numeric constant, variable, or expression.

Details

The mathematical representation of the FACT function is given by the following equation:

$$FACT(n) = n!$$

with $n \geq 0$.

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 SAS statement produces this result.

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

See Also

Functions:

- “COMB Function” on page 310
- “PERM Function” on page 743
- “LFACT Function” on page 632

FAPPEND Function

Appends the current record to the end of an external file.

Category: External Files

Syntax

FAPPEND(*file-id*<,*cc*>)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Optional Argument

cc

is a character constant, variable, or expression that specifies a carriage-control character:

blank	indicates that the record starts a new line.
0	skips one blank line before this new line.
-	skips two blank lines before this new line.
1	specifies that the line starts a new page.
+	specifies that the line overstrikes a previous line.
P	specifies that the line is a computer prompt.
=	specifies that the line contains carriage control information.
all else	specifies that the line record starts a new line.

Details

FAPPEND adds the record that is currently contained in the File Data Buffer (FDB) to the end of an external file. FAPPEND returns a 0 if the operation was successful and ≠0 if it was not successful.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it moves data into the File Data Buffer, appends a record, and then closes the file. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf,a));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fput(&fid,
                      Data for the new record));
      %let rc=%sysfunc(fappend(&fid));
      %let rc=%sysfunc(fclose(&fid));
   %end;
%else
   %do;
      /* unsuccessful open processing */
   %end;
```

See Also

Functions:

- “DOPEN Function” on page 382
- “FCLOSE Function” on page 402
- “FGET Function” on page 409

- "FOPEN Function" on page 485
- "FPUT Function" on page 494
- "FWRITE Function" on page 501
- "MOPEN Function" on page 670

FCLOSE Function

Closes an external file, directory, or directory member.

Category: External Files

See: "FCLOSE Function: z/OS" in *SAS Companion for z/OS*

Syntax

FCLOSE(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FCLOSE returns a 0 if the operation was successful and ≠0 if it was not successful. If you open a file within a DATA step, it is closed automatically when the DATA step ends.

Operating Environment Information
In some operating environments you must close the file with the FCLOSE function at the end of the DATA step. For more information, see the SAS documentation for your operating environment.

Example

This example assigns the fileref MYFILE to an external file, and attempts to open the file. If the file is opened successfully, indicated by a positive value in the variable FID, the program reads the first record, closes the file, and deassigns the fileref:

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fread(&fid));
      %let rc=%sysfunc(fclose(&fid));
   %end;
%else
   %do;
      %put %sysfunc(sysmsg());
%end;
```

```
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:

- "DCLOSE Function" on page 361
- "DOPEN Function" on page 382
- "FOPEN Function" on page 485
- "FREAD Function" on page 495
- "MOPEN Function" on page 670

FCOL Function

Returns the current column position in the File Data Buffer (FDB).

Category: External Files

Syntax

FCOL(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

Use FCOL combined with FPOS to manipulate data in the File Data Buffer (FDB).

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is successfully opened, indicated by a positive value in the variable FID, it puts more data into the FDB relative to position POS, writes the record, and closes the file:

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf,o));
%if (&fid > 0) %then
   %do;
      %let record=This is data for the record.;
      %let rc=%sysfunc(fput(&fid,&record));
      %let pos=%sysfunc(fcol(&fid));
      %let rc=%sysfunc(fpos(&fid,%eval(&pos+1)));
      %let rc=%sysfunc(fput(&fid,more data));
      %let rc=%sysfunc(fwrite(&fid));
      %let rc=%sysfunc(fclose(&fid));
```

```
    %end;
%let rc=%sysfunc(filename(filrf));
```

The new record written to the external file is

```
This is data for the record. more data
```

See Also

Functions:

- “FCLOSE Function” on page 402
- “FOPEN Function” on page 485
- “FPOS Function” on page 492
- “FPUT Function” on page 494
- “FWRITE Function” on page 501
- “MOPEN Function” on page 670

FDELETE Function

Deletes an external file or an empty directory.

Category: External Files

See: “FDELETE Function: Windows” in *SAS Companion for Windows*
“FDELETE Function: UNIX” in *SAS Companion for UNIX Environments*
“FDELETE Function: z/OS” in *SAS Companion for z/OS*

Syntax

FDELETE(*fileref* | *directory*)

Required Arguments

fileref
is a character constant, variable, or expression that specifies the fileref that you assigned to the external file. You can assign filerefs by using the FILENAME statement or the FILENAME external file access function.

Restriction The fileref that you use with FDELETE cannot be a concatenation.

Windows specifics In some operating environments, you can specify a fileref that was assigned with an environment variable. You can also assign filerefs using system commands. For details, see the SAS documentation for your operating environment.

directory
is a character constant, variable, or expression that specifies an empty directory that you want to delete.

Restriction You must have authorization to delete the directory.

Details

FDELETE returns 0 if the operation was successful or ≠0 if it was not successful.

Examples

Example 1: Deleting an External File

This example generates a fileref for an external file in the variable FNAME. Then it calls FDELETE to delete the file and calls the FILENAME function again to deassign the fileref.

```
data _null_;
    fname="tempfile";
    rc=filename(fname,"physical-filename");
    if rc = 0 and fexist(fname) then
       rc=fdelete(fname);
    rc=filename(fname);
run;
```

Example 2: Deleting a Directory

This example uses FDELETE to delete an empty directory to which you have write access. If the directory is not empty, the optional SYSMSG function returns an error message stating that SAS is unable to delete the file.

```
filename testdir 'physical-filename';
data _null_;
   rc=fdelete('testdir');
   put rc=;
   msg=sysmsg();
   put msg=;
run;
```

See Also

Functions:

- “FEXIST Function” on page 408
- “FILENAME Function” on page 411

Statements:

- “FILENAME Statement” in *SAS Statements: Reference*

FETCH Function

Reads the next non-deleted observation from a SAS data set into the Data Set Data Vector (DDV).

Category: SAS File I/O

Syntax

FETCH(*data-set-id* <,'NOSET'>)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that is returned by the OPEN function.

Optional Argument

'NOSET'
prevents the automatic passing of SAS data set variable values to macro or DATA step variables even if the SET routine has been called.

Details

FETCH returns a 0 if the operation is successful, ≠0 if it is not successful, and − 1 if the end of the data set is reached. FETCH skips observations marked for deletion.

If the SET routine has been called previously, the values for any data set variables are automatically passed from the DDV to the corresponding DATA step or macro variables. To override this behavior temporarily so that fetched values are not automatically copied to the DATA step or macro variables, use the NOSET option.

Example

This example fetches the next observation from the SAS data set MYDATA. If the end of the data set is reached or if an error occurs, SYSMSG retrieves the appropriate message and writes it to the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetch(&dsid));
%if &rc ne 0 %then
  %put %sysfunc(sysmsg());
%else
   %do;
      ...more macro statements...
   %end;
%let rc=%sysfunc(close(&dsid));
```

See Also

Functions:

- “FETCHOBS Function” on page 406
- “GETVARC Function” on page 521
- “GETVARN Function” on page 522

CALL Routines:

- “CALL SET Routine” on page 246

FETCHOBS Function

Reads a specified observation from a SAS data set into the Data Set Data Vector (DDV).

Category: SAS File I/O

Syntax

FETCHOBS(*data-set-id*,*obs-number*<,*options*>)

Required Arguments

data-set-id
is a numeric variable that specifies the data set identifier that is returned by the OPEN function.

obs-number
is a numeric constant, variable, or expression that specifies the number of the observation to read. FETCHOBS treats the observation value as a relative observation number unless you specify the ABS option. The relative observation number might not coincide with the physical observation number on disk, because the function skips observations marked for deletion. When a WHERE clause is active, the function counts only observations that meet the WHERE condition.

Default FETCHOBS skips deleted observations.

Optional Argument

options
is a character constant, variable, or expression that names one or more options, separated by blanks:

ABS specifies that the value of *obs-number* is absolute. That is, deleted observations are counted.

NOSET prevents the automatic passing of SAS data set variable values to DATA step or macro variables even if the SET routine has been called.

Details

FETCHOBS returns 0 if the operation was successful, ≠0 if it was not successful, and −1 if the end of the data set is reached. To retrieve the error message that is associated with a nonzero return code, use the SYSMSG function. If the SET routine has been called previously, the values for any data set variables are automatically passed from the DDV to the corresponding DATA step or macro variables. To override this behavior temporarily, use the NOSET option.

If *obs-number* is less than 1, the function returns an error condition. If *obs-number* is greater than the number of observations in the SAS data set, the function returns an end-of-file condition.

Example

This example fetches the tenth observation from the SAS data set MYDATA. If an error occurs, the SYSMSG function retrieves the error message and writes it to the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let rc = %sysfunc(fetchobs(&mydataid,10));
%if &rc = -1 %then
```

```
   %put End of data set has been reached.;
%if &rc > 0 %then %put %sysfunc(sysmsg());
```

See Also

Functions:

- "FETCH Function" on page 405
- "GETVARC Function" on page 521
- "GETVARN Function" on page 522

CALL Routines:

- "CALL SET Routine" on page 246

FEXIST Function

Verifies the existence of an external file that is associated with a fileref.

Category: External Files

See: "FEXIST Function: Windows" in *SAS Companion for Windows*
"FEXIST Function: UNIX" in *SAS Companion for UNIX Environments*
"FEXIST Function: z/OS" in *SAS Companion for z/OS*

Syntax

FEXIST(*fileref*)

Required Argument

fileref
is a character constant, variable, or expression that specifies the fileref that is assigned to an external file.

Restriction The *fileref* must have been previously assigned.

Windows specifics In some operating environments, you can specify a fileref that was assigned with an environment variable. For details, see the SAS documentation for your operating environment.

Details

FEXIST returns 1 if the external file that is associated with *fileref* exists, and 0 if the file does not exist. You can assign filerefs by using the FILENAME statement or the FILENAME external file access function. In some operating environments, you can also assign filerefs by using system commands.

Comparisons

FILEEXIST verifies the existence of a file based on its physical name.

Example

This example verifies the existence of an external file and writes the result to the SAS log:

```
%if %sysfunc(fexist(&fref)) %then
    %put The file identified by the fileref
       &fref exists.;
 %else
    %put %sysfunc(sysmsg());
```

See Also

Functions:

- "EXIST Function" on page 396
- "FILEEXIST Function" on page 410
- "FILENAME Function" on page 411
- "FILEREF Function" on page 414

Statements:

- "FILENAME Statement" in *SAS Statements: Reference*

FGET Function

Copies data from the File Data Buffer (FDB) into a variable.

Category: External Files

Syntax

FGET(*file-id*,*variable*<,*length*>)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

variable
in a DATA step, specifies a character variable to hold the data. In a macro, specifies a macro variable to hold the data. If *variable* is a macro variable and it does not exist, it is created.

Optional Argument

length
specifies the number of characters to retrieve from the FDB. If *length* is specified, only the specified number of characters is retrieved (or the number of characters remaining in the buffer if that number is less than length). If *length* is omitted, all characters in the FDB from the current column position to the next delimiter are returned. The default delimiter is a blank. The delimiter is not retrieved.

See The "FSEP Function" on page 499 for more information about delimiters.

Details

FGET returns 0 if the operation was successful, or −1 if the end of the FDB was reached or no more tokens were available.

After FGET is executed, the column pointer moves to the next read position in the FDB.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it reads the first record into the File Data Buffer, retrieves the first token of the record and stores i t in the variable MYSTRING, and then closes the file. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fread(&fid));
      %let rc=%sysfunc(fget(&fid,mystring));
      %put &mystring;
      %let rc=%sysfunc(fclose(&fid));
   %end;
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:

- "FCLOSE Function" on page 402
- "FILENAME Function" on page 411
- "FOPEN Function" on page 485
- "FPOS Function" on page 492
- "FREAD Function" on page 495
- "FSEP Function" on page 499
- "MOPEN Function" on page 670

FILEEXIST Function

Verifies the existence of an external file by its physical name.

Category: External Files

See: "FILEEXIST Function: Windows" in *SAS Companion for Windows*
"FILEEXIST Function: UNIX" in *SAS Companion for UNIX Environments*
"FILEEXIST Function: z/OS" in *SAS Companion for z/OS*

Syntax

FILEEXIST(*file-name*)

Required Argument

file-name
is a character constant, variable, or expression that specifies a fully qualified physical filename of the external file in the operating environment.

Details

FILEEXIST returns 1 if the external file exists and 0 if the external file does not exist. The specification of the physical name for *file-name* varies according to the operating environment.

Although your operating environment utilities might recognize partial physical filenames, you must always use fully qualified physical filenames with FILEEXIST.

Example

This example verifies the existence of an external file. If the file exists, FILEEXIST opens the file. If the file does not exist, FILEEXIST displays a message in the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%if %sysfunc(fileexist(&myfilerf)) %then
    %let fid=%sysfunc(fopen(&myfilerf));
%else
   %put The external file &myfilerf does not exist.;
```

See Also

Functions:

- “EXIST Function” on page 396
- “FEXIST Function” on page 408
- “FILENAME Function” on page 411
- “FILEREF Function” on page 414
- “FOPEN Function” on page 485

FILENAME Function

Assigns or deassigns a fileref to an external file, directory, or output device.

Category: External Files

See: “FILENAME Function: Windows” in *SAS Companion for Windows*
“FILENAME Function: UNIX” in *SAS Companion for UNIX Environments*
“FILENAME Function: z/OS” in *SAS Companion for z/OS*

Syntax

FILENAME(*fileref* <,*file-name*> <,*device-type*> <,'*host-options*'> <,*dir-ref*>)

Required Argument

fileref
specifies the fileref to assign to the external file. In a DATA step, *fileref* can be a character expression, a string enclosed in quotation marks that specifies the fileref, or a DATA step variable whose value contains the fileref. In a macro (for example, in the %SYSFUNC function), *fileref* is the name of a macro variable (without an ampersand) whose value contains the fileref to assign to the external file.

Requirement If *fileref* is a DATA step variable, its length must be no longer than eight characters.

Tip If a fileref is a DATA step character variable with a blank value and a maximum length of eight characters, or if a macro variable named in *fileref* has a null value, then a fileref is generated and assigned to the character variable or macro variable, respectively.

Optional Arguments

file-name
is a character constant, variable, or expression that specifies the external file. Specifying a blank *file-name* deassigns a fileref that was assigned previously.

device-type
is a character constant, variable, or expression that specifies the type of device or the access method that is used if the fileref points to an input or output device or location that is not a physical file:

DISK
specifies that the device is a disk drive.

Alias BASE

Tip When you assign a fileref to a file on disk, you are not required to specify DISK.

DUMMY
specifies that the output to the file is discarded.

Tip Specifying DUMMY can be useful for testing.

GTERM
indicates that the output device type is a graphics device that will be receiving graphics data.

PIPE
specifies an unnamed pipe.

Note Some operating environments do not support pipes.

PLOTTER
specifies an unbuffered graphics output device.

PRINTER
specifies a printer or printer spool file.

TAPE
specifies a tape drive.

TEMP
creates a temporary file that exists only as long as the filename is assigned. The temporary file can be accessed only through the logical name and is available only while the logical name exists.

Restriction Do not specify a physical pathname. If you do, SAS returns an error.

Tip Files that are manipulated by the TEMP device can have the same attributes and behave identically to DISK files

TERMINAL
specifies the user's personal computer.

UPRINTER
specifies a Universal Printing printer definition name.

Operating environment The FILENAME function also supports operating environment-specific devices. For more information, see the SAS documentation for your operating environment.

'*host-options*'
specifies host-specific details such as file attributes and processing attributes. For more information, see the SAS documentation for your operating environment.

dir-ref
specifies the fileref that was assigned to the directory or partitioned data set in which the external file resides.

Details

FILENAME returns 0 if the operation was successful; ≠0 if it was not successful. The name that is associated with the file or device is called a *fileref* (file reference name). Other system functions that manipulate external files and directories require that the files be identified by fileref rather than by physical filename. The association between a fileref and a physical file lasts only for the duration of the current SAS session or until you change or discontinue the association by using FILENAME. You can deassign filerefs by specifying a null string for the *file-name* argument in FILENAME.

Operating Environment Information
The term *directory* in this description refers to an aggregate grouping of files that are managed by the operating environment. Different operating environments identify these groupings with different names, such as directory, subdirectory, folder, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment.

Under some operating environments, you can also assign filerefs by using system commands. Depending on the operating environment, FILENAME might be unable to change or deassign filerefs that are assigned outside of SAS.

Examples

Example 1: Assigning a Fileref to an External File

This example assigns the fileref MYFILE to an external file. Next, it deassigns the fileref. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf, physical-filename));
%if &rc ne 0 %then
   %put %sysfunc(sysmsg());
%let rc=%sysfunc(filename(filrf));
```

Example 2: Assigning a System-Generated Fileref

This example assigns a system-generated fileref to an external file. The fileref is stored in the variable FNAME. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let rc=%sysfunc(filename(fname, physical-filename));
%if &rc %then
   %put %sysfunc(sysmsg());
%else
   %do;
      more macro statements
   %end;
```

Example 3: Assigning a Fileref to a Pipe File

This example assigns the fileref MYPIPE to a pipe file with the output from the UNIX command LS, which lists the files in the directory /u/myid. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=mypipe;
%let rc=%sysfunc(filename(filrf, %str(ls /u/myid), pipe));
```

See Also

Functions:

- “FEXIST Function” on page 408
- “FILEEXIST Function” on page 410
- “FILEREF Function” on page 414
- “SYSMSG Function” on page 905

FILEREF Function

Verifies whether a fileref has been assigned for the current SAS session.

Category: External Files

See: “FILEREF Function: Windows” in *SAS Companion for Windows*
“FILEREF Function: UNIX” in *SAS Companion for UNIX Environments*
“FILEREF Function: z/OS” in *SAS Companion for z/OS*

Syntax

FILEREF(*fileref*)

Required Argument

fileref
is a character constant, variable, or expression that specifies the fileref to be validated.

Range 1 to 8 characters

Details

A negative return code indicates that the fileref exists but the physical file associated with the fileref does not exist. A positive value indicates that the fileref is not assigned. A value of zero indicates that the fileref and external file both exist.

A fileref can be assigned to an external file by using the FILENAME statement or the FILENAME function.

Windows Specifics
Under some operating environments, filerefs can also be assigned by using system commands. For details, see the SAS documentation for your operating environment.

Examples

Example 1: Verifying That a Fileref Is Assigned

This example tests whether the fileref MYFILE is currently assigned to an external file. A system error message is issued if the fileref is not currently assigned:

```
%if %sysfunc(fileref(myfile))>0 %then
   %put MYFILE is not assigned;
```

Example 2: Verifying That Both a Fileref and a File Exist

This example tests for a zero value to determine whether both the fileref and the file exist:

```
%if %sysfunc(fileref(myfile)) ne 0 %then
   %put %sysfunc(sysmsg());
```

See Also

Functions:

- “FEXIST Function” on page 408
- “FILEEXIST Function” on page 410
- “FILENAME Function” on page 411
- “SYSMSG Function” on page 905

Statements:

- “FILENAME Statement” in *SAS Statements: Reference*

FINANCE Function

Computes financial calculations such as depreciation, maturation, accrued interest, net present value, periodic savings, and internal rates of return.

Category: Financial

Syntax

FINANCE(*string-identifier*, *parm1*, *parm2*,…)

Required Arguments

string-identifier

specifies a character constant, variable, or expression. Valid values for *string-identifier* are listed in the following table.

string-identifier	Description
"ACCRINT" on page 420	computes the accrued interest for a security that pays periodic interest.
"ACCRINTM" on page 420	computes the accrued interest for a security that pays interest at maturity.
"AMORDEGRC" on page 421	computes the depreciation for each accounting period by using a depreciation coefficient.
"AMORLINC" on page 421	computes the depreciation for each accounting period.
"COUPDAYBS" on page 422	computes the number of days from the beginning of the coupon period to the settlement date.
"COUPDAYS" on page 422	computes the number of days in the coupon period that contains the settlement date.
"COUPDAYSNC" on page 423	computes the number of days from the settlement date to the next coupon date.
"COUPNCD" on page 423	computes the next coupon date after the settlement date.
"COUPNUM" on page 423	computes the number of coupons that are payable between the settlement date and the maturity date.
"COUPPCD" on page 424	computes the previous coupon date before the settlement date.

string-identifier	Description
“CUMIPMT” on page 424	computes the cumulative interest that is paid between two periods.
“CUMPRINC” on page 424	computes the cumulative principal that is paid on a loan between two periods.
“DB” on page 425	computes the depreciation of an asset for a specified period by using the fixed-declining balance method.
“DDB” on page 425	computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.
“DISC” on page 426	computes the discount rate for a security.
“DOLLARDE” on page 426	converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.
“DOLLARFR” on page 427	converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.
“DURATION” on page 427	computes the annual duration of a security with periodic interest payments.
“EFFECT” on page 427	computes the effective annual interest rate.
“FV” on page 428	computes the future value of an investment.
“FVSCHEDULE” on page 428	computes the future value of an initial principal after applying a series of compound interest rates.
“INTRATE” on page 428	computes the interest rate for a fully invested security.
“IPMT” on page 429	computes the interest payment for an investment for a given period.
“IRR” on page 429	computes the internal rate of return for a series of cash flows.
“ISPMT” on page 429	calculates the interest paid during a specific period of an investment.
“MDURATION” on page 430	computes the Macaulay modified duration for a security with an assumed face value of $100.

string-identifier	Description
“MIRR” on page 430	computes the internal rate of return where positive and negative cash flows are financed at different rates.
“NOMINAL” on page 431	computes the annual nominal interest rate.
“NPER” on page 431	computes the number of periods for an investment.
“NPV” on page 431	computes the net present value of an investment based on a series of periodic cash flows and a discount rate.
“ODDFPRICE” on page 432	computes the price per $100 face value of a security with an odd first period.
“ODDFYIELD” on page 432	computes the yield of a security with an odd first period.
“ODDLPRICE” on page 433	computes the price per $100 face value of a security with an odd last period.
“ODDLYIELD” on page 433	computes the yield of a security with an odd last period.
“PMT” on page 434	computes the periodic payment for an annuity.
“PPMT” on page 434	computes the payment on the principal for an investment for a given period.
“PRICE” on page 435	computes the price per $100 face value of a security that pays periodic interest.
“PRICEDISC” on page 435	computes the price per $100 face value of a discounted security.
“PRICEMAT” on page 436	computes the price per $100 face value of a security that pays interest at maturity.
“PV” on page 436	computes the present value of an investment.
“RATE” on page 437	computes the interest rate per period of an annuity.
“RECEIVED” on page 437	computes the amount received at maturity for a fully invested security.
“SLN” on page 438	computes the straight-line depreciation of an asset for one period.
“SYD” on page 438	computes the sum-of-years digits depreciation of an asset for a specified period.

string-identifier	Description
"TBILLEQ" on page 438	computes the bond-equivalent yield for a treasury bill.
"TBILLPRICE" on page 439	computes the price per $100 face value for a treasury bill.
"TBILLYIELD" on page 439	computes the yield for a treasury bill.
"VDB" on page 439	computes the depreciation of an asset for a specified or partial period by using a declining balance method.
"XIRR" on page 440	computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.
"XNPV" on page 440	computes the net present value for a schedule of cash flows that is not necessarily periodic.
"YIELD" on page 441	computes the yield on a security that pays periodic interest.
"YIELDDISC" on page 441	computes the annual yield for a discounted security (for example, a treasury bill).
"YIELDMAT" on page 442	computes the annual yield of a security that pays interest at maturity.

parm
specifies a parameter that is associated with each *string-identifier*. The following parameters are available:

basis
is an optional parameter that specifies a character or numeric value that indicates the type of day count basis to use.

Numeric Value	String Value	Day Count Method
0	"30/360"	US (NASD) 30/360
1	"ACTUAL"	Actual/actual
2	"ACT/360"	Actual/360
3	"ACT/365"	Actual/365
4	"EU30/360"	European 30/360

interest-rates
specifies rates that are provided as numeric values and not as percentages.

dates
specifies that all dates in the financial functions are SAS dates.

sign-of-cash-values
for all the arguments, specifies that the cash that you pay out, such as deposits to savings or other withdrawals, is represented by negative numbers. It also specifies that the cash that you receive, such as dividend checks and other deposits, is represented by positive numbers.

Details

ACCRINT

Computes the accrued interest for a security that pays periodic interest.

FINANCE('ACCRINT', *issue*, *first-interest*, *settlement*, *rate*, *par*, *frequency*, <*basis*>);

Arguments

issue
specifies the issue date of the security.

first-interest
specifies the first interest date of the security.

settlement
specifies the settlement date.

rate
specifies the interest rate.

par
specifies the par value of the security. If you omit *par*, SAS uses the value `$1000`.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 1: Computing Accrued Interest: ACCRINT" on page 442

ACCRINTM

Computes the accrued interest for a security that pays interest at maturity.

FINANCE('ACCRINTM', *issue*, *settlement*, *rate*, *par*, <*basis*>);

Arguments

issue
specifies the issue date of the security.

settlement
specifies the settlement date.

rate
specifies the interest rate.

par
specifies the par value of the security. If you omit *par*, SAS uses the value `$1000`.

basis
specifies the optional day count value.

Featured in:
"Example 2: Computing Accrued Interest: ACCRINTM" on page 443

AMORDEGRC

Computes the depreciation for each accounting period by using a depreciation coefficient.

FINANCE('AMORDEGRC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, <*basis*>);

Arguments

cost
specifies the initial cost of the asset.

date-purchased
specifies the date of the purchase of the asset.

first-period
specifies the date of the end of the first period.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

period
specifies the depreciation period.

rate
specifies the rate of depreciation.

basis
specifies the optional day count value.

TIP When the first argument of the FINANCE function is AMORDEGRC and the value of *basis* is 2, the function returns a missing value.

Featured in:
"Example 3: Computing Depreciation: AMORDEGRC" on page 443

AMORLINC

Computes the depreciation for each accounting period.

FINANCE('AMORLINC', *cost*, *date-purchased*, *first-period*, *salvage*, *period*, *rate*, <*basis*>);

Arguments

cost
specifies the initial cost of the asset.

date-purchased
specifies the date of the purchase of the asset.

first-period
specifies the date of the end of the first period.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

period
specifies the depreciation period.

rate
specifies the rate of depreciation.

basis
specifies the optional day count value.

TIP When the first argument of the FINANCE function is AMORLINC and the value of *basis* is 2, the function returns a missing value.

Featured in:
"Example 4: Computing Description: AMORLINC" on page 443

COUPDAYBS

Computes the number of days from the beginning of the coupon period to the settlement date.

FINANCE('COUPDAYBS', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date of the security. The security settlement date is the date after the issue date when the security is traded to the buyer.

maturity
specifies the maturity date of the security. The maturity date is the date the security expires.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the type of day count basis to use.

Featured in:
"Example 5: Computing Description: COUPDAYBS" on page 443

Note: Dates should be entered using the DATE function, or as results of other formulas or functions. For example, use DATE(2011,5,23) for the 23rd day of May 2011. Problems can occur if dates are entered as text.

COUPDAYS

Computes the number of days in the coupon period that contains the settlement date.

FINANCE('COUPDAYS', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 6: Computing Description: COUPDAYS" on page 444

COUPDAYSNC

Computes the number of days from the settlement date to the next coupon date.

FINANCE('COUPDAYSNC', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 7: Computing Description: COUPDAYSNC" on page 444

COUPNCD

Computes the next coupon date after the settlement date.

FINANCE('COUPNCD', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 8: Computing Description: COUPNCD" on page 444

COUPNUM

Computes the number of coupons that are payable between the settlement date and the maturity date.

FINANCE('COUPNUM', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 9: Computing Description: COUPNUM" on page 445

COUPPCD

Computes the previous coupon date before the settlement date.

FINANCE('COUPPCD', *settlement*, *maturity*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 10: Computing Description: COUPPCD" on page 445

CUMIPMT

Computes the cumulative interest paid between two periods.

FINANCE('CUMIPMT', *rate*, *nper*, *pv*, *start-period*, *end-period*, <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

end-period
specifies the last period in the calculation.

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 11: Computing Description: CUMIPMT" on page 445

CUMPRINC

Computes the cumulative principal that is paid on a loan between two periods.

FINANCE('CUMPRINC', *rate*, *nper*, *pv*, *start-period*, *end-period*, <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently.

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

end-period
specifies the last period in the calculation.

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 12: Computing Description: CUMPRINC" on page 445

DB

Computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

FINANCE('DB', *cost*, *salvage*, *life*, *period*, <*month*>);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

month
specifies the number of months (month is an optional numeric argument). If month is omitted, it defaults to a value of 12.

Featured in:
"Example 13: Computing Description: DB" on page 446

DDB

Computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.

FINANCE('DDB', *cost*, *salvage*, *life*, *period*, <*factor*>);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same time units as *life*.

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

Featured in:
"Example 14: Computing Description: DDB" on page 446

DISC

Computes the discount rate for a security.

FINANCE('DISC', *settlement*, *maturity*, *pr*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

pr
specifies the price of security per $100 face value.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Featured in:
"Example 15: Computing Description: DISC" on page 446

DOLLARDE

Converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

FINANCE('DOLLARDE', *fractionaldollar*, *fraction*);

Arguments

fractionaldollar
specifies the number expressed as a fraction.

fraction
specifies the integer to use in the denominator of a fraction.

Featured in:
"Example 16: Computing Description: DOLLARDE" on page 447

DOLLARFR

Converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

FINANCE('DOLLARFR', *decimaldollar*, *fraction*);

Arguments

decimaldollar
specifies a decimal number.

fraction
specifies the integer to use in the denominator of a fraction.

Featured in:
"Example 17: Computing Description: DOLLARFR" on page 447

DURATION

Computes the annual duration of a security with periodic interest payments.

FINANCE('DURATION', *settlement*, *maturity*, *coupon*, *yld*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

coupon
specifies the annual coupon rate of the security.

yld
specifies the annual yield of the security.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 18: Computing Description: DURATION" on page 447

EFFECT

Computes the effective annual interest rate.

FINANCE('EFFECT', *nominalrate*, *npery*);

Arguments

nominalrate
specifies the nominal interest rate.

npery
specifies the number of compounding periods per year.

Featured in:
"Example 19: Computing Description: EFFECT" on page 447

FV

Computes the future value of an investment.

FINANCE('FV', *rate*, *nper*, <*pmt*> , <*pv*> , <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

pmt
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *pmt* contains principal and interest but no fees and taxes. If *pmt* is omitted, you must include the *pv* argument.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *pmt* argument.

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 20: Computing Description: FV" on page 448

FVSCHEDULE

Computes the future value of the initial principal after applying a series of compound interest rates.

FINANCE('FVSCHEDULE', *principal*, *schedule1*, *schedule2*...);

Arguments

principal
specifies the present value.

schedule
specifies the sequence of interest rates to apply.

Featured in:
"Example 21: Computing Description: FVSCHEDULE" on page 448

INTRATE

Computes the interest rate for a fully invested security.

FINANCE('INTRATE', *settlement*, *maturity*, *investment*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

investment
specifies the amount that is invested in the security.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Featured in:
"Example 22: Computing Description: INTRATE" on page 448

IPMT

Computes the interest payment for an investment for a specified period.

FINANCE('IPMT', *rate*, *period*, *nper*, *pv*, <*fv*> , <*type*>);

Arguments

rate
specifies the interest rate.

period
specifies the period for which you want to calculate the depreciation. *Period* must use the same units as *life*.

nper
specifies the total number of payment periods.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 23: Computing Description: IPMT" on page 448

IRR

Computes the internal rate of return for a series of cash flows.

FINANCE('IRR', *value1*, *value2*, …, *value_n*);

Arguments

value
specifies a list of numeric arguments that contain numbers for which you want to calculate the internal rate of return.

Featured in:
"Example 24: Computing Description: IRR" on page 449

ISPMT

Calculates the interest paid during a specific period of an investment.

FINANCE ('ISPMT', *interest-rate*, *period*, *number-payments*, *PV*);

Arguments

interest-rate
is the interest rate for the investment.

period
is the period to calculate the interest rate. *Period* must be a value between 1 and *number-payments*.

number-payments
is the number of payments for the annuity.

PV
is the loan amount or present value of the payments.

Featured in:
"Example 25: Computing Description: ISPMT" on page 449

MDURATION

Computes the Macaulay modified duration for a security with an assumed face value of $100.

FINANCE('MDURATION', *settlement*, *maturity*, *coupon*, *yld*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

coupon
specifies the annual coupon rate of the security.

yld
specifies the annual yield of the security.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 26: Computing Description: MDURATION" on page 449

MIRR

Computes the internal rate of return where positive and negative cash flows are financed at different rates.

FINANCE('MIRR', *value1*, …, *value_n*, *financerate*, *reinvestrate*);

Arguments

values
specifies a list of numeric arguments that contain numbers. These numbers represent a series of payments (negative values) and income (positive values) that occur at regular periods. *Values* must contain at least one positive value and one negative value to calculate the modified internal rate of return.

financerate
specifies the interest rate that you pay on the money that is used in the cash flows.

reinvestrate
specifies the interest rate that you receive on the cash flows as you reinvest them.

Featured in:
"Example 27: Computing Description: MIRR" on page 449

NOMINAL

Computes the annual nominal interest rates.

FINANCE('NOMINAL', *effectrate*, *npery*);

Arguments

effectrate
specifies the effective interest rate.

npery
specifies the number of compounding periods per year.

Featured in:
"Example 28: Computing Description: NOMINAL" on page 450

NPER

Computes the number of periods for an investment.

FINANCE('NPER', *rate*, *pmt*, *pv*, <*fv*> , <*type*>);

Arguments

rate
specifies the interest rate.

pmt
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *pmt* contains principal and interest but no other fees or taxes. If *pmt* is omitted, you must include the *pv* argument.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *pmt* argument.

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 29: Computing Description: NPER" on page 450

NPV

Computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

FINANCE('NPV', *rate*, *value−1* <,…value-n>);

Arguments

rate
specifies the interest rate.

value
represents the sequence of the cash flows.

Featured in:
"Example 30: Computing Description: NPV" on page 450

ODDFPRICE

Computes the price of a security per $100 face value with an odd first period.

FINANCE('ODDFPRICE', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *yld*, *redemption*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

first-coupon
specifies the first coupon date of the security.

rate
specifies the interest rate.

yld
specifies the annual yield of the security.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 31: Computing Description: ODDFPRICE" on page 450

ODDFYIELD

Computes the yield of a security with an odd first period.

FINANCE('ODDFYIELD', *settlement*, *maturity*, *issue*, *first-coupon*, *rate*, *pr*, *redemption*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

first-coupon
specifies the first coupon date of the security.

rate
specifies the interest rate.

pr
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 32: Computing Description: ODDFYIELD" on page 451

ODDLPRICE

Computes the price of a security per $100 face value with an odd last period.

FINANCE('ODDLPRICE', *settlement*, *maturity*, *last_interest*, *rate*, *yld*, *redemption*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

last_interest
specifies the last coupon date of the security.

rate
specifies the interest rate.

yld
specifies the annual yield of the security.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 33: Computing Description: ODDLPRICE" on page 451

ODDLYIELD

Computes the yield of a security with an odd last period.

FINANCE('ODDLYIELD', *settlement*, *maturity*, *last_interest*, *rate*, *pr*, *redemption*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

last_interest
specifies the last coupon date of the security.

rate
specifies the interest rate.

pr
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 34: Computing Description: ODDLYIELD" on page 452

PMT

Computes the periodic payment of an annuity.

FINANCE('PMT', *rate*, *nper*, *pv*, <*fv*> , <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the number of payment periods.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 35: Computing Description: PMT" on page 452

PPMT

Computes the payment on the principal for an investment for a specified period.

FINANCE('PPMT', *rate*, *per*, *nper*, *pv*, <*fv*> , <*type*>);

Arguments

rate
specifies the interest rate.

per
specifies the period.

Range: 1–nper

nper
specifies the number of payment periods.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 36: Computing Description: PPMT" on page 452

PRICE

Computes the price of a security per $100 face value that pays periodic interest.

FINANCE('PRICE', *settlement*, *maturity*, *rate*, *yld*, *redemption*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

rate
specifies the interest rate.

yld
specifies the annual yield of the security.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 37: Computing Description: PRICE" on page 452

PRICEDISC

Computes the price of a discounted security per $100 face value.

FINANCE('PRICEDISC', *settlement*, *maturity*, *discount*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

discount
specifies the discount rate of the security.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Featured in:
"Example 38: Computing Description: PRICEDISC" on page 453

PRICEMAT

Computes the price of a security per $100 face value that pays interest at maturity.

FINANCE('PRICEMAT', *settlement*, *maturity*, *issue*, *rate*, *yld*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

rate
specifies the interest rate.

yld
specifies the annual yield of the security.

basis
specifies the optional day count value.

Featured in:
"Example 39: Computing Description: PRICEMAT" on page 453

PV

Computes the present value of an investment.

FINANCE('PV', *rate*, *nper*, *pmt*, <*fv*> , <*type*>);

Arguments

rate
specifies the interest rate.

nper
specifies the total number of payment periods.

pmt
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *pmt* contains principal and interest but no other fees or taxes.

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 40: Computing Description: PV" on page 453

RATE

Computes the interest rate per period of an annuity.

FINANCE('RATE', *nper*, *pmt*, *pv*, <*fv*> , <*type*>);

Arguments

nper
specifies the total number of payment periods.

pmt
specifies the payment that is made each period; the payment cannot change over the life of the annuity. Typically, *pmt* contains principal and interest but no other fees or taxes. If *pmt* is omitted, you must include the *pv* argument.

pv
specifies the present value or the lump-sum amount that a series of future payments is worth currently. If *pv* is omitted, it is assumed to be 0 (zero), and you must include the *fv* argument.

fv
specifies the future value or a cash balance that you want to attain after the last payment is made. If *fv* is omitted, it is assumed to be 0 (for example, the future value of a loan is 0).

type
specifies the number 0 or 1 and indicates when payments are due. If *type* is omitted, it is assumed to be 0.

Featured in:
"Example 41: Computing Description: RATE" on page 454

RECEIVED

Computes the amount that is received at maturity for a fully invested security.

FINANCE('RECEIVED', *settlement*, *maturity*, *investment*, *discount*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

investment
specifies the amount that is invested in the security.

discount
specifies the discount rate of the security.

basis
specifies the optional day count value.

Featured in:
"Example 42: Computing Description: RECEIVED" on page 454

SLN

Computes the straight-line depreciation of an asset for one period.

FINANCE('SLN', *cost*, *salvage*, *life*);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of an asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

Featured in:
"Example 43: Computing Description: SLN" on page 454

SYD

Computes the sum-of-years digits depreciation of an asset for a specified period.

FINANCE('SYD', *cost*, *salvage*, *life*, *period*);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

period
specifies a period in the same time units that are used for the argument *life*.

Featured in:
"Example 44: Computing Description: SYD" on page 454

TBILLEQ

Computes the bond-equivalent yield for a treasury bill.

FINANCE('TBILLEQ', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

discount
specifies the discount rate of the security.

Featured in:
"Example 45: Computing Description: TBILLEQ" on page 455

TBILLPRICE

Computes the price of a treasury bill per $100 face value.

FINANCE('TBILLPRICE', *settlement*, *maturity*, *discount*);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

discount
specifies the discount rate of the security.

Featured in:
"Example 46: Computing Description: TBILLPRICE" on page 455

TBILLYIELD

Computes the yield for a treasury bill.

FINANCE('TBILLYIELD', *settlement*, *maturity*, *pr*);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

pr
specifies the price of the security per $100 face value.

Featured in:
"Example 47: Computing Description: TBILLYIELD" on page 455

VDB

Computes the depreciation of an asset for a specified or partial period by using a declining balance method.

FINANCE('VDB', *cost*, *salvage*, *life*, *start-period*, *end-period*, <*factor*> , <*noswitch*>);

Arguments

cost
specifies the initial cost of the asset.

salvage
specifies the value at the end of the depreciation (also called the salvage value of the asset).

life
specifies the number of periods over which the asset is depreciated (also called the useful life of the asset).

start-period
specifies the first period in the calculation. Payment periods are numbered beginning with 1.

end-period
specifies the last period in the calculation.

factor
specifies the rate at which the balance declines. If *factor* is omitted, it is assumed to be 2 (the double-declining balance method).

noswitch
specifies a logical value that determines whether to switch to straight-line depreciation when the depreciation is greater than the declining balance calculation. If *noswitch* is omitted, it is assumed to be 1.

Featured in:
"Example 48: Computing Description: VDB" on page 455

XIRR

Computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

FINANCE('XIRR', *values*, *dates*, <*guess*>);

Arguments

values
specifies a series of cash flows that corresponds to a schedule of payments in dates. The first payment is optional and corresponds to a cost or payment that occurs at the beginning of the investment. If the first value is a cost or payment, it must be a negative value. All succeeding payments are discounted based on a 365-day year. The series of values must contain at least one positive value and one negative value.

dates
specifies a schedule of payment dates that corresponds to the cash flow payments. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they can occur in any order.

guess
specifies an optional number that you guess is close to the result of XIRR.

Featured in:
"Example 49: Computing Description: XIRR" on page 456

XNPV

Computes the net present value for a schedule of cash flows that is not necessarily periodic.

FINANCE('XNPV', *rate*, *values*, *dates*);

Arguments

rate
specifies the interest rate.

values
specifies a series of cash flows that corresponds to a schedule of payments in dates. The first payment is optional and corresponds to a cost or payment that occurs at the beginning of the investment. If the first value is a cost or payment, it must be a negative value. All succeeding payments are discounted based on a 365-day year. The series of values must contain at least one positive value and one negative value.

dates
specifies a schedule of payment dates that corresponds to the cash flow payments. The first payment date indicates the beginning of the schedule of payments. All other dates must be later than this date, but they can occur in any order.

Featured in:
"Example 50: Computing Description: XNPV" on page 456

YIELD

Computes the yield on a security that pays periodic interest.

FINANCE('YIELD', *settlement*, *maturity*, *rate*, *pr*, *redemption*, *frequency*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

rate
specifies the interest rate.

pr
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

frequency
specifies the number of coupon payments per year. For annual payments, *frequency* = 1; for semiannual payments, *frequency* = 2; for quarterly payments, *frequency* = 4.

basis
specifies the optional day count value.

Featured in:
"Example 51: Computing Description: YIELD" on page 456

YIELDDISC

Computes the annual yield for a discounted security (for example, a treasury bill).

FINANCE('YIELDDISC', *settlement*, *maturity*, *rate*, *pr*, *redemption*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

rate
specifies the interest rate.

pr
specifies the price of the security per $100 face value.

redemption
specifies the amount to be received at maturity.

basis
specifies the optional day count value.

Featured in:
"Example 52: Computing Description: YIELDDISC" on page 456

YIELDMAT

Computes the annual yield of a security that pays interest at maturity.

FINANCE('YIELDMAT', *settlement*, *maturity*, *issue*, *rate*, *pr*, <*basis*>);

Arguments

settlement
specifies the settlement date.

maturity
specifies the maturity date.

issue
specifies the issue date of the security.

rate
specifies the interest rate.

pr
specifies the price of the security per $100 face value.

basis
specifies the optional day count value.

Featured in:
"Example 53: Computing Description: YIELDMAT" on page 457

Examples

Example 1: Computing Accrued Interest: ACCRINT

The following example computes the accrued interest for a security that pays periodic interest.

```
data _null_;
   issue = mdy(2,27,1996);
   firstinterest = mdy(8,31,1998);
   settlement = mdy(5,1,1998);
   rate = 0.1;
   par = 1000;
   frequency = 2;
   basis = 1;
   r = finance('accrint', issue, firstinterest,
               settlement, rate, par, frequency, basis);
   put r=;
run;
```

The value of *r* that is returned is 217.39728.

Example 2: Computing Accrued Interest: ACCRINTM

The following example computes the accrued interest for a security that pays interest at maturity.

```
data _null_;
   issue = mdy(2,28,1998);
   maturity = mdy(8,31,1998);
   rate = 0.1;
   par = 1000;
   basis = 0;
   r = finance('accrintm', issue, maturity, rate, par, basis);
   put r=;
run;
```

The value of *r* that is returned is 50.555555556.

Example 3: Computing Depreciation: AMORDEGRC

The following example computes the depreciation for each accounting period by using a depreciation coefficient.

```
data _null_;
   cost = 2400;
   datepurchased = mdy(8,19,2008);
   firstperiod = mdy(12,31,2008);
   salvage = 300;
   period = 1;
   rate = 0.15;
   basis = 1;
   r = finance('amordegrc', cost, datepurchased,
               firstperiod, salvage, period, rate, basis);
   put r=;
run;
```

The value of *r* that is returned is 776.

Example 4: Computing Description: AMORLINC

The following example computes the depreciation for each accounting period.

```
data _null_;
   cost = 2400;
   dp = mdy(9,30,1998);
   fp = mdy(12,31,1998);
   salvage = 245;
   period = 0;
   rate = 0.115;
   basis = 0;
   r = finance('amorlinc', cost, dp, fp, salvage,
                period, rate, basis);
   put r = ;
run;
```

The value of *r* that is returned is 69.

Example 5: Computing Description: COUPDAYBS

The following example computes the number of days from the beginning of the coupon period to the settlement date.

```
data _null_;
```

```
   settlement = mdy(12,30,1994);
   maturity = mdy(11,29,1997);
   frequency = 4;
   basis = 2;
   r = finance('coupdaybs', settlement, maturity, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 31.

Example 6: Computing Description: COUPDAYS

The following example computes the number of days in the coupon period that contains the settlement date.

```
data _null_;
   settlement = mdy(1,25,2007);
   maturity = mdy(11,15,2008);
   frequency = 2;
   basis = 1;
   r = finance('coupdays', settlement, maturity, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 181.

Example 7: Computing Description: COUPDAYSNC

The following example computes the number of days from the settlement date to the next coupon date.

```
data _null_;
   settlement = mdy(1,25,2007);
   maturity = mdy(11,15,2008);
   frequency = 2;
   basis = 1;
   r = finance('coupdaysnc', settlement, maturity, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 110.

Example 8: Computing Description: COUPNCD

The following example computes the next coupon date after the settlement date.

```
data _null_;
   settlement = mdy(1,25,2007);
   maturity = mdy(11,15,2008);
   frequency = 2;
   basis = 1;
   r = finance('coupncd', settlement, maturity, frequency, basis);
   put r = date7.;
run;
```

The value of *r* that is returned is 15MAY07.

Note: *r* is a numeric SAS value and can be printed using the DATE7 format.

Example 9: Computing Description: COUPNUM

The following example computes the number of coupons that are payable between the settlement date and the maturity date.

```
data _null_;
   settlement = mdy(1,25,2007);
   maturity = mdy(11,15,2008);
   frequency = 2;
   basis = 1;
   r = finance('coupnum', settlement, maturity, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 4.

Example 10: Computing Description: COUPPCD

The following example computes the previous coupon date before the settlement date.

```
data _null_;
   settlement = mdy(1,25,2007);
   maturity = mdy(11,15,2008);
   frequency = 2;
   basis = 1;
   r = finance('couppcd', settlement, maturity, frequency, basis);
   put  settlement;
   put  maturity;
   put r date7.;
run;
```

The value of *r* that is returned is 11/15/2006.

Example 11: Computing Description: CUMIPMT

The following example computes the cumulative interest that is paid between two periods.

```
data _null_;
   rate = 0.09;
   nper = 30;
   pv = 125000;
   startperiod = 13;
   endperiod = 24;
   type = 0;
   r = finance('cumipmt', rate, nper, pv,
               startperiod, endperiod, type);
   put r = ;
run;
```

The value of *r* that is returned is −94054.82033.

Example 12: Computing Description: CUMPRINC

The following example computes the cumulative principal that is paid on a loan between two periods.

```
data _null_;
   rate = 0.09;
   nper = 30;
   pv = 125000;
   startperiod = 13;
```

```
   endperiod = 24;
   type = 0;
   r = finance('cumprinc', rate, nper, pv,
               startperiod, endperiod, type);
   put r = ;
run;
```

The value of *r* that is returned is −51949.70676.

Example 13: Computing Description: DB

The following example computes the depreciation of an asset for a specified period by using the fixed-declining balance method.

```
data _null_;
   cost = 1000000;
   salvage = 100000;
   life = 6;
   period = 2;
   month = 7;
   r = finance('db', cost, salvage, life, period, month);
   put r = ;
 run;
```

The value of *r* that is returned is 259639.41667.

Example 14: Computing Description: DDB

The following example computes the depreciation of an asset for a specified period by using the double-declining balance method or some other method that you specify.

```
data _null_;
   cost = 2400;
   salvage = 300;
   life = 10*365;
   period = 1;
   factor = .;
   r = finance('ddb', cost, salvage, life, period, factor);
   put r = ;
run;
```

The value of *r* that is returned is 1.3150684932.

Example 15: Computing Description: DISC

The following example computes the discount rate for a security.

```
data _null_;
   settlement = mdy(1,25,2007);
   maturity = mdy(6,15,2007);
   pr = 97.975;
   redemption = 100;
   basis = 1;
   r = finance('disc', settlement, maturity, pr, redemption, basis);
   put r = ;
run;
```

The value of *r* that is returned is 0.052420213.

Example 16: Computing Description: DOLLARDE

The following example converts a dollar price, expressed as a fraction, to a dollar price, expressed as a decimal number.

```
data _null_;
   fractionaldollar = 1.125;
   fraction = 16;
   r = finance('dollarde', fractionaldollar, fraction);
   put r = ;
run;
```

The value of *r* that is returned is 1.78125.

Example 17: Computing Description: DOLLARFR

The following example converts a dollar price, expressed as a decimal number, to a dollar price, expressed as a fraction.

```
data _null_;
   decimaldollar = 1.125;
   fraction = 16;
   r = finance('dollarfr', decimaldollar, fraction);
   put r = ;
run;
```

The value of *r* that is returned is 1.02. In fraction form, the value of *r* is read as $1\frac{2}{16}$.

Example 18: Computing Description: DURATION

The following example computes the annual duration of a security with periodic interest payments.

```
data _null_;
   settlement = mdy(1,1,2008);
   maturity = mdy(1,1,2016);
   couponrate = 0.08;
   yield = 0.09;
   frequency = 2;
   basis = 1;
   r = finance('duration', settlement,
          maturity, couponrate, yield, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 5.993775.

Example 19: Computing Description: EFFECT

The following example computes the effective annual interest rate.

```
data _null_;
   nominalrate = 0.0525;
   npery = 4;
   r = finance('effect', nominalrate, npery);
   put r = ;
run;
```

The value of *r* that is returned is 0.053543.

Example 20: Computing Description: FV

The following example computes the future value of an investment.

```
data _null_;
   rate = 0.06/12;
   nper = 10;
   pmt = -200;
   pv = -500;
   type = 1;
   r = finance('fv', rate, nper, pmt, pv, type);
   put r = ;
run;
```

The value of *r* that is returned is 2581.4033741.

Example 21: Computing Description: FVSCHEDULE

The following example computes the future value of the initial principal after applying a series of compound interest rates.

```
data _null_;
   principal = 1;
   r1 = 0.09;
   r2 = 0.11;
   r3 = 0.1;
   r = finance('fvschedule', principal, r1, r2, r3);
   put r = ;
run;
```

The value of *r* that is returned is 1.33089.

Example 22: Computing Description: INTRATE

The following example computes the interest rate for a fully invested security.

```
data _null_;
   settlement = mdy(2,15,2008);
   maturity = mdy(5,15,2008);
   investment = 1000000;
   redemption = 1014420;
   basis = 2;
   r = finance('intrate', settlement, maturity,
        investment, redemption, basis);
   put r = ;
run;
```

The value of *r* that is returned is 0.05768

Example 23: Computing Description: IPMT

The following example computes the interest payment for an investment for a specified period.

```
data _null_;
   rate = 0.1/12;
   per = 2;
   nper = 3;
   pv = 100;
   fv = .;
   type = .;
   r = finance('ipmt', rate, per, nper, pv, fv, type);
```

```
   put r = ;
run;
```

The value of *r* that is returned is −0.557857564.

Example 24: Computing Description: IRR

The following example computes the internal rate of return for a series of cash flows.

```
data _null_;
   v1 = -70000;
   v2 = 12000;
   v3 = 15000;
   v4 = 18000;
   v5 = 21000;
   v6 = 26000;
   r = finance('irr', v1, v2, v3, v4, v5, v6);
   put r = ;
run;
```

The value of *r* that is returned is 0.086630948.

Example 25: Computing Description: ISPMT

The following example computes the interest payment for a $5,000 investment that earns 7.5% annually for two years. The interest payment is calculated for the 8th month.

```
data ispmt;
   interest=finance('ispmt', 0.075/12, 8, 2*12, 5000);
   put interest=;
run;
```

The value that is returned is –20.83333333.

Example 26: Computing Description: MDURATION

The following example computes the Macaulay modified duration for a security with an assumed face value of $100.

```
data _null_;
   settlement = mdy(1,1,2008);
   maturity = mdy(1,1,2016);
   couponrate = 0.08;
   yield = 0.09;
   frequency = 2;
   basis = 1;
   r = finance('mduration', settlement, maturity,
     couponrate, yield, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 5.7356698139.

Example 27: Computing Description: MIRR

The following example computes the internal rate of return where positive and negative cash flows are financed at different rates.

```
data _null_;
   v1 = -1000;
   v2 = 3000;
   v3 = 4000;
```

```
   v4 = 5000;
   financerate = 0.08;
   reinvestrate = 0.10;
   r = finance('mirr', v1, v2, v3, v4, financerate, reinvestrate);
   put r = ;
run;
```

The value of *r* that is returned is 1.3531420172.

Example 28: Computing Description: NOMINAL

The following example computes the annual nominal interest rate.

```
data _null_;
   effectrate = 0.08;
   npery = 4;
   r = finance('nominal', effectrate, npery);
   put r = ;
run;
```

The value of *r* that is returned is 0.0777061876.

Example 29: Computing Description: NPER

The following example computes the number of periods for an investment.

```
data _null_;
   rate = 0.08;
   pmt = 200;
   pv = 1000;
   fv = 0;
   type = 0;
   r = finance('nper', rate, pmt, pv, fv, type);
   put r = ;
run;
```

The value of *r* that is returned is −4.371981351.

Example 30: Computing Description: NPV

The following example computes the net present value of an investment based on a series of periodic cash flows and a discount rate.

```
data _null_;
   rate = 0.08;
   v1 = 200;
   v2 = 1000;
   v3 = 0.;
   r = finance('npv', rate, v1, v2, v3);
   put r = ;
run;
```

The value of *r* that is returned is 1042.5240055.

Example 31: Computing Description: ODDFPRICE

The following example computes the price of a security per $100 face value with an odd first period.

```
data _null_;
   settlement = mdy(1,15,93);
   maturity = mdy(1,1,98);
```

```
   issue = mdy(1,1,93);
   firstcoupon = mdy(7,1,94);
   rate = 0.07;
   yld = 0.06;
   redemption = 100;
   frequency = 2;
   basis = 0;
   r = finance('oddfprice',
    settlement, maturity, issue, firstcoupon, rate, yld, redemption,
    frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 103.94103984.

Example 32: Computing Description: ODDFYIELD

The following example computes the interest of a yield with an odd first period.

```
data _null_;
   settlement = mdy(1,15,93);
   maturity = mdy(1,1,98);
   issue = mdy(1,1,93);
   firstcoupon = mdy(7,1,94);
   rate = 0.07;
   pr = 103.94103984;
   redemption = 100;
   frequency = 2;
   basis = 0;
   r = finance('oddfyield',
    settlement, maturity, issue, firstcoupon, rate, pr, redemption,
    frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 0.06.

Example 33: Computing Description: ODDLPRICE

The following example computes the price of a security per $100 face value with an odd last period.

```
data _null_;
   settlement = mdy(2,7,2008);
   maturity = mdy(6,15,2008);
   lastinterest = mdy(10,15,2007);
   rate = 0.0375;
   yield = 0.0405;
   redemption = 100;
   frequency = 2;
   basis = 0;
   r = finance('oddlprice', settlement, maturity, lastinterest,
   rate, yield, redemption, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 99.878286015.

Example 34: Computing Description: ODDLYIELD

The following example computes the yield of a security with an odd last period.

```
data _null_;
   settlement = mdy(2,7,2008);
   maturity = mdy(6,15,2008);
   lastinterest = mdy(10,15,2007);
   rate = 0.0375;
   pr = 99.878286015;
   redemption = 100;
   frequency = 2;
   basis = 0;
   r = finance('oddlyield', settlement, maturity, lastinterest,
   rate, pr, redemption, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 0.0405.

Example 35: Computing Description: PMT

The following example computes the periodic payment for an annuity.

```
data _null_;
   rate = 0.08;
   nper = 5;
   pv = 91;
   fv = 3;
   type = 0;
   r = finance('pmt', rate, nper, pv, fv, type);
   put r = ;
run;
```

The value of *r* that is returned is -23.30290673.

Example 36: Computing Description: PPMT

The following example computes the payment on the principal for an investment for a specified period.

```
data _null_;
   rate = 0.08;
   per = 10;
   nper = 10;
   pv = 200000;
   fv = 0;
   type = 0;
   r = finance('ppmt', rate, per, nper, pv, fv, type);
   put r = ;
run;
```

The value of *r* that is returned is −27598.05346.

Example 37: Computing Description: PRICE

The following example computes the price of a security per $100 face value that pays periodic interest.

```
data _null_;
   settlement = mdy(2,15,2008);
   maturity = mdy(11,15,2017);
```

```
   rate = 0.0575;
   yield = 0.065;
   redemption = 100;
   frequency = 2;
   basis = 0;
   r = finance('price', settlement, maturity, rate, yield, redemption,
     frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 94.634361621.

Example 38: Computing Description: PRICEDISC

The following example computes the price of a discounted security per $100 face value.

```
data _null_;
   settlement = mdy(2,15,2008);
   maturity = mdy(11,15,2017);
   discount = 0.0525;
   redemption = 100;
   basis = 0;
   r = finance('pricedisc', settlement, maturity, discount, redemption, basis);
   put r = ;
run;
```

The value of *r* that is returned is 48.8125.

Example 39: Computing Description: PRICEMAT

The following example computes the price of a security per $100 face value that pays interest at maturity.

```
data _null_;
   settlement = mdy(2,15,2008);
   maturity = mdy(4,13,2008);
   issue = mdy(11,11,2007);
   rate = 0.061;
   yield = 0.061;
   basis = 0;
   r = finance('pricemat', settlement, maturity, issue, rate, yield, basis);
   put r = ;
run;
```

The value of *r* that is returned is 99.98449888.

Example 40: Computing Description: PV

The following example computes the present value of an investment.

```
data _null_;
   rate = 0.05;
   nper = 10;
   pmt = 1000;
   fv = 200;
   type = 0;
   r = finance('pv', rate, nper, pmt, fv, type);
   put r = ;
run;
```

The value of *r* that is returned is −7844.51758.

Example 41: Computing Description: RATE

The following example computes the interest rate per period of an annuity.

```
data _null_;
   nper = 4;
   pmt = -2481;
   pv = 8000;
   r = finance('rate', nper, pmt, pv);
   put r = ;
run;
```

The value of *r* that is returned is 0.0921476841.

Example 42: Computing Description: RECEIVED

The following example computes the amount that is received at maturity for a fully invested security.

```
data _null_;
   settlement = mdy(2,15,2008);
   maturity = mdy(5,15,2008);
   investment = 1000000;
   discount = 0.0575;
   basis = 2;
   r = finance('received', settlement, maturity, investment, discount, basis);
   put r = ;
run;
```

The value of *r* that is returned is 1014584.6544.

Example 43: Computing Description: SLN

The following example computes the straight-line depreciation of an asset for one period.

```
data _null_;
   cost = 2000;
   salvage = 200;
   life = 11;
   r = finance('sln', cost, salvage, life);
   put r = ;
run;
```

The value of *r* that is returned is 163.63636364.

Example 44: Computing Description: SYD

The following example computes the sum-of-years digits depreciation of an asset for a specified period.

```
data _null_;
   cost = 2000;
   salvage = 200;
   life = 11;
   per = 1;
   r = finance('syd', cost, salvage, life, per);
   put r = ;
run;
```

The value of *r* that is returned is 300.

Example 45: Computing Description: TBILLEQ

The following example computes the bond-equivalent yield for a treasury bill.

```
data _null_;
   settlement = mdy(3,31,2008);
   maturity = mdy(6,1,2008);
   discount = 0.0914;
   r = finance('tbilleq', settlement, maturity, discount);
   put r = ;
run;
```

The value of *r* that is returned is 0.0941514936.

Example 46: Computing Description: TBILLPRICE

The following example computes the price of a treasury bill per $100 face value.

```
data _null_;
   settlement = mdy(3,31,2008);
   maturity = mdy(6,1,2008);
   discount = 0.09;
   r = finance('tbillprice', settlement, maturity, discount);
   put r = ;
run;
```

The value of *r* that is returned is 98.45.

Example 47: Computing Description: TBILLYIELD

The following example computes the yield for a treasury bill.

```
data _null_;
   settlement = mdy(3,31,2008);
   maturity = mdy(6,1,2008);
   pr = 98;
   r = finance('tbillyield', settlement, maturity, pr);
   put r = ;
run;
```

The value of *r* that is returned is 0.1184990125.

Example 48: Computing Description: VDB

The following example computes the depreciation of an asset for a specified or partial period by using a declining balance method.

```
data _null_;
   cost = 2400;
   salvage = 300;
   life = 10;
   startperiod = 0;
   endperiod = 1;
   factor = 1.5;
   r = finance('vdb', cost, salvage, life, startperiod, endperiod, factor);
   put r = ;
run;
```

The value of *r* that is returned is 360.

Example 49: Computing Description: XIRR

The following example computes the internal rate of return for a schedule of cash flows that is not necessarily periodic.

```
data _null_;
   v1 = -10000; d1 = mdy(1,1,2008);
   v2 = 2750; d2 = mdy(3,1,2008);
   v3 = 4250; d3 = mdy(10,30,2008);
   v4 = 3250; d4 = mdy(2,15,2009);
   v5 = 2750; d5 = mdy(4,1,2009);
   r = finance('xirr', v1, v2, v3, v4, v5, d1, d2, d3, d4, d5, 0.1);
   put r = ;
run;
```

The value of *r* that is returned is 0.3733625335.

Example 50: Computing Description: XNPV

The following example computes the net present value for a schedule of cash flows that is not necessarily periodic.

```
data _null_;
   r = 0.09;
   v1 = -10000; d1 = mdy(1,1,2008);
   v2 = 2750; d2 = mdy(3,1,2008);
   v3 = 4250; d3 = mdy(10,30,2008);
   v4 = 3250; d4 = mdy(2,15,2009);
   v5 = 2750; d5 = mdy(4,1,2009);
   r = finance('xnpv', r, v1, v2, v3, v4, v5, d1, d2, d3, d4, d5);
   put r = ;
run;
```

The value of *r* that is returned is 2086.647602.

Example 51: Computing Description: YIELD

The following example computes the yield on a security that pays periodic interest.

```
data _null_;
   settlement = mdy(2,15,2008);
   maturity = mdy(11,15,2016);
   rate = 0.0575;
   pr = 95.04287;
   redemption = 100;
   frequency = 2;
   basis = 0;
   r = finance('yield', settlement, maturity, rate, pr, redemption, frequency, basis);
   put r = ;
run;
```

The value of *r* that is returned is 0.0650000069.

Example 52: Computing Description: YIELDDISC

The following example computes the annual yield for a discounted security (for example, a treasury bill).

```
data _null_;
   settlement = mdy(2,15,2008);
   maturity = mdy(11,15,2016);
   pr = 95.04287;
```

```
   redemption = 100;
   basis = 0;
   r = finance('yielddisc', settlement, maturity, pr, redemption, basis);
   put r = ;
run;
```

The value of *r* that is returned is 0.0059607748.

Example 53: Computing Description: YIELDMAT

The following example computes the annual yield of a security that pays interest at maturity.

```
data _null_;
   settlement = mdy(3,15,2008);
   maturity = mdy(11,3,2008);
   issue = mdy(11,8,2007);
   rate = 0.0625;
   pr = 100.0123;
   basis = 0;
   r = finance('yieldmat', settlement, maturity, issue, rate, pr, basis);
   put r = ;
run;
```

The value of *r* that is returned is 0.0609543337.

FIND Function

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

Category: Character

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

Tip: Use the “KINDEX Function” in *SAS National Language Support (NLS): Reference Guide* instead to write encoding independent code.

Syntax

FIND(*string*,*substring*<,*modifiers*> <,*startpos*>)

FIND(*string*,*substring*<,*startpos*> <,*modifiers*>)

Required Arguments

string

specifies a character constant, variable, or expression that will be searched for substrings.

Tip Enclose a literal string of characters in quotation marks.

substring

is a character constant, variable, or expression that specifies the substring of characters to search for in *string*.

Tip Enclose a literal string of characters in quotation marks.

Optional Arguments

modifiers
is a character constant, variable, or expression that specifies one or more modifiers. The following *modifiers* can be in uppercase or lowercase:

i
ignores character case during the search. If this modifier is not specified, FIND only searches for character substrings with the same case as the characters in *substring*.

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.

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 with an integer value that specifies the position at which the search should start and the direction of the search.

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 SAS statements produce these results.

SAS Statement	Result
`whereisshe=find('She sells seashells? Yes, she does.','she ');` `put whereisshe;`	27

`variable1='She sells seashells? Yes, she does.';`
`variable2='she ';`
`variable3='i';`
`whereisshe_i=find(variable1,variable2,variable3);`
`put 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);`
`put whereisshe_t;`
	14

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

`xyz='She sells seashells? Yes, she does.';`
`startposexp=1-23;`
`whereisShe_ineg22=find(xyz,'She','i',startposexp);`
`put whereisShe_ineg22;`
	14

See Also

Functions:

- “COUNT Function” on page 338
- “FINDC Function” on page 459
- “INDEX Function” on page 543

FINDC Function

Searches a string for any character in a list of characters.

Category: Character

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

Tip: Use the "KINDEXC Function" in *SAS National Language Support (NLS): Reference Guide* instead to write encoding independent code.

Syntax

FINDC(*string* <, *charlist*>)

FINDC(*string*, *charlist* <,*modifiers*>)

FINDC(*string*, *charlist*, *modifier(s)* <, *startpos*>)

FINDC(*string*, *charlist*, <*startpos*> , <*modifiers*)>

Required Arguments

string
is a character constant, variable, or expression that specifies the character string to be searched.

Tip Enclose a literal string of characters in quotation marks.

charlist
is an optional constant, variable, or character expression that initializes a list of characters. FINDC searches for the characters in this list provided that you do not specify the K modifier in the *modifier* argument. If you specify the K modifier, FINDC searches for all characters that are not in this list of characters. You can add more characters to the list by using other modifiers.

modifier
is an optional character constant, variable, or expression in which each character modifies the action of the FINDC function. The following characters, in upper- or lowercase, can be used as modifiers:

blank
is ignored.

a
A
adds alphabetic characters to the list of characters.

b
B
searches from right to left, instead of from left to right, regardless of the sign of the *startpos* argument.

c
C
adds control characters to the list of characters.

d
D
adds digits to the list of characters.

f
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
G
adds graphic characters to the list of characters.

h
H
adds a horizontal tab to the list of characters.

i
I
ignores character case during the search.

k
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.

l
L
adds lowercase letters to the list of characters.

n
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
O
processes the *charlist* and the *modifier* arguments only once, rather than every time the FINDC function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure can make FINDC run faster when you call it in a loop where the *charlist* and the *modifier* arguments do not change.

p
P
adds punctuation marks to the list of characters.

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

t
T
trims trailing blanks from the *string* and *charlist* arguments.

Note: If you want to remove trailing blanks from just one character argument instead of both (or all) character arguments, use the TRIM function instead of the FINDC function with the T modifier.

u
U
adds uppercase letters to the list of characters.

w
W
adds printable characters to the list of characters.

x
X
adds hexadecimal characters to the list of characters.

Tip If *modifier* is a constant, then enclose it in quotation marks. Specify multiple constants in a single set of quotation marks. *Modifier* can also be expressed as a variable or an expression.

Optional Argument

startpos
is an optional numeric constant, variable, or expression having an integer value that specifies the position at which the search should start and the direction in which to search.

Details

The FINDC function searches *string* for the first occurrence of the specified characters, and returns the position of the first character found. If no characters are found in *string*, then FINDC returns a value of 0.

The FINDC function allows character arguments to be null. Null arguments are treated as character strings that have a length of zero. Numeric arguments cannot be null.

If *startpos* is not specified, FINDC begins the search at the end of the string if you use the B modifier, or at the beginning of the string if you do not use the B modifier.

If *startpos* is specified, the absolute value of *startpos* specifies the position at which to begin the search. If you use the B modifier, the search always proceeds from right to left. If you do not use the B modifier, the sign of *startpos* specifies the direction in which to search. The following table summarizes the search directions:

Value of *startpos*	Action
greater than 0	search begins at position *startpos* and proceeds to the right. If *startpos* is greater than the length of the string, FINDC returns a value of 0.
less than 0	search begins at position –*startpos* and proceeds to the left. If *startpos* is less than the negative of the length of the string, the search begins at the end of the string.
equal to 0	returns a value of 0.

Comparisons

- The FINDC function searches for individual characters in a character string, whereas the FIND function searches for substrings of characters in a character string.
- The FINDC function and the INDEXC function both search for individual characters in a character string. However, the INDEXC function does not have the *modifier* nor the *startpos* arguments.
- The FINDC function searches for individual characters in a character string, whereas the VERIFY function searches for the first character that is unique to an expression. The VERIFY function does not have the *modifier* nor the *startpos* arguments.

Examples

Example 1: Searching for Characters in a String

This example searches a character string and returns the characters that are found.

```
data _null_;
   string = 'Hi, ho!';
   charlist = 'hi';
   j = 0;
   do until (j = 0);
      j = findc(string, charlist, j+1);
      if j = 0 then put +3 "That's all";
      else do;
         c = substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
   j=2 c=i
   j=5 c=h
   That's all
```

Example 2: Searching for Characters in a String and Ignoring Case

This example searches a character string and returns the characters that are found. The I modifier is used to ignore the case of the characters.

```
data _null_;
   string = 'Hi, ho!';
   charlist = 'ho';
   j = 0;
   do until (j = 0);
      j = findc(string, charlist, j+1, "i");
      if j = 0 then put +3 "That's all";
      else do;
         c = substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
   j=1 c=H
   j=5 c=h
   j=6 c=o
   That's all
```

Example 3: Searching for Characters and Using the K Modifier

This example searches a character string and returns the characters that do not appear in the character list.

```
data _null_;
   string = 'Hi, ho!';
   charlist = 'hi';
```

```
   j = 0;
   do until (j = 0);
      j = findc(string, charlist, "k", j+1);
      if j = 0 then put +3 "That's all";
      else do;
         c = substr(string, j, 1);
         put +3 j= c=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
   j=1 c=H
   j=3 c=,
   j=4 c=
   j=6 c=o
   j=7 c=!
   That's all
```

Example 4: Searching for the Characters h, i, and Blank

This example searches for the three characters h, i, and blank. The characters h and i are in lowercase. The uppercase characters H and I are ignored in this search.

```
   data _null_;
      whereishi=0;
      do until(whereishi=0);
         whereishi=findc('Hi there, Ian!','hi ',whereishi+1);
         if whereishi=0 then put "The End";
         else do;
            whatfound=substr('Hi there, Ian!',whereishi,1);
            put whereishi= whatfound=;
         end;
      end;
   run;
```

SAS writes the following output to the log:

```
whereishi=2 whatfound=i
whereishi=3 whatfound=
whereishi=5 whatfound=h
whereishi=10 whatfound=
The End
```

Example 5: Searching for the Characters h and i While Ignoring Case

This example searches for the four characters h, i, H, and I. FINDC with the i modifier ignores character case during the search.

```
   data _null_;
      whereishi_i=0;
      do until(whereishi_i=0);
         variable1='Hi there, Ian!';
         variable2='hi';
         variable3='i';
         whereishi_i=findc(variable1,variable2,variable3,whereishi_i+1);
         if whereishi_i=0 then put "The End";
         else do;
```

```
            whatfound=substr(variable1,whereishi_i,1);
            put whereishi_i= whatfound=;
         end;
      end;
   run;
```

SAS writes the following output to the log:

```
whereishi_i=1 whatfound=H
whereishi_i=2 whatfound=i
whereishi_i=5 whatfound=h
whereishi_i=11 whatfound=I
The End
```

Example 6: Searching for the Characters h and i with Trailing Blanks Trimmed

This example searches for the two characters h and i. FINDC with the t modifier trims trailing blanks from the string argument and the characters argument.

```
data _null_;
   whereishi_t=0;
   do until(whereishi_t=0);
      expression1='Hi there, '||'Ian!';
      expression2=kscan('bye or hi',3)||'  ';
      expression3=trim('t   ');
      whereishi_t=findc(expression1,expression2,expression3,whereishi_t+1);
      if whereishi_t=0 then put "The End";
      else do;
         whatfound=substr(expression1,whereishi_t,1);
         put whereishi_t= whatfound=;
      end;
   end;
run;
```

SAS writes the following lines output to the log:

```
whereishi_t=2 whatfound=i
whereishi_t=5 whatfound=h
The End
```

Example 7: Searching for all Characters, Excluding h, i, H, and I

This example searches for all of the characters in the string, excluding the characters h, i, H, and I. FINDC with the v modifier counts only the characters that do not appear in the characters argument. This example also includes the i modifier and therefore ignores character case during the search.

```
data _null_;
   whereishi_iv=0;
   do until(whereishi_iv=0);
      xyz='Hi there, Ian!';
      whereishi_iv=findc(xyz,'hi',whereishi_iv+1,'iv');
      if whereishi_iv=0 then put "The End";
      else do;
         whatfound=substr(xyz,whereishi_iv,1);
         put whereishi_iv= whatfound=;
      end;
   end;
run;
```

SAS writes the following output to the log:

```
whereishi_iv=3 whatfound=
whereishi_iv=4 whatfound=t
whereishi_iv=6 whatfound=e
whereishi_iv=7 whatfound=r
whereishi_iv=8 whatfound=e
whereishi_iv=9 whatfound=,
whereishi_iv=10 whatfound=
whereishi_iv=12 whatfound=a
whereishi_iv=13 whatfound=n
whereishi_iv=14 whatfound=!
The End
```

See Also

Functions:

- “ANYALNUM Function” on page 99
- “ANYALPHA Function” on page 101
- “ANYCNTRL Function” on page 103
- “ANYDIGIT Function” on page 105
- “ANYGRAPH Function” on page 108
- “ANYLOWER Function” on page 110
- “ANYPRINT Function” on page 113
- “ANYPUNCT Function” on page 116
- “ANYSPACE Function” on page 118
- “ANYUPPER Function” on page 120
- “ANYXDIGIT Function” on page 121
- “COUNTC Function” on page 340
- “INDEXC Function” on page 545
- “VERIFY Function” on page 944
- “NOTALNUM Function” on page 685
- “NOTALPHA Function” on page 687
- “NOTCNTRL Function” on page 689
- “NOTDIGIT Function” on page 690
- “NOTGRAPH Function” on page 695
- “NOTLOWER Function” on page 697
- “NOTPRINT Function” on page 701
- “NOTPUNCT Function” on page 702
- “NOTSPACE Function” on page 704
- “NOTUPPER Function” on page 707
- “NOTXDIGIT Function” on page 708

FINDW Function

Returns the character position of a word in a string, or returns the number of the word in a string.

Category: Character

Syntax

FINDW(*string*, *word* <, *chars*>)

FINDW(*string*, *word*, *chars*, *modifiers* <, *startpos*>)

FINDW(*string*, *word*, *chars*, *startpos* <, *modifiers*>)

FINDW(*string*, *word*, *startpos* <, *chars* <, *modifiers*> >)

Required Arguments

string
is a character constant, variable, or expression that specifies the character string to be searched.

word
is a character constant, variable, or expression that specifies the word to be searched.

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.

startpos
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should begin and the direction in which to search.

modifier
specifies a character constant, variable, or expression in which each non-blank character modifies the action of the FINDW function.

TIP If you use the *modifier* argument, then it must be positioned after the *chars* argument.

You can use the following characters as modifiers:

blank
is ignored.

a
A
adds alphabetic characters to the list of characters.

b
B
scans from right to left instead of from left to right, regardless of the sign of the *startpos* argument.

c
C
adds control characters to the list of characters.

d
D
adds digits to the list of characters.

e
E
counts the words that are scanned until the specified word is found, instead of determining the character position of the specified word in the string. Fragments of a word are not counted.

f
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
G
adds graphic characters to the list of characters.

h
H
adds a horizontal tab to the list of characters.

I
I
ignores the case of the characters.

k
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
L
adds lowercase letters to the list of characters.

m
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
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
O
processes the *chars* and *modifier* arguments only once, rather than every time the FINDW function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure, can make FINDW run faster when you call it in a loop where the *chars* and *modifier* arguments do not change.

p
P
adds punctuation marks to the list of characters.

q
Q
: ignores delimiters that are inside of 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
R
: removes leading and trailing delimiters from the *word* argument.

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

t
T
: trims trailing blanks from the *string*, *word*, and *chars* arguments.

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

w
W
: adds printable characters to the list of characters.

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

Details

Definition of "Delimiter"

"Delimiter" refers to any of several characters that are used to separate words. You can specify the delimiters by using the *chars* argument, the *modifier* argument, or both. If you specify the Q modifier, then the characters inside of substrings that are enclosed in quotation marks are not treated as delimiters.

Definition of "Word"

"Word" refers to a substring that has both of the following characteristics:

- bounded on the left by a delimiter or the beginning of the string
- bounded on the right by a delimiter or the end of the string

Note: A word can contain delimiters. In this case, the FINDW function differs from the SCAN function, in which words are defined as not containing delimiters.

Searching for a String

If the FINDW function fails to find a substring that both matches the specified word and satisfies the definition of a word, then FINDW returns a value of 0.

If the FINDW function finds a substring that both matches the specified word and satisfies the definition of a word, the value that is returned by FINDW depends on whether the E modifier is specified:

- If you specify the E modifier, then FINDW returns the number of complete words that were scanned while searching for the specified word. If *startpos* specifies a position in the middle of a word, then that word is not counted.
- If you do not specify the E modifier, then FINDW returns the character position of the substring that is found.

If you specify the *startpos* argument, then the absolute value of *startpos* specifies the position at which to begin the search. The sign of *startpos* specifies the direction in which to search:

Value of *startpos*	Action
greater than 0	search begins at position *startpos* and proceeds to the right. If *startpos* is greater than the length of the string, then FINDW returns a value of 0.
less than 0	search begins at position –*startpos* and proceeds to the left. If *startpos* is less than the negative of the length of the string, then the search begins at the end of the string.
equal to 0	FINDW returns a value of 0.

If you do not specify the *startpos* argument or the B modifier, then FINDW searches from left to right starting at the beginning of the string. If you specify the B modifier, but do not use the *startpos* argument, then FINDW searches from right to left starting at the end of the string.

Using the FINDW Function in ASCII and EBCDIC Environments

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

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

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

  In ASCII environments that do not contain the ^ character, the FINDW function uses the ~ character instead.
- If your computer uses EBCDIC characters, then the default delimiters are as follows:

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

Using Null Arguments

The FINDW function allows character arguments to be null. Null arguments are treated as character strings with a length of zero. Numeric arguments cannot be null.

Examples

Example 1: Searching a Character String for a Word

The following example searches a character string for the word “she”, and returns the position of the beginning of the word.

```
data _null_;
   whereisshe=findw('She sells sea shells? Yes, she does.','she');
   put whereisshe=;
```

```
run;
```

SAS writes the following output to the log:

```
whereisshe=28
```

Example 2: Searching a Character String and Using the 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.

```
data _null_;
   result = findw('At least 2.5 meters of rain falls in a rain forest.',
                  'rain', ' ', 25);
   put result=;
run;
```

SAS writes the following output to the log:

```
result=40
```

Example 3: Searching a Character String and Using the I Modifier and the Startpos Argument

The following example uses the I modifier and returns the position of the beginning of the word. The I modifier disregards case, and the *startpos* argument identifies the starting position from which to search.

```
data _null_;
   string='Artists from around the country display their art at
           an art festival.';
   result=findw(string, 'Art',' ', 'i', 10);
   put result=;
run;
```

SAS writes the following output to the log:

```
result=47
```

Example 4: Searching a Character String and Using the E Modifier

The following example uses the E modifier and returns the number of complete words that are scanned while searching for the word “art.”

```
data _null_;
   string='Artists from around the country display their art at
           an art festival.';
   result=findw(string,'art',' ','E');
   put result=;
run;
```

SAS writes the following output to the log:

```
result=8
```

Example 5: Searching a Character String and Using the E Modifier and the Startpos Argument

The following example uses the E modifier to count words in a character string. The word count begins at position 50 in the string. The result is 3 because “art” is the third word after the 50th character position.

```
data _null_;
   string='Artists from around the country display their art at
           an art festival.';
   result=findw(string, 'art',' ','E',50);
   put result=;
run;
```

SAS writes the following output to the log:

```
result=3
```

Example 6: Searching a Character String and Using Two Modifiers

The following example uses the I and the E modifiers to find a word in a string.

```
data _null_;
   string='The Great Himalayan National Park was created in 1984. Because
           of its terrain and altitude, the park supports a diversity
           of wildlife and vegetation.';
   result=findw(string,'park',' ','I E');
   put result=;
run;
```

SAS writes the following output to the log:

```
result=5
```

Example 7: Searching a Character String and Using the R Modifier

The following example uses the R modifier to remove leading and trailing delimiters from a word.

```
data _null_;
    string='Artists from around the country display their art at
              an art festival.';
    word='  art  ';
    result=findw(string, word, ' ', 'R');
    put result=;
run;
```

SAS writes the following output to the log:

```
result=47
```

See Also

Functions:

- “COUNTW Function” on page 343
- “FIND Function” on page 457
- “FINDC Function” on page 459
- “INDEXW Function” on page 546
- “SCAN Function” on page 848

CALL Routines:

- “CALL SCAN Routine” on page 237

FINFO Function

Returns the value of a file information item.

Category: External Files

See: "FINFO Function: Windows" in *SAS Companion for Windows*
"FINFO Function: UNIX" in *SAS Companion for UNIX Environments*
"FINFO Function: z/OS" in *SAS Companion for z/OS*

Syntax

FINFO(*file-id*,*info-item*)

Required Arguments

file-id
is a numeric constant, variable, or expression that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

info-item
is a character constant, variable, or expression that specifies the name of the file information item to be retrieved.

Details

FINFO returns the value of a system-dependent information item for an external file. FINFO returns a blank if the value given for *info-item* is invalid.

Operating Environment Information
The information available on files depends on the operating environment and access method.

Comparisons

- The FOPTNAME function determines the names of the available file information items.
- The FOPTNUM function determines the number of system-dependent information items that are available.

Example

This example stores information items about an external file in a SAS data set:

```
data info;
   length infoname infoval $60;
   drop rc fid infonum i close;
   rc=filename('abc','physical-filename');
   fid=fopen('abc');
   infonum=foptnum(fid);
   do i=1 to infonum;
      infoname=foptname(fid,i);
      infoval=finfo(fid,infoname);
```

```
      output;
   end;
   close=fclose(fid);
run;
```

See Also

Functions:

- “FCLOSE Function” on page 402
- “FOPTNUM Function” on page 489
- “MOPEN Function” on page 670

FINV Function

Returns a quantile from the *F* distribution.

Category: Quantile

Syntax

FINV (*p*, *ndf*, *ddf* <,*nc*>)

Required Arguments

p
is a numeric probability.

Range $0 \leq p < 1$

ndf
is a numeric numerator degrees of freedom parameter.

Range $ndf > 0$

ddf
is a numeric denominator degrees of freedom parameter.

Range $ddf > 0$

Optional Argument

nc
is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Details

The FINV function returns the p^{th} quantile from the *F* distribution with numerator degrees of freedom *ndf*, denominator degrees of freedom *ddf*, and noncentrality parameter *nc*. The probability that an observation from the *F* distribution is less than the

quantile is *p*. This function accepts noninteger degrees of freedom parameters *ndf* and *ddf*.

If the optional parameter *nc* is not specified or has the value 0, the quantile from the central *F* distribution is returned. The noncentrality parameter *nc* is defined such that if X and Y are normal random variables with means μ and 0, respectively, and variance 1, then X^2 / Y^2 has a noncentral *F* distribution with $nc = \mu^2$.

CAUTION:

For large values of *nc,* the algorithm could fail. In that case, a missing value is returned.

Note: FINV is the inverse of the PROBF function.

Example

These statements compute the 95th quantile value of a central *F* distribution with 2 and 10 degrees of freedom and a noncentral *F* distribution with 2 and 10.3 degrees of freedom and a noncentrality parameter equal to 2:

SAS Statement	Result
`q1=finv(.95,2,10);`	`4.1028210151`
`q2=finv(.95,2,10.3,2);`	`7.583766024`

See Also

Functions:

- "QUANTILE Function" on page 799

FIPNAME Function

Converts two-digit FIPS codes to uppercase state names.

Category: State and ZIP Code

Syntax

FIPNAME(*expression*)

Required Argument

expression

specifies a numeric constant, variable, or expression that represents a U.S. FIPS code.

Details

If the FIPNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPNAME function converts a U.S. Federal Information Processing Standards (FIPS) code to the corresponding state or U.S. territory name in uppercase, returning a value of up to 20 characters.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Example

The examples show the differences when using FIPNAME, FIPNAMEL, and FIPSTATE.

SAS Statement	Result
`x=fipname(37);` `put x;`	`NORTH CAROLINA`
`x=fipnamel(37);` `put x;`	`North Carolina`
`x=fipstate(37);` `put x;`	`NC`

See Also

Functions:

- “FIPNAMEL Function” on page 476
- “FIPSTATE Function” on page 477
- “STFIPS Function” on page 884
- “STNAME Function” on page 886
- “STNAMEL Function” on page 887

FIPNAMEL Function

Converts two-digit FIPS codes to mixed case state names.

Category: State and ZIP Code

Syntax

FIPNAMEL(*expression*)

Required Argument

expression
specifies a numeric constant, variable, or expression that represents a U.S. FIPS code.

Details

If the FIPNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPNAMEL function converts a U.S. Federal Information Processing Standards (FIPS) code to the corresponding state or U.S. territory name in mixed case, returning a value of up to 20 characters.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Example

The examples show the differences when using FIPNAME, FIPNAMEL, and FIPSTATE.

SAS Statement	Result
`x=fipname(37);` `put x;`	`NORTH CAROLINA`
`x=fipnamel(37);` `put x;`	`North Carolina`
`x=fipstate(37);` `put x;`	`NC`

See Also

Functions:

- “FIPNAME Function” on page 475
- “FIPSTATE Function” on page 477
- “STFIPS Function” on page 884
- “STNAME Function” on page 886
- “STNAMEL Function” on page 887

FIPSTATE Function

Converts two-digit FIPS codes to two-character state postal codes.

Category: State and ZIP Code

Syntax

FIPSTATE(*expression*)

Required Argument

expression
specifies a numeric constant, variable, or expression that represents a U.S. FIPS code.

Details

If the FIPSTATE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The FIPSTATE function converts a U.S. Federal Information Processing Standards (FIPS) code to a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Comparisons

The FIPNAME, FIPNAMEL, and FIPSTATE functions take the same argument but return different values. FIPNAME returns uppercase state names. FIPNAMEL returns mixed case state names. FIPSTATE returns a two-character state postal code (or world-wide GSA geographic code for U.S. territories) in uppercase.

Example

The examples show the differences when using FIPNAME, FIPNAMEL, and FIPSTATE.

SAS Statement	Result
`x=fipname(37);` `put x;`	`NORTH CAROLINA`
`x=fipnamel(37);` `put x;`	`North Carolina`
`x=fipstate(37);` `put x;`	`NC`

See Also

Functions:

- “FIPNAME Function” on page 475
- “FIPNAMEL Function” on page 476
- “STFIPS Function” on page 884
- “STNAME Function” on page 886

- “STNAMEL Function” on page 887

FIRST Function

Returns the first character in a character string.

Category: Character

Syntax

FIRST(*string*)

Required Argument

string
specifies a character string.

Details

In a DATA step, the default length of the target variable for the FIRST function is 1.

The FIRST function returns a string with a length of 1. If *string* has a length of 0, then the FIRST function returns a single blank.

Comparisons

The FIRST function returns the same result as CHAR(*string*, 1) and SUBPAD(*string*, 1, 1). While the results are the same, the default length of the target variable is different.

Example

The following example shows the results of using the FIRST function.

```
data test;
   string1="abc";
   result1=first(string1);
   string2="";
   result2=first(string2);
run;
proc print noobs data=test;
run;
```

Display 2.32 *Output from the FIRST Function*

The SAS System

string1	result1	string2	result2
abc	a		

See Also

Functions:

- "CHAR Function" on page 298

FLOOR Function

Returns the largest integer that is less than or equal to the argument, fuzzed to avoid unexpected floating-point results.

Category: Truncation

Syntax

FLOOR (*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

If the argument is within 1E-12 of an integer, the function returns that integer.

Comparisons

Unlike the FLOORZ function, the FLOOR function fuzzes the result. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function you might get unexpected results.

Example

The following SAS statements produce these results.

SAS Statement	Result

`var1=2.1;`
`a=floor(var1);`
`put a;`
	`2`

`var2=-2.4;`
`b=floor(var2);`
`put b;`
	`-3`

`c=floor(-1.6);` `put c;`	`-2`
`d=floor(1.-1.e-13);` `put d;`	`1`

SAS Statement	Result
`e=floor(763);` `put e;`	`763`
`f=floor(-223.456);` `put f;`	`-224`

See Also

Functions:

- "FLOORZ Function" on page 481

FLOORZ Function

Returns the largest integer that is less than or equal to the argument, using zero fuzzing.

Category: Truncation

Syntax

FLOORZ (*argument*)

Required Argument

argument
is a numeric constant, variable, or expression.

Comparisons

Unlike the FLOOR function, the FLOORZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the FLOOR function fuzzes the result to be equal to that integer. The FLOORZ function does not fuzz the result. Therefore, with the FLOORZ function you might get unexpected results.

Example

The following SAS statements produce these results.

SAS Statement	Result

`var1=2.1;`
`a=floorz(var1);`
`put a;`
	`2`

`var2=-2.4;`
`b=floorz(var2);`
`put b;`
	`-3`

SAS Statement	Result
`c=floorz(-1.6);` `put c;`	`-2`

`var6=(1.-1.e-13);`
`d=floorz(1-1.e-13);`
`put d;`
	`0`

`e=floorz(763);` `put e;`	`763`
`f=floorz(-223.456);` `put f;`	`-224`

See Also

Functions:

- "FLOOR Function" on page 480

FNONCT Function

Returns the value of the noncentrality parameter of an F distribution.

Category: Mathematical

Syntax

FNONCT(*x*,*ndf*,*ddf*,*prob*)

Required Arguments

x
is a numeric random variable.

Range $x \geq 0$

ndf
is a numeric numerator degree of freedom parameter.

Range $ndf > 0$

ddf
is a numeric denominator degree of freedom parameter.

Range $ddf > 0$

prob
is a probability.

Range $0 < prob < 1$

Details

The FNONCT function returns the nonnegative noncentrality parameter from a noncentral *F* distribution whose parameters are *x*, *ndf*, *ddf*, and *nc*. If *prob* 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

$$P_f(x \mid ndf,\ ddf,\ nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_f(x \mid ndf,\ ddf,\ nc) = \varepsilon^{\frac{-nc}{2}} \sum_{j=0}^{\infty} \frac{\left(\frac{nc}{2}\right)^j}{j!} I_{\frac{(ndf)x}{ddf+(ndf)x}}\left(\frac{ddf}{2} + j,\ \frac{ddf}{2}\right)$$

In the equation, $I(\ldots)$ is the probability from the beta distribution that is given by the following equation:

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

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

Example

```
data work;
   x=2;
   df=4;
   ddf=5;
   do nc=1 to 3 by .5;
      prob=probf(x,df,ddf,nc);
      ncc=fnonct(x,df,ddf,prob);
      output;
   end;
run;
proc print;
run;
```

Display 2.33 *FNONCT Example Output*

The SAS System

Obs	x	df	ddf	nc	prob	ncc
1	2	4	5	1.0	0.69277	1.00000
2	2	4	5	1.5	0.65701	1.50000
3	2	4	5	2.0	0.62232	2.00000
4	2	4	5	2.5	0.58878	2.50000
5	2	4	5	3.0	0.55642	3.00000

FNOTE Function

Identifies the last record that was read, and returns a value that the FPOINT function can use.

Category: External Files

Syntax

FNOTE(*file-id*)

Required Argument

file-id
is a numeric constant, variable, or expression that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

You can use FNOTE like a bookmark, marking the position in the file so that your application can later return to that position using FPOINT. The value that is returned by FNOTE is required by the FPOINT function to reposition the file pointer on a specific record.

To free the memory associated with each FNOTE identifier, use DROPNOTE.

Note: You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, indicated by a positive value in the variable FID, then it reads the records, stores in the variable NOTE 3 the position of the third record read, and then later uses FPOINT to point back to NOTE3 to update the file. After updating the record, it closes the file:

```
%let
fref=MYFILE;
%let rc=%sysfunc(filename(fref,
   physical-filename));
%let fid=%sysfunc(fopen(&fref,u));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fread(&fid));
         /* Read second record. */
      %let rc=%sysfunc(fread(&fid));
         /* Read third record. */
      %let rc=%sysfunc(fread(&fid));
         /* Note position of third record. */
      %let note3=%sysfunc(fnote(&fid));
         /* Read fourth record. */
      %let rc=%sysfunc(fread(&fid));
         /* Read fifth record. */
      %let rc=%sysfunc(fread(&fid));
```

```
         /* Point to third record. */
      %let rc=%sysfunc(fpoint(&fid,&note3));
         /* Read third record. */
      %let rc=%sysfunc(fread(&fid));
         /* Copy new text to FDB. */
      %let rc=%sysfunc(fput(&fid,New text));
         /* Update third record  */
         /* with data in FDB. */
      %let rc=%sysfunc(fwrite(&fid));
         /* Close file. */
      %let rc=%sysfunc(fclose(&fid));
   %end;
%let rc=%sysfunc(filename(fref));
```

See Also

Functions:

- “DROPNOTE Function” on page 387
- “FCLOSE Function” on page 402
- “FILENAME Function” on page 411
- “FOPEN Function” on page 485
- “FPOINT Function” on page 490
- “FPUT Function” on page 494
- “FREAD Function” on page 495
- “FREWIND Function” on page 496
- “FWRITE Function” on page 501
- “MOPEN Function” on page 670

FOPEN Function

Opens an external file and returns a file identifier value.

Category: External Files

See: “FOPEN Function: z/OS” in *SAS Companion for z/OS*

Syntax

FOPEN(*fileref*<,*open-mode*<,*record-length*<,*record-format*> > >)

Required Argument

fileref
is a character constant, variable, or expression that specifies the fileref assigned to the external file.

Tip If *fileref* is longer than eight characters, then it will be truncated to eight characters.

Optional Arguments

open-mode
is a character constant, variable, or expression that specifies the type of access to the file:

A APPEND mode allows writing new records after the current end of the file.

I INPUT mode allows reading only (default).

O OUTPUT mode defaults to the OPEN mode specified in the operating environment option in the FILENAME statement or function. If no operating environment option is specified, it allows writing new records at the beginning of the file.

S Sequential input mode is used for pipes and other sequential devices such as hardware ports.

U UPDATE mode allows both reading and writing.

Default I

record-length
is a numeric constant, variable, or expression that specifies the logical record length of the file. To use the existing record length for the file, specify a length of 0, or do not provide a value here.

record-format
is a character constant, variable, or expression that specifies the record format of the file. To use the existing record format, do not specify a value here. Valid values are:

B data are to be interpreted as binary data.

D use default record format.

E use editable record format.

F file contains fixed length records.

P file contains printer carriage control in operating environment-dependent record format. *Note:* For z/OS data sets with FBA or VBA record format, specify 'P' for the *record-format* argument.

V file contains variable length records.

Note: If an argument is invalid, FOPEN returns 0, and you can obtain the text of the corresponding error message from the SYSMSG function. Invalid arguments do not produce a message in the SAS log and do not set the _ERROR_ automatic variable.

Details

CAUTION:
Use OUTPUT mode with care. Opening an existing file for output overwrites the current contents of the file without warning.

The FOPEN function opens an external file for reading or updating and returns a file identifier value that is used to identify the open file to other functions. You must associate a fileref with the external file before calling the FOPEN function. FOPEN returns a 0 if the file could not be opened. You can assign filerefs by using the FILENAME statement or the FILENAME external file access function. Under some operating environments, you can also assign filerefs by using system commands.

If you call the FOPEN function from a macro, then the result of the call is valid only when it is passed to functions in a macro. If you call the FOPEN function from the DATA step, then the result is valid only when it is passed to functions in the same DATA step.

Operating Environment Information

It is good practice to use the FCLOSE function at the end of a DATA step if you used FOPEN to open the file, even though using FCLOSE might not be required in your operating environment. For more information about FOPEN, see the SAS documentation for your operating environment.

Examples

Example 1: Opening a File Using Defaults

This example assigns the fileref MYFILE to an external file and attempts to open the file for input using all defaults. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf));
```

Example 2: Opening a File without Using Defaults

This example attempts to open a file for input without using defaults. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let fid=%sysfunc(fopen(file2,o,132,e));
```

Example 3: Handling Errors

This example shows how to check for errors and write an error message from the SYSMSG function.

```
data _null_;
   f=fopen('bad','?');
   if not f then do;
      m=sysmsg();
      put m;
      abort;
      end;
      ... more SAS statements ...
   run;
```

See Also

Functions:

- “DOPEN Function” on page 382
- “FCLOSE Function” on page 402
- “FILENAME Function” on page 411
- “FILEREF Function” on page 414
- “MOPEN Function” on page 670
- “SYSMSG Function” on page 905

Statements:

- "FILENAME Statement" in *SAS Statements: Reference*

FOPTNAME Function

Returns the name of an item of information about a file.

Category: External Files

See: "FOPTNAME Function: Windows" in *SAS Companion for Windows*
"FOPTNAME Function: UNIX" in *SAS Companion for UNIX Environments*
"FOPTNAME Function: z/OS" in *SAS Companion for z/OS*

Syntax

FOPTNAME(*file-id*,*nval*)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

nval
is a numeric constant, variable, or expression that specifies the number of the information item.

Details

FOPTNAME returns a blank if an error occurred.

Windows Specifics
The number, value, and type of information items that are available depend on the operating environment and access method.

Examples

Example 1: Retrieving File Information Items and Writing Them to the Log

This example retrieves the system-dependent file information items that are available and writes them to the log:

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%let infonum=%sysfunc(foptnum(&fid));
%do j=1 %to &infonum;
   %let name=%sysfunc(foptname(&fid,&j));
   %let value=%sysfunc(finfo(&fid,&name));
   %put File attribute &name equals &value;
%end;
%let rc=%sysfunc(fclose(&fid));
```

```
%let rc=%sysfunc(filename(filrf));
```

Example 2: Creating a Data Set with Names and Values of File Attributes

This example creates a data set that contains the name and value of the available file attributes:

```
data fileatt;
   length name $ 20 value $ 40;
   drop rc fid j infonum;
   rc=filename("myfile","physical-filename");
   fid=fopen("myfile");
   infonum=foptnum(fid);
   do j=1 to infonum;
      name=foptname(fid,j);
      value=finfo(fid,name);
      put 'File attribute ' name
       'has a value of ' value;
      output;
   end;
   rc=filename("myfile");
run;
```

See Also

Functions:

- “DINFO Function” on page 378
- “DOPTNAME Function” on page 384
- “DOPTNUM Function” on page 385
- “FCLOSE Function” on page 402
- “FILENAME Function” on page 411
- “FINFO Function” on page 473
- “FOPEN Function” on page 485
- “FOPTNUM Function” on page 489
- “MOPEN Function” on page 670

FOPTNUM Function

Returns the number of information items that are available for an external file.

Category: External Files

See: “FOPTNUM Function: Windows” in *SAS Companion for Windows*
“FOPTNUM Function: UNIX” in *SAS Companion for UNIX Environments*
“FOPTNUM Function: z/OS” in *SAS Companion for z/OS*

Syntax

FOPTNUM(*file-id*)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

Windows Specifics
The number, value, and type of information items that are available depend on the operating environment.

Comparisons

- Use FOPTNAME to determine the names of the items that are available for a particular operating environment.
- Use FINFO to retrieve the value of a particular information item.

Example

This example opens the external file with the fileref MYFILE and determines the number of system-dependent file information items available:

```
%let fid=%sysfunc(fopen(myfile));
%let infonum=%sysfunc(foptnum(&fid));
```

See Also

Functions:

- “DINFO Function” on page 378
- “DOPTNAME Function” on page 384
- “DOPTNUM Function” on page 385
- “FINFO Function” on page 473
- “FOPEN Function” on page 485
- “FOPTNAME Function” on page 488
- “MOPEN Function” on page 670

FPOINT Function

Positions the read pointer on the next record to be read.

Category: External Files

Syntax

FPOINT(*file-id*,*note-id*)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

note-id
specifies the identifier that was assigned by the FNOTE function.

Details

FPOINT returns 0 if the operation was successful, or ≠0 if it was not successful. FPOINT determines only the record to read next. It has no impact on which record is written next. When you open the file for update, FWRITE writes to the most recently read record.

Note: You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it reads the records and uses NOTE3 to store the position of the third record read. Later, it points back to NOTE3 to update the file, and closes the file afterward:

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf,u));
%if &fid > 0 %then
   %do;
         /* Read first record. */
      %let rc=%sysfunc(fread(&fid));
         /* Read second record. */
      %let rc=%sysfunc(fread(&fid));
         /* Read third record. */
      %let rc=%sysfunc(fread(&fid));
         /* Note position of third record. */
      %let note3=%sysfunc(fnote(&fid));
         /* Read fourth record. */
      %let rc=%sysfunc(fread(&fid));
         /* Read fifth record. */
      %let rc=%sysfunc(fread(&fid));
         /* Point to third record. */
      %let rc=%sysfunc(fpoint(&fid,&note3));
         /* Read third record. */
      %let rc=%sysfunc(fread(&fid));
         /* Copy new text to FDB. */
      %let rc=%sysfunc(fput(&fid,New text));
         /* Update third record  */
         /* with data in FDB. */
      %let rc=%sysfunc(fwrite(&fid));
```

```
        /* Close file. */
      %let rc=%sysfunc(fclose(&fid));
   %end;
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:

- “DROPNOTE Function” on page 387
- “FCLOSE Function” on page 402
- “FILENAME Function” on page 411
- “FNOTE Function” on page 484
- “FOPEN Function” on page 485
- “FPUT Function” on page 494
- “FREAD Function” on page 495
- “FREWIND Function” on page 496
- “FWRITE Function” on page 501
- “MOPEN Function” on page 670

FPOS Function

Sets the position of the column pointer in the File Data Buffer (FDB).

Category: External Files

Syntax

FPOS(*file-id*,*nval*)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

nval
is a numeric constant, variable, or expression that specifies the column at which to set the pointer.

Details

FPOS returns 0 if the operation was successful, ≠0 if it was not successful. If you open a file in output mode and the specified position is past the end of the current record, the size of the record is increased appropriately. However, in a fixed block or VBA file, if you specify a column position beyond the end of the record, the record size does not change and the text string is not written to the file.

If you open a file in update mode and the specified position is not past the end of the current record, then SAS writes the record to the file. If the specified position is past the

end of the current record, then SAS returns an error message and does not write the new record:

```
ERROR: Cannot increase record length in update mode.
```

Note: If you use the update mode with the FOPEN function, then you must execute FREAD before you execute FWRITE functions.

Example

This example assigns the fileref MYFILE to an external file and opens the file in update mode. If the file is opened successfully, indicated by a positive value in the variable FID, SAS reads a record and places data into the file's buffer at column 12. If the resulting record length is less than or equal to the original record length, then SAS writes the record and closes the file. If the resulting record length is greater than the original record length, then SAS writes an error message to the log.

```
%macro ptest;
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,test.txt));
%let fid=%sysfunc(fopen(&filrf,o));
%let rc=%sysfunc(fread(&fid));
%put &fid;
%if (&fid > 0) %then
   %do;
      %let dataline=This is some data.;
         /* Position at column 12 in the FDB. */
      %let rc=%sysfunc(fpos(&fid,12));
      %put &rc one;
         /* Put the data in the FDB. */
      %let rc=%sysfunc(fput(&fid,&dataline));
      %put &rc two;
      %if (&rc ne 0) %then
         %do;
            %put %sysfunc(sysmsg());
         %end;
      %else %do;
         /* Write the record. */
      %let rc=%sysfunc(fwrite(&fid));
      %if (&rc ne 0) %then
         %do;
            %put write fails &rc;
         %end;
      %end;
         /* Close the file. */
      %let rc=%sysfunc(fclose(&fid));
   %end;
%let rc=%sysfunc(filename(filrf));
%mend;
%ptest;
```

Log 2.9 *Output from the FPOS Function*

```
1
0 one
0 two
```

See Also

Functions:

- "FCLOSE Function" on page 402
- "FCOL Function" on page 403
- "FILENAME Function" on page 411
- "FOPEN Function" on page 485
- "FPUT Function" on page 494
- "FWRITE Function" on page 501
- "MOPEN Function" on page 670

FPUT Function

Moves data to the File Data Buffer (FDB) of an external file, starting at the FDB's current column position.

Category: External Files

Syntax

FPUT(*file-id*,*cval*)

Required Arguments

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

cval
is a character constant, variable, or expression that specifies the file data.

Details

FPUT returns 0 if the operation was successful, ≠0 if it was not successful. The number of bytes moved to the FDB is determined by the length of the variable. The value of the column pointer is then increased to one position past the end of the new text.

Note: You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file in APPEND mode. If the file is opened successfully, indicated by a positive value in the variable FID, it moves data to the FDB using FPUT, appends a record using FWRITE, and then closes the file. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%macro ptest;
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,test.txt));
```

```
%let fid=%sysfunc(fopen(&filrf,a));
%if &fid > 0 %then
   %do;
      %let rc=%sysfunc(fread(&fid));
      %let mystring=This is some data.;
      %let rc=%sysfunc(fput(&fid,&mystring));
      %let rc=%sysfunc(fwrite(&fid));
      %let rc=%sysfunc(fclose(&fid));
   %end;
%else
   %put %sysfunc(sysmsg());
%let rc=%sysfunc(filename(filrf));
%put return code = &rc;
%mend;
%ptest;
```

SAS writes the following output to the log:

```
return code = 0
```

See Also

Functions:

- “FCLOSE Function” on page 402
- “FILENAME Function” on page 411
- “FNOTE Function” on page 484
- “FOPEN Function” on page 485
- “FPOINT Function” on page 490
- “FPOS Function” on page 492
- “FWRITE Function” on page 501
- “MOPEN Function” on page 670
- “SYSMSG Function” on page 905

FREAD Function

Reads a record from an external file into the File Data Buffer (FDB).

Category: External Files

Syntax

FREAD(*file-id*)

Required Argument

file-id

is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FREAD returns 0 if the operation was successful, ≠0 if it was not successful. The position of the file pointer is updated automatically after the read operation so that successive FREAD functions read successive file records.

To position the file pointer explicitly, use FNOTE, FPOINT, and FREWIND.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file opens successfully, it lists all of the file's records in the log:

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%if &fid > 0 %then
   %do %while(%sysfunc(fread(&fid)) = 0);
      %let rc=%sysfunc(fget(&fid,c,200));
           %put &c;
      %end;
   %let rc=%sysfunc(fclose(&fid));
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:

- “FCLOSE Function” on page 402
- “FGET Function” on page 409
- “FILENAME Function” on page 411
- “FNOTE Function” on page 484
- “FOPEN Function” on page 485
- “FREWIND Function” on page 496
- “FREWIND Function” on page 496
- “MOPEN Function” on page 670

FREWIND Function

Positions the file pointer to the start of the file.

Category: External Files

Syntax

FREWIND(*file-id*)

Required Argument

file-id

is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Details

FREWIND returns 0 if the operation was successful, ≠0 if it was not successful. FREWIND has no effect on a file opened with sequential access.

Example

This example assigns the fileref MYFILE to an external file. Then it opens the file and reads the records until the end of the file is reached. The FREWIND function then repositions the pointer to the beginning of the file. The first record is read again and stored in the File Data Buffer (FDB). The first token is retrieved and stored in the macro variable VAL:

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf));
%let rc=0;
%do %while (&rc ne -1);
   /* Read a record. */
   %let rc=%sysfunc(fread(&fid));
%end;
   /* Reposition pointer to beginning of file. */
%if &rc = -1 %then
  %do;
    %let rc=%sysfunc(frewind(&fid));
       /* Read first record. */
    %let rc=%sysfunc(fread(&fid));
       /* Read first token  */
       /* into macro variable VAL. */
    %let rc=%sysfunc(fget(&fid,val));
    %put val=&val;
  %end;
%else
  %put Error on fread=%sysfunc(sysmsg());
%let rc=%sysfunc(fclose(&fid));
%let rc=%sysfunc(filename(filrf));
```

See Also

Functions:

- “FCLOSE Function” on page 402
- “FGET Function” on page 409
- “FILENAME Function” on page 411
- “FOPEN Function” on page 485
- “FREAD Function” on page 495
- “MOPEN Function” on page 670

- "SYSMSG Function" on page 905

FRLEN Function

Returns the size of the last record that was read, or, if the file is opened for output, returns the current record size.

Category: External Files

Syntax

FRLEN(*file-id*)

Required Argument

file-id

is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Example

This example opens the file that is identified by the fileref MYFILE. It determines the minimum and maximum length of records in the external file and writes the results to the log:

```
%let fid=%sysfunc(fopen(myfile));
%let min=0;
%let max=0;
%if (%sysfunc(fread(&fid)) = 0) %then
   %do;
      %let min=%sysfunc(frlen(&fid));
      %let max=&min;
      %do %while(%sysfunc(fread(&fid)) = 0);
         %let reclen=%sysfunc(frlen(&fid));
         %if (&reclen > &max) %then
            %let max=&reclen;
         %if (&reclen < &min) %then
            %let min=&reclen;
      %end;
%end;
%let rc=%sysfunc(fclose(&fid));
%put max=&max min=&min;
```

See Also

Functions:

- "FCLOSE Function" on page 402
- "FOPEN Function" on page 485
- "FREAD Function" on page 495
- "MOPEN Function" on page 670

FSEP Function

Sets the token delimiters for the FGET function.

Category: External Files

Syntax

FSEP(*file-id*,*characters*<,'x' | 'X'>)

Required Arguments

file-id

is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

character

is a character constant, variable, or expression that specifies one or more delimiters that separate items in the File Data Buffer (FDB). Each character listed is a delimiter. That is, if *character* is #@, either # or @ can separate items. Multiple consecutive delimiters, such as @#@, are treated as a single delimiter.

Default blank

Optional Argument

'x' | 'X'

specifies that the character delimiter is a hexadecimal value.

Restrictions 'x' and 'X' are the only valid values for this argument. All other values will cause an error to occur.

If you pass 'x' or 'X' as the third argument, a valid hexadecimal string must be passed as the second argument, *character*. Otherwise, the function will fail. A valid hexadecimal string is an even number of 0–9 and A–F characters.

Tip If you use a macro statement, then quotation marks enclosing x or X are not required.

Details

FSEP returns 0 if the operation was successful, ≠0 if it was not successful.

Example

An external file has data in this form:

```
John J. Doe,Male,25,Weight Lifter
Pat O'Neal,Female,22,Gymnast
```

Note that each field is separated by a comma.

This example reads the file that is identified by the fileref MYFILE, using the comma as a separator, and writes the values for NAME, GENDER, AGE, and WORK to the SAS log. Note that in a macro statement that you do not enclose character strings in quotation marks, but a literal comma in a function argument must be enclosed in a macro quoting function such as %STR.

```
%let fid=%sysfunc(fopen(myfile));
%let rc=%sysfunc(fsep(&fid,%str(,)));
%do %while(%sysfunc(fread(&fid)) = 0);
   %let rc=%sysfunc(fget(&fid,name));
   %let rc=%sysfunc(fget(&fid,gender));
   %let rc=%sysfunc(fget(&fid,age));
   %let rc=%sysfunc(fget(&fid,work));
   %put name=%bquote(&name) gender=&gender
      age=&age work=&work;
%end;
%let rc=%sysfunc(fclose(&fid));
```

See Also

Functions:

- “FCLOSE Function” on page 402
- “FGET Function” on page 409
- “FOPEN Function” on page 485
- “FREAD Function” on page 495
- “MOPEN Function” on page 670

FUZZ Function

Returns the nearest integer if the argument is within 1E−12 of that integer.

Category: Truncation

Syntax

FUZZ(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The FUZZ function returns the nearest integer value if the argument is within 1E−12 of the integer (that is, if the absolute difference between the integer and argument is less than 1E−12). Otherwise, the argument is returned.

Example

The following SAS statements produce these results.

SAS Statement	Result

`var1=5.9999999999999;`
`x=fuzz(var1);`
`put x 16.14`
	`6.000000000000000`

`x=fuzz(5.99999999);` `put x 16.14;`	`5.999999990000000`

FWRITE Function

Writes a record to an external file.

Category: External Files

Syntax

FWRITE(*file-id*<,*cc*>)

Required Argument

file-id
is a numeric variable that specifies the identifier that was assigned when the file was opened, generally by the FOPEN function.

Optional Argument

cc
is a character constant, variable, or expression that specifies a carriage-control character:

blank	starts the record on a new line.
0	skips one blank line before a new line.
-	skips two blank lines before a new line.
1	starts the line on a new page.
+	overstrikes the line on a previous line.
P	interprets the line as a computer prompt.
=	interprets the line as carriage control information.
all else	starts the line record on a new line.

Details

FWRITE returns 0 if the operation was successful, ≠0 if it was not successful. FWRITE moves text from the File Data Buffer (FDB) to the external file. In order to use the carriage-control characters, you must open the file with a record format of **`P`** (print format) in FOPEN.

Note: When you use the update mode, you must execute FREAD before you execute FWRITE. You cannot write a new record in place of the current record if the new record has a length that is greater than the current record.

Example

This example assigns the fileref MYFILE to an external file and attempts to open the file. If the file is opened successfully, it writes the numbers 1 to 50 to the external file, skipping two blank lines. Note that in a macro statement that you do not enclose character strings in quotation marks.

```
%let filrf=myfile;
%let rc=%sysfunc(filename(filrf,
   physical-filename));
%let fid=%sysfunc(fopen(&filrf,o,0,P));
%do i=1 %to 50;
   %let rc=%sysfunc(fput(&fid,
      %sysfunc(putn(&i,2.))));
   %if (%sysfunc(fwrite(&fid,-)) ne 0) %then
     %put %sysfunc(sysmsg());
%end;
%let rc=%sysfunc(fclose(&fid));
```

See Also

Functions:

- “FAPPEND Function” on page 400
- “FCLOSE Function” on page 402
- “FGET Function” on page 409
- “FILENAME Function” on page 411
- “FOPEN Function” on page 485
- “FPUT Function” on page 494
- “SYSMSG Function” on page 905

GAMINV Function

Returns a quantile from the gamma distribution.

Category: Quantile

Syntax

GAMINV(*p*,*a*)

Required Arguments

p
is a numeric probability.

Range $0 \leq p < 1$

a
is a numeric shape parameter.

Range $a > 0$

Details

The GAMINV function returns the p^{th} quantile from the gamma distribution, with shape parameter a. The probability that an observation from a gamma distribution is less than or equal to the returned quantile is p.

Note: GAMINV is the inverse of the PROBGAM function.

Example

The following SAS statements produce these results.

SAS Statement	Result
`q1=gaminv(0.5,9);`	`8.6689511844`
`q2=gaminv(0.1,2.1);`	`0.5841932369`

See Also

Functions:

- "QUANTILE Function" on page 799

GAMMA Function

Returns the value of the gamma function.

Category: Mathematical

Syntax

GAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Restriction Nonpositive integers are invalid.

Details

The GAMMA function returns the integral given by

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

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

Example

This SAS statement produces this result.

SAS Statement	Result
`x=gamma(6);`	`120`

GARKHCLPRC Function

Calculates call prices for European options on stocks, based on the Garman-Kohlhagen model.

Category: Financial

Syntax

GARKHCLPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Required Arguments

E
is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t
is a nonmissing value that specifies the time to maturity.

S
is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

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

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

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*.

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:

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

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:

$$CALL = \max((S - E), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" on page 8.

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 SAS statements produce these results.

SAS Statement	Result
	----+----1----+---2--
a=garkhclprc(1000, .5, 950, 4, 4, 2); put a;	65.335687119
b=garkhclprc(850, 1.2, 125, 5, 3, 1); put b;	1.9002767538
c=garkhclprc(7500, .9, 950, 3, 2, 2); put c;	69.328647279
d=garkhclprc(5000, -.5, 237, 3, 3, 2); put d;	0

See Also

Functions:

- "GARKHPTPRC Function" on page 506

GARKHPTPRC Function

Calculates put prices for European options on stocks, based on the Garman-Kohlhagen model.

Category: Financial

Syntax

GARKHPTPRC(*E*, *t*, *S*, R_d, R_f, *sigma*)

Required Arguments

E

is a nonmissing, positive value that specifies the exercise price.

Requirement Specify *E* and *S* in the same units.

t

is a nonmissing value that specifies the time to maturity.

S

is a nonmissing, positive value that specifies the spot currency price.

Requirement Specify *S* and *E* in the same units.

***R*d**
is a nonmissing, positive fraction that specifies the risk-free domestic interest rate for period *t*.

Requirement Specify a value for R_d for the same time period as the unit of *t*.

***R*f**
is a nonmissing, positive fraction that specifies the risk-free foreign interest rate for period *t*.

Requirement Specify a value for R_t for the same time period as the unit of *t*.

sigma
is a nonmissing, positive fraction that specifies the volatility of the currency rate.

Requirement Specify a value for *sigma* for the same time period as the unit of *t*.

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:

$$PUT = CALL - S\left(\varepsilon^{-R_f t}\right) + E\left(\varepsilon^{-R_d t}\right)$$

Arguments

S
specifies the spot currency price.

E
specifies the exercise price of the option.

t
specifies the time to expiration.

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:

$$PUT = \max((E - S), 0)$$

For information about the basics of pricing, see "Using Pricing Functions" on page 8.

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 SAS statements produce these results.

SAS Statement	Result
	----+----1----+---2--
a=garkhptprc(1000, .5, 950, 4, 4, 2);	72.102451281
b=garkhptprc(850, 1.2, 125, 5, 3, 1);	0.5917507981
c=garkhptprc(7500, .9, 950, 3, 2, 2);	416.33604902
d=garkhptprc(5000, -.5, 237, 3, 3, 2);	0

See Also

Functions:

- "GARKHCLPRC Function" on page 504

GCD Function

Returns the greatest common divisor for one or more integers.

Category: Mathematical

Syntax

GCD(*x1, x2, x3, ..., xn*)

Required Argument

x
specifies a numeric constant, variable, or expression that has an integer value.

Details

The GCD (greatest common divisor) function returns the greatest common divisor of one or more integers. For example, the greatest common divisor for 30 and 42 is 6. The greatest common divisor is also called the highest common factor.

If any of the arguments are missing, then the returned value is a missing value.

Example

The following example returns the greatest common divisor of the integers 10 and 15.

```
data _null_;
   x=gcd(10, 15);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=5
```

See Also

Functions:

- "LCM Function" on page 614

GEODIST Function

Returns the geodetic distance between two latitude and longitude coordinates.

Category: Distance

Syntax

GEODIST(*latitude-1*, *longitude-1*, *latitude-2*, *longitude-2* <,*options*>)

Required Arguments

latitude
is a numeric constant, variable, or expression that specifies the coordinate of a given position north or south of the equator. Coordinates that are located north of the equator have positive values; coordinates that are located south of the equator have negative values.

Restriction If the value is expressed in degrees, it must be between 90 and –90. If the value is expressed in radians, it must be between pi/2 and –pi/2.

longitude
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.

Optional Argument

option
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.

Details

The GEODIST function computes the geodetic distance between any two arbitrary latitude and longitude coordinates. Input values can be expressed in degrees or in radians.

Examples

Example 1: Calculating the Geodetic Distance in Kilometers

The following example shows the geodetic distance in kilometers between Mobile, AL (latitude 30.68 N, longitude 88.25 W), and Asheville, NC (latitude 35.43 N, longitude 82.55 W). The program uses the default K option.

```
data _null_;
   distance=geodist(30.68, -88.25, 35.43, -82.55);
   put 'Distance= ' distance 'kilometers';
run;
```

SAS writes the following output to the log:

```
Distance= 748.6529147 kilometers
```

Example 2: Calculating the Geodetic Distance in Miles

The following example uses the M option to compute the geodetic distance in miles between Mobile, AL (latitude 30.68 N, longitude 88.25 W), and Asheville, NC (latitude 35.43 N, longitude 82.55 W).

```
data _null_;
   distance=geodist(30.68, -88.25, 35.43, -82.55, 'M');
   put 'Distance = ' distance 'miles';
run;
```

SAS writes the following output to the log:

```
Distance = 465.29081088 miles
```

Example 3: Calculating the Geodetic Distance with Input Measured in Degrees

The following example uses latitude and longitude values that are expressed in degrees to compute the geodetic distance between two locations. Both the D and the M options are specified in the program.

```
   data _null_;
      input lat1 long1 lat2 long2;
```

```
   Distance = geodist(lat1,long1,lat2,long2,'DM');
   put 'Distance = ' Distance 'miles';
   datalines;
35.2 -78.1 37.6 -79.8
;
run;
```

SAS writes the following output to the log:

```
Distance = 190.72474282 miles
```

Example 4: Calculating the Geodetic Distance with Input Measured in Radians

The following example uses latitude and longitude values that are expressed in radians to compute the geodetic distance between two locations. The program converts degrees to radians before executing the GEODIST function. Both the R and the M options are specified in this program.

```
data _null_;
   input lat1 long1 lat2 long2;
   pi = constant('pi');
   lat1 = (pi*lat1)/180;
   long1 = (pi*long1)/180;
   lat2 = (pi*lat2)/180;
   long2 = (pi*long2)/180;
   Distance = geodist(lat1,long1,lat2,long2,'RM');
   put 'Distance= ' Distance 'miles';
   datalines;
35.2 -78.1 37.6 -79.8
;
run;
```

SAS writes the following output to the log:

```
Distance= 190.72474282 miles
```

References

Vincenty, T. “Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations.” 1975. *Survey Review* 22: 99–93.

GEOMEAN Function

Returns the geometric mean.

Category: Descriptive Statistics

Syntax

GEOMEAN(*argument*<,*argument*,…>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If any argument is zero, then the geometric mean is zero. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the geometric mean of the nonmissing values.

Let n be the number of arguments with nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the n^{th} root of the product of the values:

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

Equivalently, the geometric mean is

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

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

Comparisons

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

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=geomean(1,2,2,4);`	`2`
`x2=geomean(.,2,4,8);`	`4`
`x3=geomean(of x1-x2);`	`2.8284271247`

See Also

Functions:

- “GEOMEANZ Function” on page 513
- “HARMEAN Function” on page 526

- "HARMEANZ Function" on page 527
- "MEAN Function" on page 658

GEOMEANZ Function

Returns the geometric mean, using zero fuzzing.

Category: Descriptive Statistics

Syntax

GEOMEANZ(*argument*<,*argument*,…>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If any argument is zero, then the geometric mean is zero. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the geometric mean of the nonmissing values.

Let n be the number of arguments with nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The geometric mean is the n^{th} root of the product of the values:

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

Equivalently, the geometric mean is

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

Comparisons

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

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=geomeanz(1,2,2,4);`	`2`
`x2=geomeanz(.,2,4,8);`	`4`
`x3=geomeanz(of x1-x2);`	`2.8284271247`

See Also

Functions:

- "GEOMEAN Function" on page 511
- "HARMEAN Function" on page 526
- "HARMEANZ Function" on page 527
- "MEAN Function" on page 658

GETOPTION Function

Returns the value of a SAS system or graphics option.

Category: Special

Syntax

GETOPTION(*option-name*<, *return-value-option*><*return-value-formatting-options*>)

Required Argument

option-name
is a character constant, variable, or expression that specifies the name of the system option.

Tips Do not put an equal sign after the name. For example, write PAGESIZE= as PAGESIZE.

SAS options that are passwords, such as EMAILPW and METAPASS, return the value **xxxxxxxx**, and not the actual password.

Return Value Options

DEFAULTVALUE
returns the default option value.

Restriction DEFAULTVALUE is valid only for SAS system options. SAS issues a warning message when the DEFAULTVALUE option is specified and *option-name* is a graphics option.

HOWSCOPE
returns a character string that specifies the scope of an option.

Restriction HOWSCOPE is valid only for SAS system options. SAS issues a warning message when the HOWSCOPE option is specified and *option-name* is a graphics option.

HOWSET
returns a character string that specifies how an option value was set.

Restriction HOWSET is valid only for SAS system options. SAS issues a warning message when the HOWSET option is specified and *option-name* is a graphics option.

STARTUPVALUE
returns the system option value that was used to start SAS either on the command line or in a configuration file.

Restriction STARTUPVALUE is valid only for SAS system options. SAS issues a warning message when the STARTUPVALUE option is specified and *option-name* is a graphics option.

Return Value Formatting Options

CM
reports graphic units of measure in centimeters.

Restriction CM is valid only for graphics options and the following SAS system options: BOTTOMMARGIN, TOPMARGIN, RIGHTMARGIN, and LEFTMARGIN. SAS writes a note to the log when the CM option is specified and *option-name* is not a graphics option or an option that specifies a margin value.

EXPAND
for options that contain environment variables, returns the option value with the value of the environment variable.

Restrictions Variable expansion is valid only in the Windows and UNIX operating environments.

EXPAND is valid only for character system option values. EXPAND is ignored if *option-name* has an option type of Boolean, such as CENTER or NOCENTER, or if the value of the option is numeric.

Note SAS issues a note when EXPAND is specified for Boolean options and for options that have numeric values. SAS issues a warning when EXPAND is specified and the option is a graphics option.

Tip By default, some option values are displayed with expanded variable values. Other options require the EXPAND option in the PROC OPTIONS statement. Use the DEFINE option in the PROC OPTIONS statement to determine whether an option value expands variables by default or if the EXPAND option is required. If the output from PROC OPTIONS DEFINE shows the following information, you must use the EXPAND option to expand variable values:

```
Expansion: Environment variables, within the option value, are not expanded
```

KEYEXPAND
for options that contain environment variables, returns the value in the format **option-name**=**value**.

Restriction KEYEXPAND is valid only for character system option values. SAS issues an error message when the KEYEXPAND option is specified and *option-name* is a graphics option. KEYEXPAND is ignored if *option-name* has an option type of Boolean, such as CENTER or NOCENTER, or if the value of the option is numeric.

KEYWORD
returns option values in a **`option-name=value`** format that would be suitable for direct use in the SAS OPTIONS or GOPTIONS global statements.

Restrictions KEYWORD is not valid when it is used with the HEXVALUE, EXPAND, KEYEXPAND, or LOGNUMBERFORMAT options. SAS writes a note to the log when the GETOPTION function contains conflicting options.

KEYWORD is valid only for character or numeric system option values. KEYWORD is ignored for system options whose option type is Boolean, such as CENTER or NOCENTER. SAS issues an error message when the KEYWORD option is specified and *option-name* is a graphics option.

Note For a system option with a null value, the GETOPTION function returns a value of ' ' (single quotation marks with a blank space between them). An example is EMAILID=' '.

HEXVALUE
returns the option value as a hexadecimal value.

Restriction HEXVALUE is valid only for character or numeric system option values. If HEXVALUE is specified for system options whose option type is Boolean, such as CENTER or NOCENTER, or if *option-name* is a graphics option, SAS issues an error message.

IN
reports graphic units of measure in inches.

Restriction IN is valid only for graphics options and the following SAS system options: BOTTOMMARGIN, TOPMARGIN, RIGHTMARGIN, and LEFTMARGIN. SAS writes a note to the log when the IN option is specified and *option-name* is not a graphics option or an option that specifies a margin value.

LOGNUMBERFORMAT
formats SAS system option values using locale-specific punctuation.

Restriction Do not use LOGNUMBERFORMAT if the returned value is used to set an option value by using the OPTIONS statement. The OPTIONS statement does not accept commas in numeric values.

Examples

Example 1: Using GETOPTION to Save and Restore the YEARCUTOFF Option

This example saves the value of the YEARCUTOFF option, processes SAS statements based on the value of the YEARCUTOFF option, and then resets the value to 1920 if it is not already 1920.

```
/* Save the value of the YEARCUTOFF system option */
%let cutoff=%sysfunc(getoption(yearcutoff,keyword));

data ages;
  if getoption('yearcutoff') = '1920' then
     do;
        ...more SAS statements...
     end;
     else do;
             ...more SAS statements...
             /* Reset YEARCUTOFF   */
             options &cutoff;
          end;
run;
```

Example 2: Using GETOPTION to Obtain Different Reporting Options

This example defines a macro to illustrate the use of the GETOPTION function to obtain the value of system and graphics options by using different reporting options.

```
   %macro showopts;
     %put MAPS= %sysfunc(
        getoption(MAPS));
     %put MAPSEXPANDED= %sysfunc(
        getoption(MAPS, EXPAND));
     %put PAGESIZE= %sysfunc(
        getoption(PAGESIZE));
     %put PAGESIZESETBY= %sysfunc(
        getoption(PAGESIZE, HOWSET));
     %put PAGESIZESCOPE= %sysfunc(
        getoption(PAGESIZE, HOWSCOPE));
     %put PS= %sysfunc(
        getoption(PS));
     %put LS= %sysfunc(
        getoption(LS));
     %put PS(keyword form)= %sysfunc(
        getoption(PS,keyword));
     %put LS(keyword form)= %sysfunc(
        getoption(LS,keyword));
     %put FORMCHAR= %sysfunc(
        getoption(FORMCHAR));
     %put HSIZE= %sysfunc(
        getoption(HSIZE));
     %put VSIZE= %sysfunc(
        getoption(VSIZE));
     %put HSIZE(in/keyword form)= %sysfunc(
```

```
      getoption(HSIZE,in,keyword));
   %put HSIZE(cm/keyword form)= %sysfunc(
      getoption(HSIZE,cm,keyword));
   %put VSIZE(in/keyword form)= %sysfunc(
      getoption(VSIZE,in,keyword));
   %put HSIZE(cm/keyword form)= %sysfunc(
      getoption(VSIZE,cm,keyword));
%mend;
goptions VSIZE=8.5 in HSIZE=11 in;
options PAGESIZE=67;
%showopts
```

The following is the SAS log:

```
NOTE: PROCEDURE PRINTTO used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

6    %macro showopts;
7      %put MAPS= %sysfunc(
8         getoption(MAPS));
9      %put MAPSEXPANDED= %sysfunc(
10        getoption(MAPS, EXPAND));
11     %put PAGESIZE= %sysfunc(
12        getoption(PAGESIZE));
13     %put PAGESIZESETBY= %sysfunc(
14        getoption(PAGESIZE, HOWSET));
15     %put PAGESIZESCOPE= %sysfunc(
16        getoption(PAGESIZE, HOWSCOPE));
17     %put PS= %sysfunc(
18        getoption(PS));
19     %put LS= %sysfunc(
20        getoption(LS));
21     %put PS(keyword form)= %sysfunc(
22        getoption(PS,keyword));
23     %put LS(keyword form)= %sysfunc(
24        getoption(LS,keyword));
25     %put FORMCHAR= %sysfunc(
26        getoption(FORMCHAR));
27     %put HSIZE= %sysfunc(
28        getoption(HSIZE));
29     %put VSIZE= %sysfunc(
30        getoption(VSIZE));
31     %put HSIZE(in/keyword form)= %sysfunc(
32        getoption(HSIZE,in,keyword));
33     %put HSIZE(cm/keyword form)= %sysfunc(
34        getoption(HSIZE,cm,keyword));
35     %put VSIZE(in/keyword form)= %sysfunc(
36        getoption(VSIZE,in,keyword));
37     %put HSIZE(cm/keyword form)= %sysfunc(
38        getoption(VSIZE,cm,keyword));
39   %mend;
40   goptions VSIZE=8.5 in HSIZE=11 in;
41   options PAGESIZE=67;
42   %showopts
MAPS= ("!sasroot\maps-path\en\maps")
MAPSEXPANDED= ("C:\maps-path\en\maps")
PAGESIZE= 67
PAGESIZESETBY= Options Statement
PAGESIZESCOPE= Line Mode Process
PS= 67
LS= 78
PS(keyword form)= PS=67
LS(keyword form)= LS=78
FORMCHAR= ,ƒ„…†‡ˆ‰Š‹Œ+=|-/\<>*
HSIZE= 11.0000 in
VSIZE= 8.5000 in
HSIZE(in/keyword form)= HSIZE=11.0000 in
HSIZE(cm/keyword form)= HSIZE=27.9400 cm
VSIZE(in/keyword form)= VSIZE=8.5000 in
HSIZE(cm/keyword form)= VSIZE=21.5900 cm
43   proc printto; run;
```

Example 3: Returning Default and Start-up Values

This example changes the value of the PAPERSIZE system option to a specific value, the PAPERSIZE option default value, and to the value that was assigned to the PAPERSIZE option when SAS started.

```
   /* Check the value of papersize before we change it.       */
```

```
/* The initial value is A4 as this value was used when         */
/* SAS started.                                                */

   %put %sysfunc(getoption(papersize,keyword));

/* Change the PAPERSIZE value and check the change.            */

   options papersize="600x800 Pixels";

   %put %sysfunc(getoption(papersize,keyword));

/* Change PAPERSIZE back to the default value and check it.    */
/* RESULT:  LETTER                                             */

   %let defsize = %sysfunc(getoption(papersize,keyword,defaultvalue)) ;
   options &defsize; run;
   %put %sysfunc(getoption(papersize,keyword));

/* Change the value to the startup value and check it.         */
/* RESULT:  A4                                                 */

   %let defsize = %sysfunc(getoption(papersize,keyword,startupvalue)) ;
   options &defsize; run;
   %put %sysfunc(getoption(papersize,keyword));
```

The SAS log displays the following lines:

```
22   /* Check the value of papersize before we change it.       */
23   /* The initial value is A4 as this value was used when         */
24   /* SAS started.                                                */
25
26      %put %sysfunc(getoption(papersize,keyword));
PAPERSIZE=A4
27
28   /* Change the PAPERSIZE value and check the change.            */
29
30      options papersize="600x800 Pixels";
31
32      %put %sysfunc(getoption(papersize,keyword));
PAPERSIZE=600X800 PIXELS
33
34   /* Change PAPERSIZE back to the default value and check it.    */
35   /* RESULT:  LETTER                                             */
36
37      %let defsize = %sysfunc(getoption(papersize,keyword,defaultvalue)) ;
38      options &defsize; run;
39      %put %sysfunc(getoption(papersize,keyword));
PAPERSIZE=LETTER
40
41   /* Change the value to the startup value and check it.         */
42   /* RESULT:  A4                                                 */
43
44      %let defsize = %sysfunc(getoption(papersize,keyword,startupvalue)) ;
45      options &defsize; run;
46      %put %sysfunc(getoption(papersize,keyword));
PAPERSIZE=A4
```

Note: The default settings for the PAGESIZE= and the LINESIZE= options depend on the mode that you use to run SAS.

GETVARC Function

Returns the value of a SAS data set character variable.

Category: SAS File I/O

Syntax

GETVARC(*data-set-id*,*var-num*)

Required Arguments

data-set-id
is a numeric constant, variable, or expression that specifies the data set identifier that the OPEN function returns.

var-num
is a numeric constant, variable, or expression that specifies the number of the variable in the Data Set Data Vector (DDV).

Tips You can obtain this value by using the VARNUM function.

This value is listed next to the variable when you use the CONTENTS procedure.

Details

Use VARNUM to obtain the number of a variable in a SAS data set. VARNUM can be nested or it can be assigned to a variable that can then be passed as the second argument, as shown in the following examples. GETVARC reads the value of a character variable from the current observation in the Data Set Data Vector (DDV) into a macro or DATA step variable.

Example

- This example opens the SASUSER.HOUSES data set and gets the entire tenth observation. The data set identifier value for the open data set is stored in the macro variable MYDATAID. This example nests VARNUM to return the position of the variable in the DDV, and reads in the value of the character variable STYLE.

```
%let mydataid=%sysfunc(open
                      (sasuser.houses,i));
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let style=%sysfunc(getvarc(&mydataid,
                  %sysfunc(varnum
                  (&mydataid,STYLE))));
%let rc=%sysfunc(close(&mydataid));
```

- This example assigns VARNUM to a variable that can then be passed as the second argument. This example fetches data from observation 10.

```
%let namenum=%sysfunc(varnum(&mydataid,NAME));
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let user=%sysfunc(getvarc
                  (&mydataid,&namenum));
```

See Also

Functions:

- “FETCH Function” on page 405
- “FETCHOBS Function” on page 406
- “GETVARN Function” on page 522
- “VARNUM Function” on page 940

GETVARN Function

Returns the value of a SAS data set numeric variable.

Category: SAS File I/O

Syntax

GETVARN(*data-set-id*,*var-num*)

Required Arguments

data-set-id
is a numeric constant, variable, or expression that specifies the data set identifier that the OPEN function returns.

var-num
is a numeric constant, variable, or expression that specifies the number of the variable in the Data Set Data Vector (DDV).

Tips You can obtain this value by using the VARNUM function.

This value is listed next to the variable when you use the CONTENTS procedure.

Details

Use VARNUM to obtain the number of a variable in a SAS data set. You can nest VARNUM or you can assign it to a variable that can then be passed as the second argument, as shown in the "Examples" section. GETVARN reads the value of a numeric variable from the current observation in the Data Set Data Vector (DDV) into a macro variable or DATA step variable.

Example

- This example obtains the entire tenth observation from a SAS data set. The data set must have been previously opened using OPEN. The data set identifier value for the open data set is stored in the variable MYDATAID. This example nests VARNUM, and reads in the value of the numeric variable PRICE from the tenth observation of an open SAS data set.

```
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let price=%sysfunc(getvarn(&mydataid,
```

```
%sysfunc(varnum
     (&mydataid,price))));
```

- This example assigns VARNUM to a variable that can then be passed as the second argument. This example fetches data from observation 10.

```
%let pricenum=%sysfunc(varnum
                       (&mydataid,price));
%let rc=%sysfunc(fetchobs(&mydataid,10));
%let price=%sysfunc(getvarn
                    (&mydataid,&pricenum));
```

See Also

Functions:

- "FETCH Function" on page 405
- "FETCHOBS Function" on page 406
- "GETVARC Function" on page 521
- "VARNUM Function" on page 940

GRAYCODE Function

Generates all subsets of *n* items in a minimal change order.

Category: Combinatorial

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

Syntax

GRAYCODE(*k*, *numeric-variable-1*, ..., *numeric-variable-n*)

GRAYCODE(*k*, *character-variable* <, *n* <, *in-out*> >)

Required Arguments

k

specifies a numeric variable. Initialize *k* to either of the following values before executing the GRAYCODE function:

- a negative number to cause GRAYCODE to initialize the subset to be empty
- the number of items in the initial set indicated by *numeric-variable-1* through *numeric-variable-n*, or *character-variable*, which must be an integer value between 0 and *n* inclusive

The value of *k* is updated when GRAYCODE is executed. The value that is returned is the number of items in the subset.

numeric-variable

specifies numeric variables that have values of 0 or 1 which are updated when GRAYCODE is executed. A value of 1 for *numeric-variable-j* indicates that the j^{th}

item is in the subset. A value of 0 for *numeric-variable-j* indicates that the j^{th} item is not in the subset.

If you assign a negative value to *k* before you execute GRAYCODE, then you do not need to initialize *numeric-variable-1* through *numeric-variable-n* before executing GRAYCODE unless you want to suppress the note about uninitialized variables.

If you assign a value between 0 and *n* inclusive to *k* before you execute GRAYCODE, then you must initialize *numeric-variable-1* through *numeric-variable-n* to *k* values of 1 and *n-k* values of 0.

character-variable
specifies a character variable that has a length of at least *n* characters. The first *n* characters indicate which items are in the subset. By default, an "I" in the j^{th} position indicates that the j^{th} item is in the subset, and an "O" in the j^{th} position indicates that the j^{th} item is out of the subset. You can change the two characters by specifying the *in-out* argument.

If you assign a negative value to *k* before you execute GRAYCODE, then you do not need to initialize *character-variable* before executing GRAYCODE unless you want to suppress the note about an uninitialized variable.

If you assign a value between 0 and *n* inclusive to *k* before you execute GRAYCODE, then you must initialize *character-variable* to *k* characters that indicate an item is in the subset, and *n-k* characters that indicate an item is out of the subset.

Optional Arguments

n
specifies a numeric constant, variable, or expression. By default, *n* is the length of *character-variable*.

in-out
specifies a character constant, variable, or expression. The default value is "IO." The first character is used to indicate that an item is in the subset. The second character is used to indicate that an item is out of the subset.

Details

When you execute GRAYCODE with a negative value of *k*, the subset is initialized to be empty. The GRAYCODE function returns zero.

When you execute GRAYCODE with an integer value of *k* between 0 and *n* inclusive, one item is either added to the subset or removed from the subset, and the value of *k* is updated to equal the number of items in the subset. If the j^{th} item is added to the subset or removed from the subset, the GRAYCODE function returns *j*.

To generate all subsets of *n* items, you can initialize *k* to a negative value and execute GRAYCODE in a loop that iterates 2**n* times. If you want to start with a non-empty subset, then initialize *k* to be the number of items in the subset, initialize the other arguments to specify the desired initial subset, and execute GRAYCODE in a loop that iterates 2**n*-1 times. The sequence of subsets that are generated by GRAYCODE is cyclical, so you can begin with any subset that you want.

Examples

Example 1: Using n=4 Numeric Variables and Negative Initial k

The following program uses numeric variables to generate subsets in a minimal change order.

```
data _null_;
   array x[4];
   n=dim(x);
   k=-1;
   nsubs=2**n;
   do i=1 to nsubs;
      rc=graycode(k, of x[*]);
      put i 5. +3 k= ' x=' x[*] +3 rc=;
   end;
run;
```

SAS writes the following output to the log:

```
    1   k=0  x=0 0 0 0    rc=0
    2   k=1  x=1 0 0 0    rc=1
    3   k=2  x=1 1 0 0    rc=2
    4   k=1  x=0 1 0 0    rc=1
    5   k=2  x=0 1 1 0    rc=3
    6   k=3  x=1 1 1 0    rc=1
    7   k=2  x=1 0 1 0    rc=2
    8   k=1  x=0 0 1 0    rc=1
    9   k=2  x=0 0 1 1    rc=4
   10   k=3  x=1 0 1 1    rc=1
   11   k=4  x=1 1 1 1    rc=2
   12   k=3  x=0 1 1 1    rc=1
   13   k=2  x=0 1 0 1    rc=3
   14   k=3  x=1 1 0 1    rc=1
   15   k=2  x=1 0 0 1    rc=2
   16   k=1  x=0 0 0 1    rc=1
```

Example 2: Using a Character Variable and Positive Initial k

The following example uses a character variable to generate subsets in a minimal change order.

```
data _null_;
   x='++++';
   n=length(x);
   k=countc(x, '+');
   put '    1' +3 k= +2 x=;
   nsubs=2**n;
   do i=2 to nsubs;
      rc=graycode(k, x, n, '+-');
      put i 5. +3 k= +2 x= +3 rc=;
   end;
run;
```

SAS writes the following output to the log:

```
    1   k=4   x=++++
    2   k=3   x=-+++    rc=1
    3   k=2   x=-+-+    rc=3
    4   k=3   x=++-+    rc=1
```

```
 5   k=2   x=+--+    rc=2
 6   k=1   x=---+    rc=1
 7   k=0   x=----    rc=4
 8   k=1   x=+---    rc=1
 9   k=2   x=++--    rc=2
10   k=1   x=-+--    rc=1
11   k=2   x=-++-    rc=3
12   k=3   x=+++-    rc=1
13   k=2   x=+-+-    rc=2
14   k=1   x=--+-    rc=1
15   k=2   x=--++    rc=4
16   k=3   x=+-++    rc=1
```

See Also

CALL Routines:

- "CALL GRAYCODE Routine" on page 168

HARMEAN Function

Returns the harmonic mean.

Category: Descriptive Statistics

Syntax

HARMEAN(*argument*<,*argument*,…>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the harmonic mean of the nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

Floating-point arithmetic often produces tiny numerical errors. Some computations that result in zero when exact arithmetic is used might result in a tiny nonzero value when

floating-point arithmetic is used. Therefore, HARMEAN fuzzes the values of arguments that are approximately zero. When the value of one argument is extremely small relative to the largest argument, then the former argument is treated as zero. If you do not want SAS to fuzz the extremely small values, then use the HARMEANZ function.

Comparisons

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

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=harmean(1,2,4,4);`	2
`x2=harmean(.,4,12,24);`	8
`x3=harmean(of x1-x2);`	3.2

See Also

Functions:

- “GEOMEAN Function” on page 511
- “GEOMEANZ Function” on page 513
- “HARMEANZ Function” on page 527
- “MEAN Function” on page 658

HARMEANZ Function

Returns the harmonic mean, using zero fuzzing.

Category: Descriptive Statistics

Syntax

HARMEANZ(*argument*<,*argument*,…>)

Required Argument

argument
is a nonnegative numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

If any argument is negative, then the result is a missing value. A message appears in the log that the negative argument is invalid, and _ERROR_ is set to 1. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the harmonic mean of the nonmissing values.

If any argument is zero, then the harmonic mean is zero. Otherwise, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals of the values.

Let n be the number of arguments with nonmissing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The harmonic mean is

$$\frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \ldots + \frac{1}{x_n}}$$

Comparisons

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

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=harmeanz(1,2,4,4);`	`2`
`x2=harmeanz(.,4,12,24);`	`8`
`x3=harmeanz(of x1-x2);`	`3.2`

See Also

Functions:

- “GEOMEAN Function” on page 511
- “GEOMEANZ Function” on page 513
- “HARMEAN Function” on page 526
- “MEAN Function” on page 658

HBOUND Function

Returns the upper bound of an array.

Category: Array

Syntax

HBOUND<*n*> (*array-name*)

HBOUND(*array-name*,*bound-n*)

Required Arguments

array-name
is the name of an array that was defined previously in the same DATA step.

bound-n
is a numeric constant, variable, or expression that specifies the dimension for which you want to know the upper bound. Use *bound-n* only if *n* is not specified.

Optional Argument

n
is an integer constant that specifies the dimension for which you want to know the upper bound. If no *n* value is specified, the HBOUND function returns the upper bound of the first dimension of the array.

Details

The HBOUND function returns the upper bound of a one-dimensional array or the upper bound of a specified dimension of a multidimensional array. Use HBOUND in array processing to avoid changing the upper bound of an iterative DO group each time you change the bounds of the array. HBOUND and LBOUND can be used together to return the values of the upper and lower bounds of an array dimension.

Comparisons

- HBOUND returns the literal value of the upper bound of an array dimension.
- DIM always returns a total count of the number of elements in an array dimension.

  Note: This distinction is important when the lower bound of an array dimension has a value other than 1 and the upper bound has a value other than the total number of elements in the array dimension.

Examples

Example 1: One-Dimensional Array

In this example, HBOUND returns the upper bound of the dimension, a value of 5. Therefore, SAS repeats the statements in the DO loop five times.

```
array big{5} weight sex height state city;
do i=1 to hbound(big5);
   more SAS statements;
end;
```

Example 2: Multidimensional Array

This example shows two ways of specifying the HBOUND function for multidimensional arrays. Both methods return the same value for HBOUND, as shown in the table that follows the SAS code example.

```
array mult{2:6,4:13,2} mult1-mult100;
```

Syntax	Alternative Syntax	Value
HBOUND(MULT)	HBOUND(MULT,1)	6
HBOUND2(MULT)	HBOUND(MULT,2)	13
HBOUND3(MULT)	HBOUND(MULT,3)	2

See Also

Functions:

- "DIM Function" on page 377
- "LBOUND Function" on page 612

Statements:

- "ARRAY Statement" in *SAS Statements: Reference*
- "Array Reference Statement" in *SAS Statements: Reference*
- "Array Processing" in Chapter 23 of *SAS Language Reference: Concepts*

HMS Function

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

Category: Date and Time

Syntax

HMS(*hour*,*minute*,*second*)

Required Arguments

hour
is numeric.

minute
is numeric.

second
is numeric.

Details

The HMS function returns a positive numeric value that represents a SAS time value.

Example

The following SAS statements produce these results:

SAS Statement	Result
hrid=hms(12,45,10); put hrid / hrid time.;	 45910 12:45:10

See Also

Functions:

- “DHMS Function” on page 373
- “HOUR Function” on page 534
- “MINUTE Function” on page 661
- “SECOND Function” on page 859

HOLIDAY Function

Returns a SAS date value of a specified holiday for a specified year.

Category: Date and Time

Syntax

HOLIDAY('*holiday*', *year*)

Required Arguments

'*holiday*'
is a character constant, variable, or expression that specifies one of the values listed in the following table.

Values for *holiday* can be in uppercase or lowercase.

Table 2.1 *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

Holiday Value	Description	Date Celebrated
FATHERS	Father's Day	3rd Sunday in June
HALLOWEEN	Halloween	October 31
LABOR	Labor Day	1st Monday in September
MLK	Martin Luther King, Jr. 's birthday	3rd Monday in January beginning in 1986
MEMORIAL	Memorial Day	last Monday in May (since 1971)
MOTHERS	Mother's Day	2nd Sunday in May
NEWYEAR	New Year's Day	January 1
THANKSGIVING	U.S. Thanksgiving Day	4th Thursday in November
THANKSGIVINGCANADA	Canadian Thanksgiving Day	2nd Monday in October
USINDEPENDENCE	U.S. Independence Day	July 4
USPRESIDENTS	Abraham Lincoln's and George Washington's birthdays observed	3rd Monday in February (since 1971)
VALENTINES	Valentine's Day	February 14
VETERANS	Veterans Day	November 11
VETERANSUSG	Veterans Day - U.S. government-observed	U.S. government-observed date for Monday–Friday schedule
VETERANSUSPS	Veterans Day - U.S. post office observed	U.S. government-observed date for Monday–Saturday schedule (U.S. Post Office)
VICTORIA	Victoria Day	Monday on or preceding May 24

year
is a numeric constant, variable, or expression that specifies a four-digit year. If you use a two-digit year, then you must specify the YEARCUTOFF= system option.

Details

The HOLIDAY function computes the date on which a specific holiday occurs in a specified year. Only certain common U.S. and Canadian holidays are defined for use with this function. (See Table 2.1 on page 531 for a list of valid holidays.)

The definition of many holidays has changed over the years. In the U.S., Executive Order 11582, issued on February 11, 1971, fixed the observance of many U.S. federal holidays.

The current holiday definition is extended indefinitely into the past and future, although many holidays have a fixed date at which they were established. Some holidays have not had a consistent definition in the past.

The HOLIDAY function returns a SAS date value. To convert the SAS date value to a calendar date, use any valid SAS date format, such as the DATE9. format.

Comparisons

In some cases, the HOLIDAY function and the NWKDOM function return the same result. For example, the statement **`HOLIDAY('THANKSGIVING', 2007);`** returns the same value as **`NWKDOM(4, 5, 11, 2007);`**.

In other cases, the HOLIDAY function and the MDY function return the same result. For example, the statement **`HOLIDAY('CHRISTMAS', 2007);`** returns the same value as **`MDY(12, 25, 2007);`**.

Example

The following SAS statements produce these results.

SAS Statement	Result

`thanks = holiday('thanksgiving', 2007);`
`format thanks date9.;`
`put thanks;`
	`22NOV2007`

`boxing = holiday('boxing', 2007);`
`format boxing date9.;`
`put boxing;`
	`26DEC2007`

`easter = holiday('easter', 2007);`
`format easter date9.;`
`put easter;`
	`08APR2007`

`canada = holiday('canada', 2007);`
`format canada date9.;`
`put canada;`
	`01JUL2007`

`fathers = holiday('fathers', 2007);`
`format fathers date9.;`
`put fathers;`
	`17JUN2007`

`valentines = holiday('valentines', 2007);`
`format valentines date9.;`
`put valentines;`
	`14FEB2007`

SAS Statement	Result

`victoria = holiday('victoria', 2007);`
`format victoria date9.;`
`put victoria;`
	`21MAY2007`

See Also

Functions:

- "NWKDOM Function" on page 713
- "MDY Function" on page 657

HOUR Function

Returns the hour from a SAS time or datetime value.

Category: Date and Time

Syntax

HOUR(*time* | *datetime* **)**

Required Arguments

time
is a numeric constant, variable, or expression that specifies a SAS time value.

datetime
is a numeric constant, variable, or expression that specifies a SAS datetime value.

Details

The HOUR function returns a numeric value that represents the hour from a SAS time or datetime value. Numeric values can range from 0 through 23. HOUR always returns a positive number.

Example

The following SAS statements produce these results.

SAS Statement	Result

`now='1:30't;`
`h=hour(now);`
`put h;`
	`1`

See Also

Functions:

- "SECOND Function" on page 859

HTMLDECODE Function

Decodes a string that contains HTML numeric character references or HTML character entity references, and returns the decoded string.

Category: Web Tools

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

Syntax

HTMLDECODE(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression.

Details

The HTMLDECODE function recognizes the following character entity references:

Character Entity Reference	Decoded Character
&amp;	&
&lt;	<
&gt;	>
&quot;	"
&apos;	'

Unrecognized entities (&<name>;) are left unmodified in the output string.

The HTMLDECODE function recognizes numeric entity references that are of the form

&#*nnn*;
where *nnn* specifies a decimal number that contains one or more digits.

&#X*nnn*;
where *nnn* specifies a hexadecimal number that contains one or more digits.

Note: Numeric character references that cannot be represented in the current SAS session encoding will not be decoded. The reference will be copied unchanged to the output string.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=htmldecode('not a <tag>');`	`not a <tag>`
`x2=htmldecode('&');`	`'&'`
`x3=htmldecode ('&#65;&#66;&#67;');`	`'ABC'`

See Also

Functions:

- "HTMLENCODE Function" on page 536

HTMLENCODE Function

Encodes characters using HTML character entity references, and returns the encoded string.

Category: Web Tools

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

Syntax

HTMLENCODE(*expression*, <*options*>)

Required Argument

expression
specifies a character constant, variable, or expression. By default, any greater-than (>), less-than (<), and ampersand (&) characters are encoded as **`>`**, **`<`**, and **`&`**, respectively. In SAS 9 only, this behavior can be modified with the *options* argument.

Note: The encoded string can be longer than the output string. You should take the additional length into consideration when you define your output variable. If the encoded string exceeds the maximum length that is defined, the output string might be truncated.

Optional Argument

options
is a character constant, variable, or expression that specifies the type of characters to encode. If you use more than one option, separate the options by spaces. The following options are available:

Option	Character	Character Entity Reference	Description
amp	&	&amp;	The HTMLENCODE function encodes these characters by default. If you need to encode these characters only, then you do not need to specify the options argument. However, if you specify any value for the options argument, then the defaults are overridden, and you must explicitly specify the options for all of the characters that you want to encode.
gt	>	&gt;	
lt	<	&lt;	
apos	'	&apos;	Use this option to encode the apostrophe (') character in text that is used in an HTML or XML tag attribute.
quot	"	&quot;	Use this option to encode the double quotation mark (") character in text that is used in an HTML or XML tag attribute.
7bit	any character that is not represented in 7-bit ASCII encoding	&#x*nnn*; (Unicode)	*nnn* is a one or more digit hexadecimal number. Encode these characters to create HTML or XML that is easily transferred through communication paths that might support only 7-bit ASCII encodings (for example, ftp or e-mail).

Example

The following SAS statements produce these results.

SAS Statement	Result
`htmlencode("John's test <tag>")`	`John's test <tag>`
`htmlencode("John's test <tag>",'apos')`	`John&apos;s test <tag>`
`htmlencode('John "Jon" Smith <tag>','quot')`	`John &quot;Jon&quot; Smith <tag>`
`htmlencode("'A&B&C'",'amp lt gt apos')`	`&apos;A&amp;B&amp;C&apos;`
`htmlencode('80'x, '7bit')` ('80'x is the euro symbol in Western European locales.)	`&#x20AC;` (20AC is the Unicode code point for the euro symbol.)

See Also

Functions:

- “HTMLDECODE Function” on page 535

IBESSEL Function

Returns the value of the modified Bessel function.

Category: Mathematical

Syntax

IBESSEL(*nu*,*x*,*kode*)

Required Arguments

nu
specifies a numeric constant, variable, or expression.

Range $nu \geq 0$

x
specifies a numeric constant, variable, or expression.

Range $x \geq 0$

kode
is a numeric constant, variable, or expression that specifies a nonnegative integer.

Details

The IBESSEL function returns the value of the modified Bessel function of order *nu* evaluated at *x* (Abramowitz, Stegun 1964; Amos, Daniel, Weston 1977). When *kode* equals 0, the Bessel function is returned. Otherwise, the value of the following function is returned:

$$\varepsilon^{-x} I_{nm}(x)$$

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=ibessel(2,2,0);`	`0.6889484477`
`x=ibessel(2,2,1);`	`0.0932390333`

IFC Function

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

Category: Character

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

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.

Example

In the following example, IFC evaluates the expression **grade>80** to implement the logic that determines the performance of several members on a team. The results are written to the SAS log.

```
data _null_;
   input name $ grade;
   performance = ifc(grade>80, 'Pass             ', 'Needs Improvement');
   put name= performance=;
   datalines;
John 74
Kareem 89
Kati 100
Maria 92
;
run;
```

Log 2.10 *Partial SAS Log: IFC Function*

```
name=John performance=Needs Improvement
name=Kareem performance=Pass
name=Kati performance=Pass
name=Maria performance=Pass
```

This example uses an IF/THEN/ELSE construct to generate the same output that is generated by the IFC function. The results are written to the SAS log.

```
data _null_;
   input name $ grade;
   if grade>80 then performance='Pass             ';
      else performance = 'Needs Improvement';
   put name= performance=;
   datalines;
John 74
Sam 89
Kati 100
Maria 92
;
run;
```

Log 2.11 *Partial SAS Log: IF/THEN/ELSE Construct*

```
name=John performance=Needs Improvement
name=Sam performance=Pass
name=Kati performance=Pass
name=Maria performance=Pass
```

See Also

Functions:

- “IFN Function” on page 541

IFN Function

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

Category: Numeric

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

Syntax

IFN(*logical-expression*, *value-returned-when-true*, *value-returned-when-false* <,*value-returned-when-missing*>)

Required Arguments

logical-expression
specifies a numeric constant, variable, or expression.

value-returned-when-true
specifies a numeric constant, variable, or expression that is returned when the value of *logical-expression* is true.

value-returned-when-false
specifies a numeric constant, variable, or expression that is returned when the value of *logical-expression* is false.

Optional Argument

value-returned-when-missing
specifies a numeric constant, variable or expression that is returned when the value of *logical-expression* is missing.

Details

The IFN function uses conditional logic that enables you to select among several values based on the value of a logical expression.

IFN evaluates the first argument, then *logical-expression*. If *logical-expression* is true (that is, not zero and not missing), then IFN returns the value in the second argument. If *logical-expression* is a missing value, and you have a fourth argument, then IFN returns the value in the fourth argument. Otherwise, if *logical-expression* is false, IFN returns the value in the third argument.

The IFN function, an IF/THEN/ELSE construct, or a WHERE statement can produce the same results. (See examples.) However, the IFN function is useful in DATA step expressions when it is not convenient or possible to use an IF/THEN/ELSE construct or a WHERE statement.

Comparisons

The IFN function is similar to the IFC function, except that IFN returns a numeric value whereas IFC returns a character value.

Examples

Example 1: Calculating Commission Using the IFN Function

In the following example, IFN evaluates the expression **TotalSales > 10000**. If total sales exceeds $10,000, then the sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales.

```
data _null_;
   input TotalSales;
   commission=ifn(TotalSales > 10000, TotalSales*.05, TotalSales*.02);
   put commission=;
   datalines;
25000
10000
500
10300
;
run;
```

SAS writes the following output to the log:

```
commission=1250
commission=200
commission=10
commission=515
```

Example 2: Calculating Commission Using an IF/THEN/ELSE Construct

In the following example, an IF/THEN/ELSE construct evaluates the expression **TotalSales > 10000**. If total sales exceeds $10,000, then the sales commission is 5% of the total sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales.

```
data _null_;
   input TotalSales;
   if TotalSales > 10000 then commission = .05 * TotalSales;
      else commission = .02 * TotalSales;
   put commission=;
   datalines;
25000
10000
500
10300
;
run;
```

SAS writes the following output to the log:

```
commission=1250
commission=200
commission=10
commission=515
```

Example 3: Calculating Commission Using a WHERE Statement

In the following example, a WHERE statement evaluates the expression **TotalSales > 10000**. If total sales exceeds $10,000, then the sales commission is 5% of the total

sales. If total sales is less than $10,000, then the sales commission is 2% of the total sales. The output shows only those salespeople whose total sales exceed $10,000.

```
data sales;
   input SalesPerson $ TotalSales;
   datalines;
Michaels 25000
Janowski 10000
Chen 500
Gupta 10300
;
data commission;
   set sales;
   where TotalSales > 10000;
   commission = TotalSales * .05;
run;
proc print data=commission;
   title 'Commission for Total Sales > 1000';
run;
```

Display 2.34 *Output from a WHERE Statement*

Commission for Total Sales > 1000

Obs	SalesPerson	TotalSales	commission
1	Michaels	25000	1250
2	Gupta	10300	515

See Also

Functions:

- "IFC Function" on page 538

INDEX Function

Searches a character expression for a string of characters, and returns the position of the string's first character for the first occurrence of the string.

Category: Character

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

Tip: DBCS equivalent function is KINDEX in *SAS National Language Support (NLS): Reference Guide*. See "DBCS Compatibility" on page 544 .

Syntax

INDEX(*source*,*excerpt*)

Required Arguments

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

excerpt
is a character constant, variable, or expression that specifies the string of characters to search for in *source*.

Tips Enclose a literal string of characters in quotation marks.

Both leading and trailing spaces are considered part of the *excerpt* argument. To remove trailing spaces, include the TRIM function with the *excerpt* variable inside the INDEX function.

Details

The Basics

The INDEX function searches *source*, from left to right, for the first occurrence of the string specified in *excerpt*, and returns the position in *source* of the string's first character. If the string is not found in *source*, INDEX returns a value of 0. If there are multiple occurrences of the string, INDEX returns only the position of the first occurrence.

DBCS Compatibility

The DBCS equivalent function is KINDEX, which is documented in *SAS National Language Support (NLS): Reference Guide*. However, there is a minor difference in the way trailing blanks are handled. In KINDEX, multiple blanks in the second argument match a single blank in the first argument. The following example shows the differences between the two functions:

```
index('ABC,DE F(X=Y)','       ')            => 0
kindex('ABC,DE F(X=Y)','       ')          => 7
```

Examples

Example 1: Finding the Position of a Variable in the Source String

The following example finds the first position of the *excerpt* argument in *source*.

```
data _null_;
   a = 'ABC.DEF(X=Y)';
   b = 'X=Y';
   x = index(a,b);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=9
```

Example 2: Removing Trailing Spaces When You Use the INDEX Function with the TRIM Function

The following example shows the results when you use the INDEX function with and without the TRIM function. If you use INDEX without the TRIM function, leading and trailing spaces are considered part of the *excerpt* argument. If you use INDEX with the

TRIM function, TRIM removes trailing spaces from the *excerpt* argument as you can see in this example. Note that the TRIM function is used inside the INDEX function.

```
options nodate nostimer ls=78 ps=60;
data _null_;
   length a b $14;
   a='ABC.DEF (X=Y)';
   b='X=Y';
   q=index(a,b);
   w=index(a,trim(b));
   put q= w=;
run;
```

SAS writes the following output to the log:

```
q=0 w=10
```

See Also

Functions:

- "FIND Function" on page 457
- "INDEXC Function" on page 545
- "INDEXW Function" on page 546

INDEXC Function

Searches a character expression for any of the specified characters, and returns the position of that character.

Category: Character

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

Tip: DBCS equivalent function is KINDEXC in *SAS National Language Support (NLS): Reference Guide*.

Syntax

INDEXC(*source*,*excerpt-1*<,… *excerpt-n*>)

Required Arguments

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

excerpt
specifies the character constant, variable, or expression to search for in *source*.

Tip If you specify more than one excerpt, separate them with a comma.

Details

The INDEXC function searches *source*, from left to right, for the first occurrence of any character present in the excerpts and returns the position in *source* of that character. If none of the characters in *excerpt-1* through *excerpt-n* in *source* are found, INDEXC returns a value of 0.

Comparisons

The INDEXC function searches for the first occurrence of any individual character that is present within the character string, whereas the INDEX function searches for the first occurrence of the character string as a substring. The FINDC function provides more options.

Example

The following SAS statements produce these results.

SAS Statement	Result

`a='ABC.DEP (X2=Y1)';`
`x=indexc(a,'0123',';()=.');`
`put x;`
	4

`b='have a good day';`
`x=indexc(b,'pleasant','very');`
`put x;`
	2

See Also

Functions:

- “FINDC Function” on page 459
- “INDEX Function” on page 543
- “INDEXW Function” on page 546

INDEXW Function

Searches a character expression for a string that is specified as a word, and returns the position of the first character in the word.

Category: Character

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

Syntax

INDEXW(*source*,*excerpt*<,*delimiters*>)

Required Arguments

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

excerpt
specifies a character constant, variable, or expression to search for in *source*. SAS removes leading and trailing delimiters from *excerpt*.

Optional Argument

delimiter
specifies a character constant, variable, or expression containing the characters that you want INDEXW to use as delimiters in the character string. The default delimiter is the blank character.

Details

The INDEXW function searches *source*, from left to right, for the first occurrence of *excerpt* and returns the position in *source* of the substring's first character. If the substring is not found in *source*, then INDEXW returns a value of 0. If there are multiple occurrences of the string, then INDEXW returns only the position of the first occurrence.

The substring pattern must begin and end on a word boundary. For INDEXW, word boundaries are delimiters, the beginning of *source*, and the end of *source*. If you use an alternate delimiter, then INDEXW does not recognize the end of the text as the end data.

INDEXW has the following behavior when the second argument contains blank spaces or has a length of 0:

- If both *source* and *excerpt* contain only blank spaces or have a length of 0, then INDEXW returns a value of 1.
- If *excerpt* contains only blank spaces or has a length of 0, and *source* contains character or numeric data, then INDEXW returns a value of 0.

Comparisons

The INDEXW function searches for strings that are words, whereas the INDEX function searches for patterns as separate words or as parts of other words. INDEXC searches for any characters that are present in the excerpts. The FINDW function provides more options.

Examples

Example 1: Table of SAS Examples

The following SAS statements produce these results.

SAS Statement	Result

`s='asdf adog dog';`
`p='dog  ';`
`x=indexw(s,p);`
`put x;`
	`11`

SAS Statement	Result
s='abcdef x=y'; p='def'; x=indexw(s,p); put x;	0
x="abc,def@ xyz"; abc=indexw(x, " abc ", "@"); put abc;	0
x="abc,def@ xyz"; comma=indexw(x, ",", "@"); put comma;	0
x='abc,def% xyz'; def=indexw(x, 'def', '%,'); put def;	5
x="abc,def@ xyz"; at=indexw(x, "@", "@"); put at;	0
x="abc,def@ xyz"; xyz=indexw(x, " xyz", "@"); put xyz;	9
c=indexw(trimn(' '), ' ');	1
g=indexw(' x y ', trimn(' '));	0

Example 2: Using a Semicolon (;) As the Delimiter

The following example shows how to use the semicolon delimiter in a SAS program that also calls the CATX function. A semicolon delimiter must be in place after each call to CATX, and the second argument in the INDEXW function must be trimmed or searches will not be successful.

```
data temp;
   infile datalines;
   input name $12.;
   datalines;
abcdef
abcdef
;
run;
data temp2;
   set temp;
   format name_list $1024.;
   retain name_list ' ';
   exists=indexw(name_list, trim(name), ';');
   if exists=0 then do
      name_list=catx(';', name_list, name)||';' ;
      name_count +1;
      put '------------------------------';
      put exists= ;
```

```
      put name_list= ;
      put name_count= ;
      end;
    run;
```

Log 2.12 *Output from Using a Semicolon As the Delimiter*

```
------------------------------
exists=0
name_list=abcdef;
name_count=1
```

In this example, the first time CATX is called *name_list* is blank and the value of *name* is 'abcdef'. CATX returns 'abcdef' with no semicolon appended. However, when INDEXW is called the second time, the value of *name_list* is 'abcdef' followed by 1018 (1024–6) blanks, and the value of *name* is 'abcdef' followed by six blanks. Because the third argument in INDEXW is a semicolon (;), the blanks are significant and do not denote a word boundary. Therefore, the second argument cannot be found in the first argument.

If the example has no blanks, the behavior of INDEXW is easier to understand. In the following example, we expect the value of *x* to be 0 because the complete word ABCDE was not found in the first argument:

```
x = indexw('ABCDEF;XYZ', 'ABCDE', ';');
```

The only values for the second argument that would return a nonzero result are ABCDEF and XYZ.

Example 3: Using a Space As the Delimiter

The following example uses a space as a delimiter:

```
data temp;
   infile datalines;
   input name $12.;
   datalines;
abcdef
abcdef
;
run;
data temp2;
   set temp;
   format name_list $1024.;
   retain name_list ' ';
   exists=indexw(name_list, name, ' ');
   if exists=0 then do
      name_list=catx(' ', name_list, name) ;
      name_count +1;
      put '------------------------------';
      put exists= ;
      put name_list= ;
      put name_count= ;
      end;
run;
```

Log 2.13 *Output from Using a Space as the Delimiter*

```
------------------------------
exists=0
name_list=abcdef
name_count=1
```

See Also

Functions:

- "FINDW Function" on page 467
- "INDEX Function" on page 543
- "INDEXC Function" on page 545

INPUT Function

Returns the value that is produced when SAS converts an expression using the specified informat.

Category: Special

Syntax

INPUT(*source*,<? | ??>,*informat*.)

Required Arguments

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

? or ??
specifies the optional question mark (?) and double question mark (??) modifiers that suppress the printing of both the error messages and the input lines when invalid data values are read. The ? modifier suppresses the invalid data message. The ?? modifier also suppresses the invalid data message and, in addition, prevents the automatic variable _ERROR_ from being set to 1 when invalid data are read.

informat.
is the SAS informat that you want to apply to the source. This argument must be the name of an informat followed by a period, and cannot be a character constant, variable, or expression.

Details

If the INPUT function returns a character value to a variable that has not yet been assigned a length, by default the variable length is determined by the width of the informat.

The INPUT function enables you to convert the value of *source* by using a specified informat. The informat determines whether the result is numeric or character. Use INPUT to convert character values to numeric values or other character values.

Comparisons

The INPUT function returns the value produced when a SAS expression is converted using a specified informat. You must use an assignment statement to store that value in a variable. The INPUT statement uses an informat to read a data value. Storing that value in a variable is optional.

The INPUT function requires the informat to be specified as a name followed by a period and optional decimal specification. The INPUTC and INPUTN functions allow the informat to be specified as a character constant, variable, or expression.

Examples

Example 1: Converting Character Values to Numeric Values

This example uses the INPUT function to convert a character value to a numeric value and store it in another variable. The COMMA9. informat reads the value of the SALE variable, stripping the commas. The resulting value, 2115353, is stored in FMTSALE.

```
data testin;
   input sale $9.;
   fmtsale=input(sale,comma9.);
   datalines;
2,115,353
;
```

Example 2: Using PUT and INPUT Functions

In this example, PUT returns a numeric value as a character string. The value 122591 is assigned to the CHARDATE variable. INPUT returns the value of the character string as a SAS date value using a SAS date informat. The value 11681 is stored in the SASDATE variable.

```
numdate=122591;
chardate=put(numdate,z6.);
sasdate=input(chardate,mmddyy6.);
```

Example 3: Suppressing Error Messages

In this example, the question mark (?) modifier tells SAS not to print the invalid data error message if it finds data errors. The automatic variable _ERROR_ is set to 1 and input data lines are written to the SAS log.

```
y=input(x,? 3.1);
```

Because the double question mark (??) modifier suppresses printing of error messages and input lines and prevents the automatic variable _ERROR_ from being set to 1 when invalid data are read, the following two examples produce the same result:

- **`y=input(x,?? 2.);`**
- **`y=input(x,? 2.); _error_=0;`**

See Also

Functions:

- “INPUTC Function” on page 552
- “INPUTN Function” on page 554

- “PUT Function” on page 791
- “PUTC Function” on page 793
- “PUTN Function” on page 795

Statements:

- “INPUT Statement” in *SAS Statements: Reference*

INPUTC Function

Enables you to specify a character informat at run time.

Category: Special

Syntax

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

Required Arguments

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

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

Optional Argument

w
is a numeric constant, variable, or expression that specifies a width to apply to the informat.

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

Details

If the INPUTC function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the length of the first argument.

Comparisons

The INPUTN function enables you to specify a numeric informat at run time. Using the INPUT function is faster because you specify the informat at compile time.

Example

This example shows how to specify character informats. The PROC FORMAT step in this example creates a format, TYPEFMT., that formats the variable values 1, 2, and 3 with the name of one of the three informats that this step also creates. The informats store responses of "positive," "negative," and "neutral" as different words, depending on

the type of question. After PROC FORMAT creates the format and informats, the DATA step creates a SAS data set from raw data consisting of a number identifying the type of question and a response. After reading a record, the DATA step uses the value of TYPE to create a variable, RESPINF, that contains the value of the appropriate informat for the current type of question. The DATA step also creates another variable, WORD, whose value is the appropriate word for a response. The INPUTC function assigns the value of WORD based on the type of question and the appropriate informat.

```
proc format;
   value typefmt 1='$groupx'
                 2='$groupy'
                 3='$groupz';
   invalue $groupx 'positive'='agree'
                   'negative'='disagree'
                   'neutral'='notsure';
   invalue $groupy 'positive'='accept'
                   'negative'='reject'
                   'neutral'='possible';
   invalue $groupz 'positive'='pass'
                   'negative'='fail'
                   'neutral'='retest';
run;
data answers;
   input type response $;
   respinformat = put(type, typefmt.);
   word = inputc(response, respinformat);
   datalines;
1 positive
1 negative
1 neutral
2 positive
2 negative
2 neutral
3 positive
3 negative
3 neutral
;
```

The value of WORD for the first observation is **agree**. The value of WORD for the last observation is **retest**.

See Also

Functions:

- “INPUT Function” on page 550
- “INPUTN Function” on page 554
- “PUT Function” on page 791
- “PUTC Function” on page 793
- “PUTN Function” on page 795

INPUTN Function

Enables you to specify a numeric informat at run time.

Category: Special

Syntax

INPUTN(*source*, *informat*<,*w*<,*d*> >)

Required Arguments

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

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

Optional Arguments

w
is a numeric constant, variable, or expression that specifies a width to apply to the informat.

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

d
is a numeric constant, variable, or expression that specifies the number of decimal places to use.

Interaction If you specify a number here, it overrides any decimal-place specification in the informat.

Comparisons

The INPUTC function enables you to specify a character informat at run time. Using the INPUT function is faster because you specify the informat at compile time.

Example

This example shows how to specify numeric informats. The PROC FORMAT step in this example creates a format, READDATE., that formats the variable values 1 and 2 with the name of a SAS date informat. The DATA step creates a SAS data set from raw data originally from two different sources (indicated by the value of the variable SOURCE). Each source specified dates differently. After reading a record, the DATA step uses the value of SOURCE to create a variable, DATEINF, that contains the value of the appropriate informat for reading the date. The DATA step also creates a new variable, NEWDATE, whose value is a SAS date. The INPUTN function assigns the value of NEWDATE based on the source of the observation and the appropriate informat.

```
proc format;
   value readdate 1='date7.'
                  2='mmddyy8.';
run;
options yearcutoff=1920;
data fixdates (drop=start dateinformat);
   length jobdesc $12;
   input source id lname $ jobdesc $ start $;
   dateinformat=put(source, readdate.);
   newdate = inputn(start, dateinformat);
   datalines;
1 1604 Ziminski writer 09aug90
1 2010 Clavell editor 26jan95
2 1833 Rivera writer 10/25/92
2 2222 Barnes proofreader 3/26/98
;
```

See Also

Functions:

- "INPUT Function" on page 550
- "INPUTC Function" on page 552
- "PUT Function" on page 791
- "PUTC Function" on page 793
- "PUTN Function" on page 795

INT Function

Returns the integer value, fuzzed to avoid unexpected floating-point results.

Category: Truncation

Syntax

INT(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The INT function returns the integer portion of the argument (truncates the decimal portion). If the argument's value is within 1E-12 of an integer, the function results in that integer. If the value of *argument* is positive, the INT function has the same result as the FLOOR function. If the value of *argument* is negative, the INT function has the same result as the CEIL function.

Comparisons

Unlike the INTZ function, the INT function fuzzes the result. If the argument is within 1E-12 of an integer, the INT function fuzzes the result to be equal to that integer. The INTZ function does not fuzz the result. Therefore, with the INTZ function you might get unexpected results.

Example

The following SAS statements produce these results.

SAS Statement	Result

`var1=2.1;`
`x=int(var1);`
`put x;`
	`2`

`var2=-2.4;`
`y=int(var2);`
`put y;`
	`-2`

`a=int(1+1.e-11);` `put a;`	`1`
`b=int(-1.6);` `put b;`	`-1`

See Also

Functions:

- “CEIL Function” on page 294
- “FLOOR Function” on page 480
- “INTZ Function” on page 596

INTCINDEX Function

Returns the cycle index when a date, time, or datetime interval and value are specified.

Category: Date and Time

Syntax

INTCINDEX(*interval*<<*multiple*.<*shift-index*> > > , *date-time-value*)

Required Arguments

interval
specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or

lowercase. The possible values of *interval* are listed in Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts* .

TIP If *interval* is a character constant, then enclose the value in quotation marks.

TIP Valid values for *interval* depend on whether *date-time-value* is a date, time, or datetime value.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>
The three parts of the interval name are as follows:

interval
specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH intervals shift by MONTH periods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

date-time-value
specifies a date, time, or datetime value that represents a time period of a specified interval.

Details

The INTCINDEX function returns the index of the seasonal cycle when you specify an interval and a SAS date, time, or datetime value. For example, if the interval is MONTH, each observation in the data corresponds to a particular month. Monthly data is

considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12. WEEK is the seasonal cycle for an interval that is equal to DAY. Therefore, **`intcindex('day','01SEP78'd);`** returns a value of 35 because September 1, 1978, is the sixth day of the 35th week of the year. For more information about working with date and time intervals, see "Date and Time Intervals" on page 31.

The INTCINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For a list of these intervals, see "Retail Calendar Intervals: ISO 8601 Compliant" in *SAS Language Reference: Concepts*.

Comparisons

The INTCINDEX function returns the cycle index, whereas the INTINDEX function returns the seasonal index.

In the example **`cycle_index = intcindex('day','04APR2005'd);`**, the INTCINDEX function returns the week of the year. In the example **`index = intindex('day','04APR2005'd);`**, the INTINDEX function returns the day of the week.

In the example **`cycle_index = intcindex('minute','01Sep78:00:00:00'dt);`**, the INTCINDEX function returns the hour of the day. In the example **`index = intindex('minute','01Sep78:00:00:00'dt);`**, the INTINDEX function returns the minute of the hour.

In the example **`intseas(intcycle('interval'));`**, the INTSEAS function returns the maximum number that could be returned by **`intcindex('interval',date);`**.

Example

The following SAS statements produce these results.

SAS Statement	Result
`cycle_index1 = intcindex('day', '01SEP05'd);` `put cycle_index1;`	35
`cycle_index2 = intcindex('dtqtr', '23MAY2005:05:03:01'dt);` `put cycle_index2;`	1
`cycle_index3 = intcindex('tenday', '13DEC2005' d);` `put cycle_index3;`	1
`cycle_index4 = intcindex('minute', '23:13:02't);` `put cycle_index4;`	24

`var1 = 'semimonth';`
`cycle_index5 = intcindex(var1, '05MAY2005:10:54:03'dt);`
`put cycle_index5;`
	1

See Also

Functions:

- “INTINDEX Function” on page 574
- “INTCYCLE Function” on page 565
- “INTSEAS Function” on page 589

INTCK Function

Returns the number of interval boundaries of a given kind that lie between two dates, times, or datetime values.

Category: Date and Time

Syntax

INTCK(*interval*<*multiple*> <.*shift-index*>, *start-date*, *end-date*, <'*method*'>)

INTCK(*custom-interval*, *start-date*, *end-date*, <'*method*'>)

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in Table 7.3, “Intervals Used with Date and Time Functions,” in *SAS Language Reference: Concepts* .

The type of interval (date, datetime, or time) must match the type of value in *start-date*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<*multiple.shift-index*>

The three parts of the interval name are listed below:

interval

specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

multiple

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

See “Incrementing Dates and Times by Using Multipliers and by Shifting Intervals” on page 31 for more information.

custom-interval

specifies a user-defined interval that is defined by a SAS data set. Each observation contains two variables, *begin* and *end*.

Requirement You must use the INTERVALDS system option if you use the *custom-interval* variable.

See “Details” on page 561 for more information about custom intervals.

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

Restrictions The shift index cannot be greater than the number of subperiods in the entire interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, MONTH type intervals shift by MONTH subperiods by default. Thus, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

start-date
specifies a SAS expression that represents the starting SAS date, time, or datetime value.

end-date
specifies a SAS expression that represents the ending SAS date, time, or datetime value.

Optional Argument

'*method*'
specifies that intervals are counted using either a discrete or a continuous method.

You must enclose *method* in quotation marks. *Method* can be one of these values:

CONTINUOUS
specifies that continuous time is measured. The interval is shifted based on the starting date.

The continuous method is useful for calculating anniversaries. For example, you can calculate the number of years married by executing the following program:

```
data b;
   WeddingDay='14feb2000'd;
   Today=today();
   YearsMarried=INTCK('YEAR',WeddingDay,today(),'C');
   format WeddingDay Today date9.;
run;
proc print data=b;
run;
```

The results are WeddingDay=14FEB2000, Today=17NOV2010, and YearsMarried=10.

For the CONTINUOUS method, the distance in months between January 15, 2000, and February 15, 2000, 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, 2000, and February 1, 2000, is one month.

Alias D or DISC

Default DISCRETE

Details

Calendar Interval Calculations

All values within a discrete time interval are interpreted as being equivalent. This means that the dates of January 1, 2005 and January 15, 2005 are equivalent when you specify a monthly interval. Both of these dates represent the interval that begins on January 1, 2005 and ends on January 31, 2005. You can use the date for the beginning of the interval (January 1, 2005) or the date for the end of the interval (January 31, 2005) to identify the interval. These dates represent all of the dates within the monthly interval.

In the example **`intck('qtr','14JAN2005'd,'02SEP2005'd);`**, the *start-date* ('14JAN2005'd) is equivalent to the first quarter of 2005. The *end-date* ('02SEP2005'd) is equivalent to the third quarter of 2005. The interval count, that is, the number of times the beginning of an interval is reached in moving from the *start-date* to the *end-date* is 2.

The INTCK function using the default discrete method counts the number of times the beginning of an interval is reached in moving from the first date to the second. It does not count the number of complete intervals between two dates:

- The function **`INTCK('MONTH','1jan1991'd,'31jan1991'd)`** returns 0, because the two dates are within the same month.
- The function **`INTCK('MONTH','31jan1991'd,'1feb1991'd)`** returns 1, because the two dates lie in different months that are one month apart.
- The function **`INTCK('MONTH','1feb1991'd,'31jan1991'd)`** returns –1 because the first date is in a later discrete interval than the second date. (INTCK returns a negative value whenever the first date is later than the second date and the two dates are not in the same discrete interval.)

Using the discrete method, WEEK intervals are determined by the number of Sundays, the default first day of the week, that occur between the *start-date* and the *end-date*, and not by how many seven-day periods fall between those dates. To count the number of seven-day periods between *start-date* and *end-date*, use the continuous method.

Both the *multiple* and the *shift-index* arguments are optional and default to 1. For example, YEAR, YEAR1, YEAR.1, and YEAR1.1 are all equivalent ways of specifying ordinary calendar years.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 31.

Date and Datetime Intervals

The intervals that you need to use with SAS datetime values are SAS datetime intervals. Datetime intervals are formed by adding the prefix "DT" to any date interval. For example, MONTH is a SAS date interval, and DTMONTH is a SAS datetime interval. Similarly, YEAR is a SAS date interval, and DTYEAR is a SAS datetime interval.

To ensure correct results with interval functions, use date intervals with date values and datetime intervals with datetime values. SAS does not return an error message if you use a date value with a datetime interval, but the results are incorrect.

The following example uses the DTDAY datetime interval and returns the number of days between August 1, 2011, and February 1, 2012:

```
data _null_;
   days=intck('dtday', '01aug2011:00:10:48'dt, '01feb2012:00:10:48'dt);
   put days=;
run;
```

SAS writes the following output to the log:

```
days=184
```

Custom Time Intervals

A custom time interval is defined by a SAS data set. The data set must contain the *begin* variable; it can also contain the *end* and *season* variables. Each observation represents one interval with the *begin* variable containing the start of the interval, and the *end* variable, if present, containing the end of the interval. The intervals must be listed in ascending order. There cannot be gaps between intervals, and intervals cannot overlap.

The SAS system option INTERVALDS= is used to define custom intervals and associate interval data sets with new interval names. The following example shows how to specify the INTERVALDS= system option:

```
options intervalds=(interval=libref.dataset-name);
```

Arguments

interval
specifies the name of an interval. The value of *interval* is the data set that is named in *libref.dataset-name*.

libref.dataset-name
specifies the libref and data set name of the file that contains user-supplied holidays.

For more information, see "Custom Time Intervals" on page 34.

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in Chapter 7 of *SAS Language Reference: Concepts* .

Examples

Example 1: Interval Examples Using INTCK

The following SAS statements produce these results.

SAS Statement	Result
`qtr=intck('qtr','10jan95'd,'01jul95'd);` `put qtr;`	2

`year=intck('year','31dec94'd,`
`'01jan95'd);`
`put year;`
	1

`year=intck('year','01jan94'd,`
`'31dec94'd);`
`put year;`
	0

`semi=intck('semiyear','01jan95'd,`
`'01jan98'd);`
`put semi;`
	6

`weekvar=intck('week2.2','01jan97'd,`
`'31mar97'd);`
`put weekvar;`
	7

`wdvar=intck('weekday7w','01jan97'd,`
`'01feb97'd);`
`put wdvar;`
	26

`y='year';`
`date1='1sep1991'd;`
`date2='1sep2001'd;`
`newyears=intck(y,date1,date2);`
`put newyears;`
	10

`y=trim('year  ');`
`date1='1sep1991'd + 300;`
`date2='1sep2001'd - 300;`
`newyears=intck(y,date1,date2);`
`put newyears;`
	8

In the second example, INTCK returns a value of 1 even though only one day has elapsed. This result is returned because the interval from December 31, 1994, to January 1, 1995, contains the starting point for the YEAR interval. However, in the third example, a value of 0 is returned even though 364 days have elapsed. This result is because the period between January 1, 1994, and December 31, 1994, does not contain the starting point for the interval.

In the fourth example, SAS returns a value of 6 because January 1, 1995, through January 1, 1998, contains six semiyearly intervals. (Note that if the ending date were December 31, 1997, SAS would count five intervals.) In the fifth example, SAS returns a value of 6 because there are six two-week intervals beginning on a first Monday during the period of January 1, 1997, through March 31, 1997. In the sixth example, SAS returns the value 26. That indicates that beginning with January 1, 1997, and counting

only Saturdays as weekend days through February 1, 1997, the period contains 26 weekdays.

In the seventh example, the use of variables for the arguments is illustrated. The use of expressions for the arguments is illustrated in the last example.

Example 2: An Example That Compares Methods

The following example shows different values for *method*:

```
data a;
   interval='month';
   start='14FEB2000'd;
   end='13MAR2000'd;
   months_default=intck(interval, start, end);
   months_discrete=intck(interval, start, end,'d');
   months_continuous=intck(interval, start, end,'c');
   output;

   end='14MAR2000'd;
   months_default=intck(interval, start, end);
   months_discrete=intck(interval, start, end,'d');
   months_continuous=intck(interval, start, end,'c');
   output;

   start='31JAN2000'd;
   end='01FEB2000'd;
   months_default=intck(interval, start, end);
   months_discrete=intck(interval, start, end,'d');
   months_continuous=intck(interval, start, end,'c');
   output;
   format start end date.;
run;

proc print data=a;
run;
```

Display 2.35 *Comparisons among Methods*

The SAS System

Obs	interval	start	end	months_default	months_discrete	months_continuous
1	month	14FEB00	13MAR00	1	1	0
2	month	14FEB00	14MAR00	1	1	1
3	month	31JAN00	01FEB00	1	1	0

See Also

Functions:

- “INTNX Function” on page 580

System Options:

- "INTERVALDS= System Option" in *SAS System Options: Reference*

INTCYCLE Function

Returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified.

Category: Date and Time

Syntax

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

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>

The three parts of the interval name are listed below:

interval

specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

multiple

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

shift-index

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the

shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

Optional Argument

seasonality
specifies a numeric value.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Example In the following example, the function

```
INTCYCLE('MONTH', 3);
```

has a *seasonality* argument and returns the value QTR. The function

```
INTCYCLE('MONTH');
```

does not have a *seasonality* argument and returns the value YEAR.

Details

The Basics

The INTCYCLE function returns the interval of the seasonal cycle, depending on a date, time, or datetime interval. For example, **`INTCYCLE('MONTH');`** returns the value YEAR because the months from January through December constitute a yearly cycle. **`INTCYCLE('DAY');`** returns the value WEEK because the days from Sunday through Saturday constitute a weekly cycle.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for information about multipliers and shift indexes. See "Commonly Used Time Intervals" on page 32 for information about how intervals are calculated.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 31 .

The INTCYCLE function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" on page 34.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTCYCLE function uses the concept of seasonality and returns the date, time, or datetime interval at the next higher seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and using the forecasting methods in PROC FORCAST, see the *SAS/ETS User's Guide*.

Example

The following SAS statements produce these results.

SAS Statement	Result
`cycle_year = intcycle('year');` `put cycle_year;`	YEAR
`cycle_quarter = intcycle('qtr');` `put cycle_quarter;`	YEAR
`cycle_3 = intcycle('month', 3);` `put cycle_3;`	QTR
`cycle_month = intcycle('month');` `put cycle_month;`	YEAR
`cycle_weekday = intcycle('weekday');` `put cycle_weekday;`	WEEK
`cycle_weekday2 = intcycle('weekday', 5);` `put cycle_weekday2;`	WEEK
`cycle_day = intcycle('day');` `put cycle_day;`	WEEK
`cycle_day2 = intcycle('day', 10);` `put cycle_day2;`	TENDAY

`var1 = 'second';`
`cycle_second = intcycle(var1);`
`put cycle_second;`
	DTMINUTE

See Also

Functions:

- “INTSEAS Function” on page 589
- “INTINDEX Function” on page 574
- “INTCINDEX Function” on page 556

Other References:

- *SAS/ETS User's Guide*

INTFIT Function

Returns a time interval that is aligned between two dates.

Category: Date and Time

Syntax

INTFIT(*argument-1*, *argument-2*, '*type*')

Required Arguments

argument
specifies a SAS expression that represents a SAS date or datetime value, or an observation.

Tip Observation numbers are more likely to be used as arguments if date or datetime values are not available.

'*type*'
specifies whether the arguments are SAS date values, datetime values, or observations.

The following values for *type* are valid:

d specifies that *argument-1* and *argument-2* are date values.

dt specifies that *argument-1* and *argument-2* are datetime values.

obs specifies that *argument-1* and *argument-2* are observations.

Details

The INTFIT function returns the most likely time interval based on two dates, datetime values, or observations that have been aligned within an interval. INTFIT assumes that the alignment value is SAME, which specifies that the date is aligned to the same calendar date with the corresponding interval increment. For more information about the *alignment* argument, see "INTNX Function" on page 580.

If the arguments that are used with INTFIT are observations, you can determine the cycle of an occurrence by using observation numbers. In the following example, the first two arguments of INTFIT are observation numbers, and the *type* argument is **obs**. If Jason used the gym the first time and the 25th time that a researcher recorded data, you could determine the interval by using the following statement: **interval=intfit(1,25,'obs');**. In this case, the value of interval is OBS24.2.

For information about time series, see the *SAS/ETS 9.3 User's Guide*.

The INTFIT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in Chapter 7 of *SAS Language Reference: Concepts* .

Examples

Example 1: Finding Intervals That Are Aligned between Two Dates

The following example shows the intervals that are aligned between two dates. The *type* argument in this example identifies the input as date values.

```
data a;
   length interval $20;
   date1='01jan11'd;
   do i=1 to 25;
      date2=intnx('day', date1, i);
      interval=intfit(date1, date2, 'd');
      output;
```

```
   end;
   format date1 date2 date.;
run;
proc print data=a;
run;
```

Display 2.36 *Interval Output from the INTFIT Function*

The SAS System

Obs	interval	date1	i	date2
1	DAY	01JAN11	1	02JAN11
2	DAY2	01JAN11	2	03JAN11
3	DAY3.2	01JAN11	3	04JAN11
4	DAY4	01JAN11	4	05JAN11
5	DAY5.4	01JAN11	5	06JAN11
6	DAY6.5	01JAN11	6	07JAN11
7	WEEK.7	01JAN11	7	08JAN11
8	DAY8.5	01JAN11	8	09JAN11
9	DAY9.8	01JAN11	9	10JAN11
10	TENDAY	01JAN11	10	11JAN11
11	DAY11.6	01JAN11	11	12JAN11
12	DAY12.5	01JAN11	12	13JAN11
13	DAY13.13	01JAN11	13	14JAN11
14	WEEK2.14	01JAN11	14	15JAN11
15	SEMIMON	01JAN11	15	16JAN11
16	DAY16.5	01JAN11	16	17JAN11
17	DAY17.14	01JAN11	17	18JAN11
18	DAY18.17	01JAN11	18	19JAN11
19	DAY19.9	01JAN11	19	20JAN11
20	TENDAY2	01JAN11	20	21JAN11
21	WEEK3.7	01JAN11	21	22JAN11
22	DAY22.17	01JAN11	22	23JAN11
23	DAY23.22	01JAN11	23	24JAN11
24	DAY24.5	01JAN11	24	25JAN11
25	DAY25.4	01JAN11	25	26JAN11

The output shows that if the increment value is one day, then the result of the INTFIT function is DAY. If the increment value is two days, then the result of the INTFIT function is DAY2. If the increment value is three days, then the result is DAY3.2, with a shift index of 3. (If the two input dates are a Friday and a Monday, then the result is WEEKDAY.) If the increment value is seven days, then the result is WEEK.

Example 2: Finding Intervals That Are Aligned between Two Dates When the Dates Are Identified As Observations

The following example shows the intervals that are aligned between two dates. The *type* argument in this example identifies the input as observations.

```
data a;
   length interval $20;
   date1='01jan11'd;
   do i=1 to 25;
```

```
      date2=intnx('day', date1, i);
      interval=intfit(date1, date2, 'obs');
      output;
   end;
   format date1 date2 date.;
run;
proc print data=a;
run;
```

Display 2.37 *Interval Output from the INTFIT Function When Dates Are Identified as Observations*

The SAS System

Obs	interval	date1	i	date2
1	OBS	01JAN11	1	02JAN11
2	OBS2	01JAN11	2	03JAN11
3	OBS3.2	01JAN11	3	04JAN11
4	OBS4	01JAN11	4	05JAN11
5	OBS5.4	01JAN11	5	06JAN11
6	OBS6.5	01JAN11	6	07JAN11
7	OBS7.2	01JAN11	7	08JAN11
8	OBS8.5	01JAN11	8	09JAN11
9	OBS9.8	01JAN11	9	10JAN11
10	OBS10.9	01JAN11	10	11JAN11
11	OBS11.6	01JAN11	11	12JAN11
12	OBS12.5	01JAN11	12	13JAN11
13	OBS13.13	01JAN11	13	14JAN11
14	OBS14.9	01JAN11	14	15JAN11
15	OBS15.14	01JAN11	15	16JAN11
16	OBS16.5	01JAN11	16	17JAN11
17	OBS17.14	01JAN11	17	18JAN11
18	OBS18.17	01JAN11	18	19JAN11
19	OBS19.9	01JAN11	19	20JAN11
20	OBS20.9	01JAN11	20	21JAN11
21	OBS21.2	01JAN11	21	22JAN11
22	OBS22.17	01JAN11	22	23JAN11
23	OBS23.22	01JAN11	23	24JAN11
24	OBS24.5	01JAN11	24	25JAN11
25	OBS25.4	01JAN11	25	26JAN11

See Also

Functions:

- “INTCK Function” on page 559
- “INTNX Function” on page 580

INTFMT Function

Returns a recommended SAS format when a date, time, or datetime interval is specified.

Category: Date and Time

Syntax

INTFMT(*interval*<<*multiple*.<*.shift-index*> > > , '*size*')

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts* in *SAS Language Reference: Concepts*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<*multiple.shift-index*>

The three parts of the interval name are as follows:

interval

specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

multiple

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

shift-index

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval

name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See “Incrementing Dates and Times by Using Multipliers and by Shifting Intervals” on page 31 for more information.

'size'
specifies either LONG or SHORT. When a format includes a year value, LONG or L specifies a format that uses a four-digit year. SHORT or S specifies a format that uses a two-digit year.

Details

The INTFMT function returns a recommended format depending on a date, time, or datetime interval for displaying the time ID values that are associated with a time series of a given interval. The valid values of SIZE (LONG, L, SHORT, or S) specify whether to use a two-digit or a four-digit year when the format refers to a SAS date value. For more information about working with date and time intervals, see “Date and Time Intervals” on page 31 .

The INTFMT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For a list of these intervals, see “Retail Calendar Intervals: ISO 8601 Compliant” in *SAS Language Reference: Concepts*.

Example

The following SAS statements produce these results.

SAS Statement	Result
`fmt1 = intfmt('qtr', 's');` `put fmt1;`	`YYQC4.`
`fmt2 = intfmt('qtr', 'l');` `put fmt2;`	`YYQC6.`
`fmt3 = intfmt('month', 'l');` `put fmt3;`	`MONYY7.`
`fmt4 = intfmt('week', 'short');` `put fmt4;`	`WEEKDATX15.`
`fmt5 = intfmt('week3.2', 'l');` `put fmt5;`	`WEEKDATX17.`
`fmt6 = intfmt('day', 'long');` `put fmt6;`	`DATE9.`

`var1 = 'month2';`
`fmt7 = intfmt(var1, 'long');`
`put fmt7;`
	`MONYY7.`

INTGET Function

Returns a time interval based on three date or datetime values.

Category: Date and Time

Syntax

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

Required Argument

date
specifies a SAS date or datetime value.

Details

INTGET Function Intervals

The INTGET function returns a time interval based on three date or datetime values. The function first determines all possible intervals between the first two dates, and then determines all possible intervals between the second and third dates. If the intervals are the same, INTGET returns that interval. If the intervals for the first and second dates differ, and the intervals for the second and third dates differ, INTGET compares the intervals. If one interval is a multiple of the other, then INTGET returns the smaller of the two intervals. Otherwise, INTGET returns a missing value. INTGET works best with dates generated by the INTNX function whose alignment value is BEGIN.

In the following example, INTGET returns the interval DAY2:

```
interval=intget('01mar00'd, '03mar00'd, '09mar00'd);
```

The interval between the first and second dates is DAY2, because the number of days between March 1, 2000, and March 3, 2000, is two. The interval between the second and third dates is DAY6, because the number of days between March 3, 2000, and March 9, 2000, is six. DAY6 is a multiple of DAY2. INTGET returns the smaller of the two intervals.

In the following example, INTGET returns the interval MONTH4:

```
interval=intget('01jan00'd, '01may00'd, '01may01'd);
```

The interval between the first two dates is MONTH4, because the number of months between January 1, 2000, and May 1, 2000, is four. The interval between the second and third dates is YEAR. INTGET determines that YEAR is a multiple of MONTH4 (there are three MONTH4 intervals in YEAR), and returns the smaller of the two intervals.

In the following example, INTGET returns a missing value:

```
interval=intget('01Jan2006'd, '01Apr2006'd, '01Dec2006'd);
```

The interval between the first two dates is MONTH3, and the interval between the second and third dates is MONTH8. INTGET determines that MONTH8 is not a multiple of MONTH3, and returns a missing value.

The intervals that are returned are valid SAS intervals, including multiples of the intervals and shift intervals. Valid SAS intervals are listed in Table 7.3, “Intervals Used with Date and Time Functions,” in *SAS Language Reference: Concepts* .

Note: If INTGET cannot determine a matching interval, then the function returns a missing value. No message is written to the SAS log.

Retail Calendar Intervals

The INTGET function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in Chapter 7 of *SAS Language Reference: Concepts* .

Example

The following SAS statements produce these results.

SAS Statement	Result
`interval=intget('01jan00'd,'01jan01'd,'01may01'd); put interval;`	MONTH4
`interval=intget('29feb80'd,'28feb82'd,'29feb84'd); put interval;`	YEAR2.2
`interval=intget('01feb80'd,'16feb80'd,'01mar80'd); put interval;`	SEMIMONTH
`interval=intget('2jan09'd,'2feb10'd,'2mar11'd); put interval;`	MONTH13.4
`interval=intget('10feb80'd,'19feb80'd,'28feb80'd); put interval;`	DAY9.2
`interval=intget('01apr2006:00:01:02'dt, '01apr2006:00:02:02'dt, '01apr2006:00:03:02'dt); put interval;`	MINUTE

See Also

Functions:

- "INTFIT Function" on page 567
- "INTNX Function" on page 580

INTINDEX Function

Returns the seasonal index when a date, time, or datetime interval and value are specified.

Category: Date and Time

Syntax

INTINDEX(*interval*<<*multiple*.<*shift-index*> > >, *date-value*, <*seasonality*>)

Required Arguments

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts* .

TIP If *interval* is a character constant, then enclose the value in quotation marks.

TIP Valid values for *interval* depend on whether *date-value* is a date is a date, time, or datetime value. For more information, see "Commonly Used Time Intervals" on page 32 .

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is listed below:

interval<multiple.shift-index>

The three parts of the interval name are as follows:

interval

specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

multiple

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

shift-index

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

date-value

specifies a date, time, or datetime value that represents a time period of the given interval.

Optional Argument

seasonality
specifies a number or a cycle.

This argument enables you to have more flexibility in working with dates and time cycles. You can specify whether you want a 52-week or a 53-week seasonality in a year.

Example In the following example, the function

```
INTINDEX('MONTH', date, 3);
```

produces the same result as

```
INTINDEX('MONTH', date, 'QTR');
```

Seasonality in the first example is a number (the number of months), and in the second example *seasonality* is a cycle (QTR).

Details

INTINDEX Function Intervals

The INTINDEX function returns the seasonal index when you supply an interval and an appropriate date, time, or datetime value. The seasonal index is a number that represents the position of the date, time, or datetime value in the seasonal cycle of the specified interval. For example, **`intindex('month', '01DEC2000'd);`** returns a value of 12 because there are 12 months in a yearly cycle and December is the 12th month of the year. In the following examples, INTINDEX returns the same value because both statements have values that occur in the first quarter of the year 2000: **`intindex('qtr', '01JAN2000'd);`** and **`intindex('qtr', '31MAR2000'd);`**. The statement **`intindex('day', '01DEC2000'd);`** returns a value of 6 because daily data is weekly periodic and December 1, 2000, is a Friday, the sixth day of the week.

How Interval and Date-Time-Value Are Related

To correctly identify the seasonal index, the interval should agree with the date, time, or datetime value. For example, **`intindex('month', '01DEC2000'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. In the following example, **`intindex('day', '01DEC2000'd);`** returns a value of 6 because there are seven days in a weekly interval and December 1, 2000, is a Friday, the sixth day of the week. The DAY interval requires a SAS date value.

The example **`intindex('qtr', '01JAN2000:00:00:00'dt);`** results in an error because the QTR interval expects the date to be a SAS date value rather than a datetime value. The example **`intindex('dtmonth', '01DEC2000:00:00:00'dt);`** returns a value of 12. The DTMONTH interval requires a datetime value.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 31.

Retail Calendar Intervals

The INTINDEX function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" on page 34.

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTINDEX function uses the concept of seasonality and returns the seasonal index when a date, time, or datetime interval and value are specified. For more information about seasonality and using the forecasting methods in PROC FORECAST, see the *SAS/ETS User's Guide*.

Comparisons

The INTINDEX function returns the seasonal index whereas the INTCINDEX function returns the cycle index.

In the example **`index = intindex('day', '04APR2005'd);`**, the INTINDEX function returns the day of the week. In the example **`cycle_index = intcindex('day', '04APR2005'd);`**, the INTCINDEX function returns the week of the year.

In the example **`index = intindex('minute','01Sep78:00:00:00'dt);`**, the INTINDEX function returns the minute of the hour. In the example **`cycle_index = intcindex('minute','01Sep78:00:00:00'dt);`**, the INTCINDEX function returns the hour of the day.

In the example **`intseas('interval');`**, INTSEAS returns the maximum number that could be returned by **`intindex('interval', date);`**.

Examples

Example 1: Examples of Using INTINDEX with Two Arguments

The following SAS statements produce these results.

SAS Statement	Result
`interval1 = intindex('qtr', '14AUG2005'd);` `put interval1;`	`3`
`interval2 = intindex('dtqtr','23DEC2005:15:09:19'dt);` `put interval2;`	`4`
`interval3 = intindex('hour', '09:05:15't);` `put interval3;`	`10`
`interval4 = intindex('month', '26FEB2005'd);` `put interval4;`	`2`
`interval5 = intindex('dtmonth', '28MAY2005:05:15:00'dt);` `put interval5;`	`5`
`interval6 = intindex('week', '09SEP2005'd);` `put interval6;`	`36`

SAS Statement	Result
`interval7 = intindex('tenday', '16APR2005'd);` `put interval7;`	11

Example 2: Example of Seasonality

SAS uses a default seasonal cycle. For example, the assumption is that monthly data is yearly seasonal. However, monthly data could also have a seasonal cycle of semiyearly. This example shows that to use a third argument, *seasonality*, enables you to specify the seasonality rather than using the default. It also shows how to handle leap years:

```
data weekly;
     *do year = 2000 to 2010;
     year = 2004;
        NewYear = HOLIDAY('NEWYEAR',year);
        do i = -5 to 5;
           date = INTNX('week',NewYear,i);
           output;
        end;
     *end;
     format date date.;
            format NewYear date.;
run;

/* The standard leap week is the first week of year. */
/* An alternative method uses a third argument:leap week is week 53. */
title "Using a Third Argument to Control Weekly Seasonality";
data LeapWeekExample;
     set weekly;
     StandardIndex = INTINDEX('week',date);
     IndexWithLeap = INTINDEX('week',date,53);
run;
proc print;
run;

/* Using a number and an interval can be equivalent for the third argument. */
title "Using the Third Argument as a Number or Cycle";
data Equiv3rdArg;
     set sashelp.air(obs=12);
     defaultSeasonal = INTINDEX('MONTH',date);
     SeasonalArg12 = INTINDEX('MONTH',date,12);
     SeasonalArgYear = INTINDEX('MONTH',date,'YEAR');
     format date date.;
run;
proc print;
run;


/* Use the third argument for non-standard seasonality. */
title "Using the Third Argument for Non-Standard Seasonality";
data NonStandardSeasonal;
     set sashelp.air(obs=24);
     /* Standard Index - MONTH is Yearly Seasonal */
     StandardIndex = INTINDEX('MONTH',date);
```

```
    SemiYrIndex = INTINDEX('MONTH',date,'SEMIYR');
    Index6 = INTINDEX('MONTH',date,6);
    format date date.;
run;

proc print;
run;
```

Display 2.38 *Output from the Seasonality Example*

Using a Third Argument to Control Weekly Seasonality

Obs	year	NewYear	i	date	StandardIndex	IndexWithLeap
1	2004	01JAN04	-5	23NOV03	48	48
2	2004	01JAN04	-4	30NOV03	49	49
3	2004	01JAN04	-3	07DEC03	50	50
4	2004	01JAN04	-2	14DEC03	51	51
5	2004	01JAN04	-1	21DEC03	52	52
6	2004	01JAN04	0	28DEC03	1	53
7	2004	01JAN04	1	04JAN04	1	1
8	2004	01JAN04	2	11JAN04	2	2
9	2004	01JAN04	3	18JAN04	3	3
10	2004	01JAN04	4	25JAN04	4	4
11	2004	01JAN04	5	01FEB04	5	5

Using the Third Argument as a Number or Cycle

Obs	DATE	AIR	defaultSeasonal	SeasonalArg12	SeasonalArgYear
1	01JAN49	112	1	1	1
2	01FEB49	118	2	2	2
3	01MAR49	132	3	3	3
4	01APR49	129	4	4	4
5	01MAY49	121	5	5	5
6	01JUN49	135	6	6	6
7	01JUL49	148	7	7	7
8	01AUG49	148	8	8	8
9	01SEP49	136	9	9	9
10	01OCT49	119	10	10	10
11	01NOV49	104	11	11	11
12	01DEC49	118	12	12	12

Using the Third Argument for Non-Standard Seasonality

Obs	DATE	AIR	StandardIndex	SemiYrIndex	Index6
1	01JAN49	112	1	1	1
2	01FEB49	118	2	2	2
3	01MAR49	132	3	3	3
4	01APR49	129	4	4	4
5	01MAY49	121	5	5	5
6	01JUN49	135	6	6	6
7	01JUL49	148	7	1	1
8	01AUG49	148	8	2	2
9	01SEP49	136	9	3	3
10	01OCT49	119	10	4	4
11	01NOV49	104	11	5	5
12	01DEC49	118	12	6	6
13	01JAN50	115	1	1	1
14	01FEB50	126	2	2	2
15	01MAR50	141	3	3	3
16	01APR50	135	4	4	4
17	01MAY50	125	5	5	5
18	01JUN50	149	6	6	6
19	01JUL50	170	7	1	1
20	01AUG50	170	8	2	2
21	01SEP50	158	9	3	3
22	01OCT50	133	10	4	4
23	01NOV50	114	11	5	5
24	01DEC50	140	12	6	6

See Also

Functions:

- “INTCINDEX Function” on page 556
- “INTSEAS Function” on page 589
- “INTCYCLE Function” on page 565

Other References:

- *SAS/ETS User's Guide*

INTNX Function

Increments a date, time, or datetime value by a given time interval, and returns a date, time, or datetime value.

Category: Date and Time

Syntax

INTNX(*interval*<*multiple*> <.*shift-index*> , *start-from*, *increment*<, '*alignment*'>)

INTNX(*custom-interval*, *start-from*, *increment* <, '*alignment*'>)

Required Arguments

interval

specifies a character constant, variable, or expression that contains a time interval such as WEEK, SEMIYEAR, QTR, or HOUR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in the "Intervals Used with Date and Time Functions" table in *SAS Language Reference: Concepts*.

TIP The type of interval (date, datetime, or time) must match the type of value in *start-from*.

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<*multiple.shift-index*>

The three parts of the interval name are listed below:

interval

specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

multiple

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

shift-index

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, MONTH type intervals shift by MONTH subperiods by default. Thus, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index because there are two MONTH intervals in each MONTH2 interval. The interval name MONTH2.2, for example, specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

start-from
specifies a SAS expression that represents a SAS date, time, or datetime value that identifies a starting point.

increment
specifies a negative, positive, or zero integer that represents the number of date, time, or datetime intervals. *Increment* is the number of intervals to shift the value of *start-from*.

Optional Arguments

'*alignment*'
controls the position of SAS dates within the interval. You must enclose *alignment* in quotation marks. *Alignment* can be one of these values:

BEGINNING
specifies that the returned date or datetime value is aligned to the beginning of the interval.

Alias B

MIDDLE
specifies that the returned date or datetime value is aligned to the midpoint of the interval, which is the average of the beginning and ending alignment values.

Alias M

END
specifies that the returned date or datetime value is aligned to the end of the interval.

Alias E

SAME
specifies that the date that is returned has the same alignment as the input date.

Aliases S

SAMEDAY

See "SAME Alignment" on page 584 for more information.

Default BEGINNING

See "Aligning SAS Date Output within Its Intervals" on page 583 for more information.

custom-interval
specifies an interval that you define.

Details

The Basics

The INTNX function increments a date, time, or datetime value by intervals such as DAY, WEEK, QTR, and MINUTE, or a custom interval that you define. The increment is based on a starting date, time, or datetime value, and on the number of time intervals that you specify.

The INTNX function returns the SAS date value for the beginning date, time, or datetime value of the interval that you specify in the *start–from* argument. (To convert the SAS date value to a calendar date, use any valid SAS date format, such as the DATE9. format.) The following example shows how to determine the date of the start of the week that is six weeks from the week of October 17, 2003.

```
x=intnx('week', '17oct03'd, 6);
put x date9.;
```

INTNX returns the value 23NOV2003.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 31.

Date and Datetime Intervals

The intervals that you need to use with SAS datetime values are SAS datetime intervals. Datetime intervals are formed by adding the prefix "DT" to any date interval. For example, MONTH is a SAS date interval, and DTMONTH is a SAS datetime interval. Similarly, YEAR is a SAS date interval, and DTYEAR is a SAS datetime interval.

To ensure correct results with interval functions, use date intervals with date values and datetime intervals with datetime values. SAS does not return an error message if you use a date value with a datetime interval, but the results are incorrect:

```
data _null_;
      /* The following statement creates expected results. */
   date1=intnx('dtday','01aug11:00:10:48'dt,1);
      /* The following two statements create unexpected results. */
   date2=intnx('dtday','01aug11'd,1);
   date3=intnx('dtday','01aug11:00:10:48'd,1);
   put 'Correct Datetime Value   ' date1= datetime19. /
       'Incorrect Datetime Value ' date2= datetime19. /
       'Incorrect Datetime Value ' date3= datetime19.;
run;
```

SAS writes the following output to the log:

```
Correct Datetime Value   date1=02AUG2011:00:00:00
Incorrect Datetime Value date2=02JAN1960:00:00:00
Incorrect Datetime Value date3=02JAN1960:00:00:00
```

Aligning SAS Date Output within Its Intervals

SAS date values are typically aligned with the beginning of the time interval that is specified with the *interval* argument.

You can use the optional *alignment* argument to specify the alignment of the date that is returned. The values BEGINNING, MIDDLE, or END align the date to the beginning, middle, or end of the interval, respectively.

SAME Alignment

If you use the SAME value of the *alignment* argument, then INTNX returns the same calendar date after computing the interval increment that you specified. The same calendar date is aligned based on the interval's shift period, not the interval. To view the valid shift periods, see Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts*.

Most of the values of the shift period are equal to their corresponding intervals. The exceptions are the intervals WEEK, WEEKDAY, QTR, SEMIYEAR, YEAR, and their DT counterparts. WEEK and WEEKDAY intervals have a shift period of DAYS; and QTR, SEMIYEAR, and YEAR intervals have a shift period of MONTH. When you use SAME alignment with YEAR, for example, the result is same-day alignment based on MONTH, the interval's shift period. The result is not aligned to the same day of the YEAR interval. If you specify a multiple interval, then the default shift interval is based on the interval, and not on the multiple interval.

When you use SAME alignment for QTR, SEMIYEAR, and YEAR intervals, the computed date is the same number of months from the beginning of the interval as the input date. The day of the month matches as closely as possible. Because not all months have the same number of days, it is not always possible to match the day of the month.

For more information about shift periods, see Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts* .

Alignment Intervals

Use the SAME value of the *alignment* argument if you want to base the alignment of the computed date on the alignment of the input date:

```
intnx('week', '15mar2000'd, 1, 'same');         returns 22MAR2000
intnx('dtweek', '15mar2000:8:45'dt, 1, 'same'); returns 22MAR00:08:45:00
intnx('year', '15mar2000'd, 5, 'same');        returns 15MAR2005
```

Adjusting Dates

The INTNX function automatically adjusts for the date if the date in the interval that is incremented does not exist. For example:

```
intnx('month', '15mar2000'd, 5, 'same');  returns 15AUG2000
intnx('year', '29feb2000'd, 2, 'same');   returns 28FEB2002
intnx('month', '31aug2001'd, 1, 'same');  returns 30SEP2001
intnx('year', '01mar1999'd, 1, 'same');   returns 01MAR2000 (the first day of the
                                                             third month of the year)
```

In the example **`intnx('year', '29feb2000'd, 2);`**, the INTNX function returns the value 01JAN2002, which is the beginning of the year two years from the starting date (2000).

In the example **`intnx('year', '29feb2000'd, 2, 'same');`**, the INTNX function returns the value 28FEB2002. In this case, the starting date begins in the year 2000, the year is two years later (2002), the month is the same (February), and the date is the 28th, because that is the closest date to the 29th in February 2002.

Custom Intervals

A custom interval is defined by a SAS data set. The data set must contain the *begin* variable, and it can also contain the *end* and *season* variables. Each observation represents one interval with the *begin* variable containing the start of the interval, and the *end* variable, if present, containing the end of the interval. The intervals must be listed in ascending order. You cannot have gaps between intervals, and intervals cannot overlap.

The SAS system option INTERVALDS= is used to define custom intervals and associate interval data sets with new interval names. The following example shows how to specify the INTERVALDS= system option:

```
options intervalds=(interval=libref.dataset-name);
```

Argument

interval
specifies the name of an interval. The value of *interval* is the data set that is named in *libref.dataset-name*.

libref.dataset-name
specifies the libref and data set name of the file that contains user-supplied holidays.

For more information, see "Custom Time Intervals" on page 34.

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in Chapter 7 of *SAS Language Reference: Concepts* .

Examples

Example 1

The following SAS statements produce these results.

SAS Statement	Result
`yr=intnx('year','05feb94'd,3);` `put yr / yr date7.;`	13515 01JAN97
`x=intnx('month','05jan95'd,0);` `put x / x date7.;`	12784 01JAN95
`next=intnx('semiyear','01jan97'd,1);` `put next / next date7.;`	13696 01JUL97
`past=intnx('month2','01aug96'd,-1);` `put past / past date7.;`	13270 01MAY96
`sm=intnx('semimonth2.2','01apr97'd,4);` `put sm / sm date7.;`	13711 16JUL97

`x='month';`
`date='1jun1990'd;`
`nextmon=intnx(x,date,1);`
`put nextmon / nextmon date7.;`
	11139
	01JUL90

SAS Statement	Result

x1='month ';
x2=trim(x1);
date='1jun1990'd - 100;
nextmonth=intnx(x2,date,1);
put nextmonth / nextmonth date7.;
	11017
	01MAR90

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

SAS Statement	Result
date1=intnx('month','01jan95'd,5,'beginning'); put date1 / date1 date7.;	12935 01JUN95
date2=intnx('month','01jan95'd,5,'middle'); put date2 / date2 date7.;	12949 15JUN95
date3=intnx('month','01jan95'd,5,'end'); put date3 / date3 date7.;	12964 30JUN95
date4=intnx('month','01jan95'd,5,'sameday'); put date4 / date4 date7.;	12935 01JUN95
date5=intnx('month','15mar2000'd,5,'same'); put date5 / date5 date9.;	14837 15AUG2000

interval='month';
date='1sep2001'd;
align='m';
date4=intnx(interval,date,2,align);
put date4 / date4 date7.;
	15294
	15NOV01

x1='month ';
x2=trim(x1);
date='1sep2001'd + 90;
date5=intnx(x2,date,2,'m');
put date5 / date5 date7.;
	15356
	16JAN02

Example 2: Example of Using Custom Intervals

The following example uses the *custom-interval* form of the INTNX function to increment a date, time, or datetime value by a given time interval.

```
options intervalds=(weekdaycust=dstest);
data dstest;
   format begin end date9.;
   begin='01jan2008'd; end='01jan2008'd; output;
   begin='02jan2008'd; end='02jan2008'd; output;
```

```
   begin='03jan2008'd; end='03jan2008'd; output;
   begin='04jan2008'd; end='06jan2008'd; output;
   begin='07jan2008'd; end='07jan2008'd; output;
   begin='08jan2008'd; end='08jan2008'd; output;
   begin='09jan2008'd; end='09jan2008'd; output;
   begin='10jan2008'd; end='10jan2008'd; output;
   begin='11jan2008'd; end='13jan2008'd; output;
   begin='14jan2008'd; end='14jan2008'd; output;
   begin='15jan2008'd; end='15jan2008'd; output;
run;

data _null_;
   format start date9. endcustom date9.;
   start='01jan2008'd;
   do i=0 to 9;
      endcustom=intnx('weekdaycust', start, i);
      put endcustom;
   end;
run;
```

SAS writes the following output to the log:

```
01JAN2008
02JAN2008
03JAN2008
04JAN2008
07JAN2008
08JAN2008
09JAN2008
10JAN2008
11JAN2008
14JAN2008
```

See Also

Functions:

- “INTCK Function” on page 559
- “INTSHIFT Function” on page 592

System Options:

- “INTERVALDS= System Option” in *SAS System Options: Reference*

INTRR Function

Returns the internal rate of return as a fraction.

Category: Financial

Syntax

INTRR(*freq*,*c0*, *c1*,..., *cn*)

Required 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 $freq > 0$

Tip The case $freq = 0$ is a flag to allow continuous compounding.

c0,c1, ... ,cn
are numeric, the optional cash payments.

Details

The INTRR function returns the internal rate of return over a specified base period of time for the set of cash payments *c0*, *c1*,..., *cn*. The time intervals between any two consecutive payments are assumed to be equal. The argument $freq > 0$ describes the number of payments that occur over the specified base period of time. The number of notes issued from each instance is limited.

The internal rate of return is the interest rate such that the sequence of payments has a 0 net present value. (See the "NETPV Function" on page 680 .) It is given by

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

where x is the real root of the polynomial.

$$\sum_{i=0}^{n} c_i x^i = 0$$

In the case of multiple roots, one real root is returned and a warning is issued concerning the non-uniqueness of the returned internal rate of return. Depending on the value of payments, a root for the equation does not always exist. In that case, a missing value is returned.

Missing values in the payments are treated as 0 values. When $freq > 0$, 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 returned is 0.19438.

See Also

Functions:

- "IRR Function" on page 600

INTSEAS Function

Returns the length of the seasonal cycle when a date, time, or datetime interval is specified.

Category: Date and Time

Syntax

INTSEAS(*interval*<<*multiple*.<*shift-index*> > > <*seasonality*>)

Required Argument

interval

specifies a character constant, a variable, or an expression that contains an interval name such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in Table 7.3, "Intervals Used with Date and Time Functions," in *SAS Language Reference: Concepts* ..

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<*multiple.shift-index*>

The three parts of the interval name are as follows:

interval

specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

multiple

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

shift-index

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use

YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

Optional Argument

seasonality
specifies a number or a cycle.

This argument enables you to have more flexibility in working with dates and time cycles. If there is a 53-week year, you can easily determine the seasonality by using 53 as the value for *seasonality*, as the following example shows: **`INTSEAS('WEEK', 53);`**. By default, **`INTSEAS('WEEK');`** equals 52.

Example The function

```
INTSEAS('interval', seasonality);
```

returns a number when you specify a numeric value for *seasonality*. The function

```
INTSEAS('MONTH', 'QTR');
```

returns a value of 3 when you specify the QTR cycle.

Details

The Basics

The INTSEAS function returns the number of intervals in a seasonal cycle. For example, when the interval for a time series is described as monthly, then many procedures use the option INTERVAL=MONTH. Each observation in the data then corresponds to a particular month. Monthly data is considered to be periodic for a one-year period. A year contains 12 months, so the number of intervals (months) in a seasonal cycle (year) is 12.

Quarterly data is also considered to be periodic for a one-year period. A year contains four quarters, so the number of intervals in a seasonal cycle is four.

The periodicity is not always one year. For example, INTERVAL=DAY is considered to have a period of one week. Because there are seven days in a week, the number of intervals in the seasonal cycle is seven.

For more information about working with date and time intervals, see "Date and Time Intervals" on page 31 .

Retail Calendar Intervals

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, or 5-4-4. The first, second, and third numbers specify the number of weeks in the first, second, and third

month of each period, respectively. For more information, see “Retail Calendar Intervals: ISO 8601 Compliant” in Chapter 7 of *SAS Language Reference: Concepts* .

Seasonality

Seasonality is a time series concept that measures cyclical variations at different intervals during the year. In specifying seasonality, the time of year is the most common source of the variations. For example, sales of home heating oil are regularly greater in winter than during other times of the year. Often, certain days of the week cause regular fluctuations in daily time series, such as increased spending on leisure activities during weekends. The INTSEAS function uses the concept of seasonality and returns the length of the seasonal cycle when a date, time, or datetime interval is specified. For more information about seasonality and forecasting, see the *SAS/ETS User's Guide*.

Example

The following SAS statements produce these results.

SAS Statement	Result
`cycle_years = intseas('year');` `put cycle_years;`	`1`
`cycle_smiyears = intseas('semiyear');` `put cycle_smiyears;`	`2`
`cycle_quarters = intseas('quarter');` `put cycle_quarters;`	`4`
`cycle_number = intseas('month', 'qtr');` `put cycle_number;`	`3`
`cycle_months = intseas('month');` `put cycle_months;`	`12`
`cycle_smimonths = intseas('semimonth');` `put cycle_smimonths;`	`24`
`cycle_tendays = intseas('tenday');` `put cycle_tendays;`	`36`
`cycle_weeks = intseas('week');` `put cycle_weeks;`	`52`
`cycle_wkdays = intseas('weekday');` `put cycle_wkdays;`	`5`
`cycle_hours = intseas('hour');` `put cycle_hours;`	`24`
`cycle_minutes = intseas('minute');` `put cycle_minutes;`	`60`
`cycle_month2 = intseas('month2.2');` `put cycle_month2;`	`6`

SAS Statement	Result
`cycle_week2 = intseas('week2');` `put cycle_week2;`	`26`

`var1 = 'month4.3';`
`cycle_var1 = intseas(var1);`
`put cycle_var1;`
	`3`

`cycle_day1 = intseas('day1');` `put cycle_day1;`	`7`

See Also

Functions:

- “INTCYCLE Function” on page 565
- “INTINDEX Function” on page 574

Other References:

- *SAS/ETS User's Guide*

INTSHIFT Function

Returns the shift interval that corresponds to the base interval.

Category: Date and Time

Syntax

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

Required Arguments

interval
specifies a character constant, a variable, or an expression that contains a time interval such as WEEK, SEMIYEAR, QTR, or HOUR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in Table 7.3, “Intervals Used with Date and Time Functions,” in *SAS Language Reference: Concepts* .

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<multiple.shift-index>
The three parts of the interval name are as follows:

interval
specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 would be an error because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

Details

The INTSHIFT function returns the shift interval that corresponds to the base interval. For custom intervals, the value that is returned is the base custom interval name. INTSHIFT ignores multiples of the interval and interval shifts.

The INTSHIFT function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in Chapter 7 of *SAS Language Reference: Concepts* .

Example

The following SAS statements produce these results.

SAS Statement	Result
`shift1 = intshift('year');` `put shift1;`	`MONTH`
`shift2 = intshift('dtyear');` `put shift2;`	`DTMONTH`

SAS Statement	Result
`shift3 = intshift('minute');` `put shift3;`	DTMINUTE

`interval = 'weekdays';`
`shift4 = intshift(interval);`
`put shift4;`
	WEEKDAY

`shift5 = intshift('weekday5.4');` `put shift5;`	WEEKDAY
`shift6 = intshift('qtr');` `put shift6;`	MONTH
`shift7 = intshift('dttenday');` `put shift7;`	DTTENDAY

INTTEST Function

Returns 1 if a time interval is valid, and returns 0 if a time interval is invalid.

Category: Date and Time

Syntax

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

Required Argument

interval

specifies a character constant, variable, or expression that contains an interval name, such as WEEK, MONTH, or QTR. *Interval* can appear in uppercase or lowercase. The possible values of *interval* are listed in Table 7.3, “Intervals Used with Date and Time Functions,” in *SAS Language Reference: Concepts* .

Multipliers and shift indexes can be used with the basic interval names to construct more complex interval specifications. The general form of an interval name is as follows:

interval<*multiple.shift-index*>

Here are the three parts of the interval name:

interval

specifies the name of the basic interval type. For example, YEAR specifies yearly intervals.

multiple

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

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

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

Restrictions The shift index cannot be greater than the number of subperiods in the whole interval. For example, you could use YEAR2.24, but YEAR2.25 is invalid because there is no 25th month in a two-year interval.

If the default shift period is the same as the interval type, then only multiperiod intervals can be shifted with the optional shift index. For example, because MONTH type intervals shift by MONTH subperiods by default, monthly intervals cannot be shifted with the shift index. However, bimonthly intervals can be shifted with the shift index, because there are two MONTH intervals in each MONTH2 interval. For example, the interval name MONTH2.2 specifies bimonthly periods starting on the first day of even-numbered months.

See "Incrementing Dates and Times by Using Multipliers and by Shifting Intervals" on page 31 for more information.

Details

The INTTEST function checks for a valid interval name. This function is useful when checking for valid values of *multiple* and *shift-index*. For more information about multipliers and shift indexes, see "Multiunit Intervals" in *SAS Language Reference: Concepts*.

The INTTEST function can also be used with calendar intervals from the retail industry. These intervals are ISO 8601 compliant. For more information, see "Retail Calendar Intervals: ISO 8601 Compliant" in Chapter 7 of *SAS Language Reference: Concepts* .

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.

SAS Statement	Result
`test1 = inttest('month');` `put test1;`	1
`test2 = inttest('week6.13');` `put test2;`	1
`test3 = inttest('tenday');` `put test3;`	1

SAS Statement	Result
`test4 = inttest('twoweeks');` `put test4;`	`0`

`var1 = 'hour2.2';`
`test5 = inttest(var1);`
`put test5;`
	`1`

INTZ Function

Returns the integer portion of the argument, using zero fuzzing.

Category: Truncation

Syntax

INTZ (*argument*)

Required Argument

argument
is a numeric constant, variable, or expression.

Details

The following rules apply:

- If the value of the argument is an exact integer, INTZ returns that integer.
- If the argument is positive and not an integer, INTZ returns the largest integer that is less than the argument.
- If the argument is negative and not an integer, INTZ returns the smallest integer that is greater than the argument.

Comparisons

Unlike the INT function, the INTZ function uses zero fuzzing. If the argument is within 1E-12 of an integer, the INT function fuzzes the result to be equal to that integer. The INTZ function does not fuzz the result. Therefore, with the INTZ function you might get unexpected results.

Example

The following SAS statements produce these results.

SAS Statement	Result

`var1=2.1;`
`a=intz(var1);`
`put a;`
	`2`

SAS Statement	Result

`var2=-2.4;`
`b=intz(var2);`
`put b;`
	`-2`

`var3=1+1.e-11;`
`c=intz(var3);`
`put c;`
	`1`

`f=intz(-1.6);` `put f;`	`-1`

See Also

Functions:

- “CEIL Function” on page 294
- “CEILZ Function” on page 296
- “FLOOR Function” on page 480
- “FLOORZ Function” on page 481
- “INT Function” on page 555
- “ROUND Function” on page 833
- “ROUNDZ Function” on page 843

IORCMSG Function

Returns a formatted error message for _IORC_.

Category: SAS File I/O

Syntax

IORCMSG()

Details

If the IORCMSG function returns a value to a variable that has not yet been assigned a length, then by default the variable is assigned a length of 200.

The IORCMSG function returns the formatted error message that is associated with the current value of the automatic variable _IORC_. The _IORC_ variable is created when you use the MODIFY statement, or when you use the SET statement with the KEY= option. The value of the _IORC_ variable is internal and is meant to be read in conjunction with the SYSRC autocall macro. If you try to set _IORC_ to a specific value, you might get unexpected results.

Example

In the following program, observations are either rewritten or added to the updated master file that contains bank accounts and current bank balance. The program queries the _IORC_ variable and returns a formatted error message if the _IORC_ value is unexpected.

```
libname bank 'SAS-library';
data bank.master(index=(AccountNum));
   infile 'external-file-1';
   format balance dollar8.;
   input @ 1 AccountNum $ 1-3 @ 5 balance 5-9;
run;
data bank.trans(index=(AccountNum));
   infile 'external-file-2';
   format deposit dollar8.;
   input @ 1 AccountNum $ 1-3 @ 5 deposit 5-9;
run;
data bank.master;
   set bank.trans;
   modify bank.master key=AccountNum;
   if (_IORC_ EQ %sysrc(_SOK)) then
      do;
         balance=balance+deposit;
         replace;
      end;
else
   if (_IORC_ = %sysrc(_DSENOM)) then
      do;
         balance=deposit;
         output;
         _error_=0;
      end;
else
   do;
      errmsg=IORCMSG();
      put 'Unknown error condition:'
      errmsg;
   end;
run;
```

IPMT Function

Returns the interest payment for a given period for a constant payment loan or the periodic savings for a future balance.

Category: Financial

Syntax

IPMT (*rate*, *period*, *number-of-periods*, *principal-amount*, <*future-amount*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

period
specifies the payment period for which the interest payment is computed. *period* must be a positive integer value that is less than or equal to the value of *number-of-periods*.

number-of-periods
specifies the number of payment periods. *number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

future-amount
specifies the future amount. *future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a missing value is specified.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Example

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 66.67.

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 0.0.

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

  This computation returns a value of 292.447.

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

  This computation returns a value of 7.4314473.

IQR Function

Returns the interquartile range.

Category: Descriptive Statistics

Syntax

IQR(*value-1* <,*value-2*...>)

Required Argument

value
specifies a numeric constant, variable, or expression for which the interquartile range is to be computed.

Details

If all arguments have missing values, the result is a missing value. Otherwise, the result is the interquartile range of the nonmissing values. The formula for the interquartile range is the same as the one that is used in the UNIVARIATE procedure. For more information, see *Base SAS Procedures Guide*.

Example

The following SAS statements produce these results.

SAS Statement	Result
`iqr=iqr(2,4,1,3,999999);` `put iqr;`	2

See Also

Functions:

- “MAD Function” on page 650
- “PCTL Function” on page 720

IRR Function

Returns the internal rate of return as a percentage.

Category: Financial

Syntax

IRR(*freqc0,c1,...,cn*)

Required 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.

Tip The case *freq* = 0 is a flag to allow continuous compounding.

c0,c1,…,cn
are numeric, the optional cash payments.

Details

The IRR 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.

Comparisons

The IRR function is identical to INTRR, except for in the IRR function, the internal rate of return is a percentage.

See Also

Functions:

- "INTRR Function" on page 587

JBESSEL Function

Returns the value of the Bessel function.

Category: Mathematical

Syntax

JBESSEL(*nu*,*x*)

Required Arguments

nu
specifies a numeric constant, variable, or expression.

Range $nu \geq 0$

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 SAS statement produces this result.

SAS Statement	Result
`x=jbessel(2,2);`	`0.3528340286`

JULDATE Function

Returns the Julian date from a SAS date value.

Category: Date and Time

Syntax

JULDATE(*date*)

Required Argument

date
specifies a SAS date value.

Details

A SAS date value is a number that represents the number of days from January 1, 1960 to a specific date. The JULDATE function converts a SAS date value to a Julian date. If *date* falls within the 100-year span defined by the system option YEARCUTOFF=, the result has three, four, or five digits. In a five digit result, the first two digits represent the year, and the next three digits represent the day of the year (1 to 365, or 1 to 366 for leap years). Because leading zeros are dropped from the result, the year portion of a Julian date might be omitted (for years ending in 00), or it might have only one digit (for years ending 01–09). Otherwise, the result has seven digits: the first four digits represent the year, and the next three digits represent the day of the year. For example, if YEARCUTOFF=1920, JULDATE would return 97001 for January 1, 1997, and return 1878365 for December 31, 1878.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four digit year. Thus JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two digit year.

Example

The following SAS statements produce these results.

SAS Statement	Result
`julian=juldate('31dec99'd);`	`99365`

SAS Statement	Result
`julian=juldate('01jan2099'd);`	`2099001`

See Also

Functions:

- "DATEJUL Function" on page 358
- "JULDATE7 Function" on page 603

System Options:

- "Using the YEARCUTOFF= System Option" in Chapter 7 of *SAS Language Reference: Concepts*

JULDATE7 Function

Returns a seven-digit Julian date from a SAS date value.

Category: Date and Time

Syntax

JULDATE7(*date*)

Required Argument

date
specifies a SAS date value.

Details

The JULDATE7 function returns a seven digit Julian date from a SAS date value. The first four digits represent the year, and the next three digits represent the day of the year.

Comparisons

The function JULDATE7 is similar to JULDATE except that JULDATE7 always returns a four digit year. Thus JULDATE7 is year 2000 compliant because it eliminates the need to consider the implications of a two digit year.

Example

The following SAS statements produce these results.

SAS Statement	Result
`julian=juldate7('31dec96'd);`	`1996366`

SAS Statement	Result
`julian=juldate7('01jan2099'd);`	`2099001`

See Also

Functions:

- "JULDATE Function" on page 602

KURTOSIS Function

Returns the kurtosis.

Category: Descriptive Statistics

Syntax

KURTOSIS(*argument-1*,*argument-2*,*argument-3*,*argument-4*<,…,*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

At least four non-missing arguments are required. Otherwise, the function returns a missing value. If all non-missing arguments have equal values, the kurtosis is mathematically undefined. The KURTOSIS function returns a missing value and sets _ERROR_ equal to 1.

The argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=kurtosis(5,9,3,6);`	`0.928`
`x2=kurtosis(5,8,9,6,.);`	`-3.3`
`x3=kurtosis(8,9,6,1);`	`1.5`
`x4=kurtosis(8,1,6,1);`	`-4.483379501`
`x5=kurtosis(of x1-x4);`	`-5.065692754`

LAG Function

Returns values from a queue.

Category: Special

Syntax

LAG<*n*> (*argument*)

Required Argument

argument
specifies a numeric or character constant, variable, or expression.

Optional Argument

n
specifies the number of lagged values.

Details

The Basics

If the LAG function returns a value to a character variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

The LAG functions, LAG1, LAG2, ..., LAG*n* return values from a queue. LAG1 can also be written as LAG. A LAG*n* function stores a value in a queue and returns a value stored previously in that queue. Each occurrence of a LAG*n* function in a program generates its own queue of values.

The queue for each occurrence of LAG*n* is initialized with *n* missing values, where *n* is the length of the queue (for example, a LAG2 queue is initialized with two missing values). When an occurrence of LAG*n* is executed, the value at the top of its queue is removed and returned, the remaining values are shifted upwards, and the new value of the argument is placed at the bottom of the queue. Hence, missing values are returned for the first *n* executions of each occurrence of LAG*n*, after which the lagged values of the argument begin to appear.

Note: Storing values at the bottom of the queue and returning values from the top of the queue occurs only when the function is executed. An occurrence of the LAG*n* function that is executed conditionally will store and return values only from the observations for which the condition is satisfied.

If the argument of LAG*n* is an array name, a separate queue is maintained for each variable in the array.

Memory Limit for the LAG Function

When the LAG function is compiled, SAS allocates memory in a queue to hold the values of the variable that is listed in the LAG function. For example, if the variable in function LAG100(x) is numeric with a length of 8 bytes, then the memory that is needed is 8 times 100, or 800 bytes. Therefore, the memory limit for the LAG function is based on the memory that SAS allocates, which varies with different operating environments.

Examples

Example 1: Generating Two Lagged Values

The following program generates two lagged values for each observation.

```
data one;
   input x @@;
   y=lag1(x);
   z=lag2(x);
   datalines;
1 2 3 4 5 6
;
proc print data=one;
   title 'LAG Output';
run;
```

Display 2.39 *Output from Generating Two Lagged Values*

LAG Output

Obs	x	y	z
1	1	.	.
2	2	1	.
3	3	2	1
4	4	3	2
5	5	4	3
6	6	5	4

LAG1 returns one missing value and the values of X (lagged once). LAG2 returns two missing values and the values of X (lagged twice).

Example 2: Generating Multiple Lagged Values in BY-Groups

The following example shows how to generate up to three lagged values within each BY group.

```
/*************************************************************************/
/* This program generates up to three lagged values.  By increasing the  */
/* size of the array and the number of assignment statements that use     */
/* the LAGn functions, you can generate as many lagged values as needed.  */
/*************************************************************************/
/* Create starting data. */
data old;
  input start end;
datalines;
1 1
1 2
1 3
1 4
1 5
1 6
1 7
2 1
```

```
2 2
3 1
3 2
3 3
3 4
3 5
;
data new(drop=i count);
  set old;
  by start;
  /* Create and assign values to three new variables.  Use ENDLAG1-      */
  /* ENDLAG3 to store lagged values of END, from the most recent to the  */
  /* third preceding value.                                              */
  array x(*) endlag1-endlag3;
  endlag1=lag1(end);
  endlag2=lag2(end);
  endlag3=lag3(end);
  /* Reset COUNT at the start of each new BY-Group */
  if first.start then count=1;
  /* On each iteration, set to missing array elements   */
  /* that have not yet received a lagged value for the  */
  /* current BY-Group.  Increase count by 1.            */
  do i=count to dim(x);
    x(i)=.;
  end;
  count + 1;
run;
proc print;
run;
```

Display 2.40 *Output from Generating Three Lagged Values*

The SAS System

Obs	start	end	endlag1	endlag2	endlag3
1	1	1	.	.	.
2	1	2	1	.	.
3	1	3	2	1	.
4	1	4	3	2	1
5	1	5	4	3	2
6	1	6	5	4	3
7	1	7	6	5	4
8	2	1	.	.	.
9	2	2	1	.	.
10	3	1	.	.	.
11	3	2	1	.	.
12	3	3	2	1	.
13	3	4	3	2	1
14	3	5	4	3	2

Example 3: Computing the Moving Average of a Variable

The following is an example that computes the moving average of a variable in a data set.

```
/* Title: Compute the moving average of a variable
   Goal: Compute the moving average of a variable through the entire data set,
         of the last n observations and of the last n observations within a
         BY-group.
   Input:
*/
data x;
do x=1 to 10;
  output;
  end;
run;
/* Compute the moving average of the entire data set. */
data avg;
retain s 0;
set x;
s=s+x;
a=s/_n_;
run;
proc print;
run;
/* Compute the moving average of the last 5 observations. */
%let n = 5;
data avg (drop=s);
retain s;
set x;
s = sum (s, x, -lag&n(x)) ;
a = s / min(_n_, &n);
run;
proc print;
run;
/* Compute the moving average within a BY-group of last n observations.
  For the first n-1 observations within the BY-group, the moving average
  is set to missing. */
data ds1;
do patient='A','B','C';
 do month=1 to 7;
  num=int(ranuni(0)*10);
  output;
 end;
end;
run;
proc sort;
by patient;
%let n = 4;
data ds2;
set ds1;
by patient;
retain num_sum 0;
if first.patient then do;
  count=0;
  num_sum=0;
end;
```

```
count+1;
last&n=lag&n(num);
if count gt &n then num_sum=sum(num_sum,num,-last&n);
else num_sum=sum(num_sum,num);
if count ge &n then mov_aver=num_sum/&n;
else mov_aver=.;
run;
proc print;
run;
```

Display 2.41 *Output from Computing the Moving Average of a Variable*

The SAS System

Obs	s	x	a
1	1	1	1.0
2	3	2	1.5
3	6	3	2.0
4	10	4	2.5
5	15	5	3.0
6	21	6	3.5
7	28	7	4.0
8	36	8	4.5
9	45	9	5.0
10	55	10	5.5

The SAS System

Obs	x	a
1	1	1.0
2	2	1.5
3	3	2.0
4	4	2.5
5	5	3.0
6	6	4.0
7	7	5.0
8	8	6.0
9	9	7.0
10	10	8.0

The SAS System

Obs	patient	month	num	num_sum	count	last4	mov_aver
1	A	1	9	9	1	.	.
2	A	2	0	9	2	.	.
3	A	3	1	10	3	.	.
4	A	4	6	16	4	.	4.00
5	A	5	3	10	5	9	2.50
6	A	6	9	19	6	0	4.75
7	A	7	5	23	7	1	5.75
8	B	1	7	7	1	6	.
9	B	2	8	15	2	3	.
10	B	3	8	23	3	9	.
11	B	4	1	24	4	5	6.00
12	B	5	0	17	5	7	4.25
13	B	6	6	15	6	8	3.75
14	B	7	4	11	7	8	2.75
15	C	1	5	5	1	1	.
16	C	2	7	12	2	0	.
17	C	3	1	13	3	6	.
18	C	4	8	21	4	4	5.25
19	C	5	0	16	5	5	4.00
20	C	6	0	9	6	7	2.25
21	C	7	6	14	7	1	3.50

Example 4: Generating a Fibonacci Sequence of Numbers

The following example generates a Fibonacci sequence of numbers. You start with 0 and 1, and then add the two previous Fibonacci numbers to generate the next Fibonacci number.

```
data _null_;
   put 'Fibonacci Sequence';
   n=1;
   f=1;
   put n= f=;
   do n=2 to 10;
      f=sum(f,lag(f));
      put n= f=;
   end;
run;
```

SAS writes the following output to the log:

```
Fibonacci Sequence
n=1 f=1
n=2 f=1
n=3 f=2
n=4 f=3
n=5 f=5
n=6 f=8
n=7 f=13
n=8 f=21
```

```
n=9 f=34
n=10 f=55
```

Example 5: Using Expressions for the LAG Function Argument

The following program uses an expression for the value of *argument* and creates a data set that contains the values for X, Y, and Z. LAG dequeues the previous values of the expression and enqueues the current value.

```
data one;
   input X @@;
   Y=lag1(x+10);
   Z=lag2(x);
   datalines;
1 2 3 4 5 6
;
proc print;
   title 'Lag Output: Using an Expression';
run;
```

Display 2.42 *Output from the LAG Function: Using an Expression*

Lag Output: Using an Expression

Obs	X	Y	Z
1	1	.	.
2	2	11	.
3	3	12	1
4	4	13	2
5	5	14	3
6	6	15	4

See Also

Functions:

- "DIF Function" on page 374

LARGEST Function

Returns the *k*th largest non-missing value.

Category: Descriptive Statistics

Syntax

LARGEST (*k*, *value-1*<, *value-2* ...>)

Required Arguments

k
is a numeric constant, variable, or expression that specifies which value to return.

value
specifies the value of a numeric constant, variable, or expression to be processed.

Details

If *k* is missing, less than zero, or greater than the number of values, the result is a missing value and _ERROR_ is set to 1. Otherwise, if *k* is greater than the number of non-missing values, the result is a missing value but _ERROR_ is not set to 1.

Example

The following SAS statements produce these results.

SAS Statement	Result

`k=1;`
`largest1=largest(k, 456, 789, .Q, 123);`
`put largest1;`
	`789`

`k=2;`
`largest2=largest(k, 456, 789, .Q, 123);`
`put largest2;`
	`456`

`k=3;`
`largest3=largest(k, 456, 789, .Q, 123);`
`put largest3;`
	`123`

`k=4;`
`largest4=largest(k, 456, 789, .Q, 123);`
`put largest4;`
	`.`

See Also

Functions:

- “ORDINAL Function” on page 718
- “PCTL Function” on page 720
- “SMALLEST Function” on page 864

LBOUND Function

Returns the lower bound of an array.

Category: Array

Syntax

LBOUND<*n*> (*array-name*)

LBOUND(*array-name*,*bound-n*)

Required Arguments

array-name
is the name of an array that was defined previously in the same DATA step.

bound-n
is a numeric constant, variable, or expression that specifies the dimension for which you want to know the lower bound. Use *bound-n* only if *n* is not specified.

Optional Argument

n
is an integer constant that specifies the dimension for which you want to know the lower bound. If no *n* value is specified, the LBOUND function returns the lower bound of the first dimension of the array.

Details

The LBOUND function returns the lower bound of a one-dimensional array or the lower bound of a specified dimension of a multidimensional array. Use LBOUND in array processing to avoid changing the lower bound of an iterative DO group each time you change the bounds of the array. LBOUND and HBOUND can be used together to return the values of the lower and upper bounds of an array dimension.

Examples

Example 1: One-Dimensional Array

In this example, LBOUND returns the lower bound of the dimension, a value of 2. SAS repeats the statements in the DO loop five times.

```
array big{2:6} weight sex height state city;
do i=lbound(big) to hbound(big);
   ...more SAS statements...;
end;
```

Example 2: Multidimensional Array

This example shows two ways of specifying the LBOUND function for multidimensional arrays. Both methods return the same value for LBOUND, as shown in the table that follows the SAS code example.

```
array mult{2:6,4:13,2} mult1-mult100;
```

Syntax	Alternative Syntax	Value
LBOUND(MULT)	LBOUND(MULT,1)	2
LBOUND2(MULT)	LBOUND(MULT,2)	4
LBOUND3(MULT)	LBOUND(MULT,3)	1

See Also

Functions:

- "DIM Function" on page 377
- "HBOUND Function" on page 528

Statements:

- "ARRAY Statement" in *SAS Statements: Reference*
- "Array Reference Statement" in *SAS Statements: Reference*
- "Definitions for Array Processing" in Chapter 23 of *SAS Language Reference: Concepts*

LCM Function

Returns the least common multiple.

Category: Mathematical

Syntax

LCM(*x1*,*x2*,*x3*, …, *xn***)**

Required Argument

x
specifies a numeric constant, variable, or expression that has an integer value.

Details

The LCM (least common multiple) function returns the smallest multiple that is exactly divisible by every member of a set of numbers. For example, the least common multiple of 12 and 18 is 36.

If any of the arguments are missing, then the returned value is a missing value.

Example

The following example returns the smallest multiple that is exactly divisible by the integers 10 and 15.

```
data _null_;
   x=lcm(10,15);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=30
```

See Also

Functions:

- "GCD Function" on page 508

LCOMB Function

Computes the logarithm of the COMB function, which is the logarithm of the number of combinations of *n* objects taken *r* at a time.

Category: Combinatorial

Syntax

LCOMB(*n*,*r*)

Required Arguments

n
is a non-negative integer that represents the total number of elements from which the sample is chosen.

r
is a non-negative integer 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 SAS statements produce these results.

SAS Statement	Result
x=lcomb(5000,500); put x;	1621.4411361
y=lcomb(100,10); put y;	30.482323362

See Also

Functions:

- "COMB Function" on page 310

LEFT Function

Left-aligns a character string.

Category: Character

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

Tip: DBCS equivalent function is KLEFT . See "DBCS Compatibility" on page 616.

Syntax

LEFT(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

The Basics

In a DATA step, if the LEFT function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

LEFT returns an argument with leading blanks moved to the end of the value. The argument's length does not change.

DBCS Compatibility

The LEFT function left-aligns a character string. You can use the LEFT function in most cases. If an application can be executed in an ASCII environment, or if the application does not manipulate character strings, then using the LEFT function rather than the KLEFT function.

Example

The following SAS statements produce these results.

SAS Statement	Result
	`----+----1----+`

`a='  DUE DATE';`
`b=left(a);`
`put b;`
	`DUE DATE`

See Also

Functions:

- "COMPRESS Function" on page 327
- "RIGHT Function" on page 832
- "STRIP Function" on page 888
- "TRIM Function" on page 924

LENGTH Function

Returns the length of a non-blank character string, excluding trailing blanks, and returns 1 for a blank character string.

Category: Character

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

Tips: DBCS equivalent function is KLENGTH .

The LENGTH function returns a length in bytes, while the KLENGTH function returns a length in a character based unit.

Syntax

LENGTH(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTH function returns an integer that represents the position of the rightmost non-blank character in *string*. If the value of *string* is blank, LENGTH returns a value of 1. If *string* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTH returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

- The LENGTH and LENGTHN functions return the same value for non-blank character strings. LENGTH returns a value of 1 for blank character strings, whereas LENGTHN returns a value of 0.
- The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks.
- The LENGTH function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string.

Example

The following SAS statements produce these results.

SAS Statement	Result
`len=length('ABCDEF');` `put len;`	`6`
`len2=length(' ');` `put len2;`	`1`

See Also

Functions:

- “LENGTHC Function” on page 618
- “LENGTHM Function” on page 619
- “LENGTHN Function” on page 621

LENGTHC Function

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

Category: Character

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

Syntax

LENGTHC(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTHC function returns the number of characters, both blanks and non-blanks, in *string*. If *string* is a numeric constant, variable or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTHC returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTH and LENGTHN functions return the length of a character string, excluding trailing blanks. LENGTHC always returns a value that is greater than or equal to the value of LENGTHN.
- The LENGTHC function returns the length of a character string, including trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that

is allocated for a character string. For fixed-length character strings, LENGTHC and LENGTHM always return the same value. For varying-length character strings, LENGTHC always returns a value that is less than or equal to the value returned by LENGTHM.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=lengthc('variable with trailing blanks   ');` `put x;`	32

`length fixed $35;`
`fixed='variable with trailing blanks   ';`
`x=lengthc(fixed);`
`put x;`
	35

See Also

Functions:

- “LENGTH Function” on page 617
- “LENGTHM Function” on page 619
- “LENGTHN Function” on page 621

LENGTHM Function

Returns the amount of memory (in bytes) that is allocated for a character string.

Category: Character

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

Syntax

LENGTHM(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTHM function returns an integer that represents the amount of memory in bytes that is allocated for *string*. If *string* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTHM

returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

The LENGTHM function returns the amount of memory in bytes that is allocated for a character string, whereas the LENGTH, LENGTHC, and LENGTHN functions return the length of a character string. LENGTHM always returns a value that is greater than or equal to the values that are returned by LENGTH, LENGTHC, and LENGTHN.

Examples

Example 1: Determining the Amount of Allocated Memory for a Character Expression

This example determines the amount of memory (in bytes) that is allocated for a buffer that stores intermediate results in a character expression. Because SAS does not know how long the value of the expression CAT(x,y) will be, SAS allocates memory for values up to 32767 bytes long.

```
data _null_;
   x='x';
   y='y';
   lc=lengthc(cat(x,y));
   lm=lengthm(cat(x,y));
   put lc= lm=;
run;
```

SAS writes the following output to the log:

```
lc=2 lm=32767
```

Example 2: Determining the Amount of Allocated Memory for a Variable from an External File

This example determines the amount of memory (in bytes) that is allocated to a variable that is input into a SAS file from an external file.

```
data _null_;
   file 'test.txt';
   put 'trailing blanks   ';
run;
data test;
   infile 'test.txt';
   input;
   x=lengthm(_infile_);
   put x;
run;
```

The following line is written to the SAS log:

```
256
```

See Also

Functions:

- “LENGTH Function” on page 617

- "LENGTHC Function" on page 618
- "LENGTHN Function" on page 621

LENGTHN Function

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

Category: Character

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

Syntax

LENGTHN(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Details

The LENGTHN function returns an integer that represents the position of the rightmost non-blank character in *string*. If the value of *string* is blank, LENGTHN returns a value of 0. If *string* is a numeric constant, variable, or expression (either initialized or uninitialized), SAS automatically converts the numeric value to a right-justified character string by using the BEST12. format. In this case, LENGTHN returns a value of 12 and writes a note in the SAS log stating that the numeric values have been converted to character values.

Comparisons

- The LENGTHN and LENGTH functions return the same value for non-blank character strings. LENGTHN returns a value of 0 for blank character strings, whereas LENGTH returns a value of 1.
- The LENGTHN function returns the length of a character string, excluding trailing blanks, whereas the LENGTHC function returns the length of a character string, including trailing blanks. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHC.
- The LENGTHN function returns the length of a character string, excluding trailing blanks, whereas the LENGTHM function returns the amount of memory in bytes that is allocated for a character string. LENGTHN always returns a value that is less than or equal to the value returned by LENGTHM.

Example

The following SAS statements produce these results.

SAS Statement	Result
`len=lengthn('ABCDEF');` `put len;`	`6`
`len2=lengthn(' ');` `put len2;`	`0`

See Also

Functions:

- "LENGTH Function" on page 617
- "LENGTHC Function" on page 618
- "LENGTHM Function" on page 619

LEXCOMB Function

Generates all distinct combinations of the non-missing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

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

Syntax

LEXCOMB(*count*, *k*, *variable-1*, …, *variable-n*)

Required Arguments

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

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

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

Requirement Initialize these variables before you execute the LEXCOMB function.

Tip After executing the LEXCOMB function, the first *k* variables contain the values in one combination.

Details

The Basics

Use the LEXCOMB function in a loop where the first argument to LEXCOMB takes each integral value from 1 to the number of distinct combinations of the non-missing values of the variables. In each execution of LEXCOMB within this loop, *k* should have the same value.

Number of Combinations

When all of the variables have non-missing, unequal values, then the number of combinations is COMB(*n*,*k*). If the number of variables that have missing values is m, and all the non-missing values are unequal, then LEXCOMB produces COMB(*n*-*m*,*k*) combinations because the missing values are omitted from the combinations.

When some of the variables have equal values, the exact number of combinations is difficult to compute, but COMB(*n*,*k*) provides an upper bound. You do not need to compute the exact number of combinations, provided that your program leaves the loop when LEXCOMB returns a value that is less than zero.

LEXCOMB Processing

On the first execution of the LEXCOMB function, the following actions occur:

- The argument types and lengths are checked for consistency.
- The m missing values are assigned to the last m arguments.
- The *n*-*m* non-missing values are assigned in ascending order to the first *n*-*m* arguments following *count*.
- LEXCOMB returns 1.

On subsequent executions, up to and including the last combination, the following actions occur:

- The next distinct combination of the non-missing values is generated in lexicographic order.
- If *variable-1* through *variable-i* did not change, but *variable-j* did change, where *j*=*i* +1, then LEXCOMB returns *j*.

If you execute the LEXCOMB function after generating all the distinct combinations, then LEXCOMB returns –1.

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

Comparisons

The LEXCOMB function generates all distinct combinations of the non-missing values of n variables taken *k* at a time in lexicographic order. The ALLCOMB function generates all combinations of the values of k variables taken *k* at a time in a minimum change order.

Examples

Example 1: Generating Distinct Combinations in Lexicographic Order

The following example uses the LEXCOMB function to generate distinct combinations in lexicographic order.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   n=dim(x);
   k=3;
   ncomb=comb(n,k);
   do j=1 to ncomb+1;
      rc=lexcomb(j, k, of x[*]);
      put j 5. +3 x1-x3 +3 rc=;
      if rc<0 then leave;
   end;
run;
```

SAS writes the following output to the log:

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

Example 2: Generating Distinct Combinations in Lexicographic Order: Another Example

The following is another example of using the LEXCOMB function.

```
data _null_;
   array x[5] $3 ('X' 'Y' 'Z' 'Z' 'Y');
   n=dim(x);
   k=3;
   ncomb=comb(n,k);
   do j=1 to ncomb+1;
      rc=lexcomb(j, k, of x[*]);
      put j 5. +3 x1-x3 +3 rc=;
      if rc<0 then leave;
   end;
run;
```

SAS writes the following output to the log:

```
    1   X Y Y    rc=1
    2   X Y Z    rc=3
    3   X Z Z    rc=2
    4   Y Y Z    rc=1
    5   Y Z Z    rc=2
    6   Y Z Z    rc=-1
```

See Also

Functions:

- "ALLCOMB Function" on page 95

CALL Routines:

- "CALL LEXCOMB Routine" on page 177

LEXCOMBI Function

Generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order.

Category: Combinatorial

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

Syntax

LEXCOMBI(*n*, *k*, *index-1*, …, *k*)

Required Arguments

n
is a numeric constant, variable, or expression that specifies the total number of objects.

K
is a numeric constant, variable, or expression that specifies the number of objects in each combination.

index
is a numeric variable that contains indices of the objects in the combination that is returned. Indices are integers between 1 and *n* inclusive.

Tip If *index-1* is missing or zero, then the LEXCOMBI function initializes the indices to **`index-1=1`** through **`index-k=k`**. Otherwise, LEXCOMBI creates a new combination by removing one index from the combination and adding another index.

Details

Before the first execution of the LEXCOMBI function, complete one of the following tasks:

- Set *index-1* equal to zero or to a missing value.
- Initialize *index-1* through *index-k* to distinct integers between 1 and *n* inclusive.

The number of combinations of *n* objects taken *k* at a time can be computed as COMB(*n*,*k*). To generate all combinations of *n* objects taken *k* at a time, execute the LEXCOMBI function in a loop that executes COMB(*n*,*k*) times.

In the LEXCOMBI function, the returned value indicates which, if any, indices changed. If *index-1* through *index-i* did not change, but *index-j* did change, where*j*=*i*+1, then

LEXCOMBI returns*i*. If LEXCOMBI is called after the last combinations in lexicographic order have been generated, then LEXCOMBI returns –1.

Comparisons

The LEXCOMBI function generates all combinations of the indices of *n* objects taken *k* at a time in lexicographic order. The ALLCOMBI function generates all combinations of the indices of *n* objects taken *k* at a time in a minimum change order.

Example

The following example uses the LEXCOMBI function to generate combinations of indices in lexicographic order.

```
data _null_;
   array x[5] $3 ('ant' 'bee' 'cat' 'dog' 'ewe');
   array c[3] $3;
   array i[3];
   n=dim(x);
   k=dim(i);
   i[1]=0;
   ncomb=comb(n,k);
   do j=1 to ncomb+1;
      rc=lexcombi(n, k, of i[*]);
      do h=1 to k;
         c[h]=x[i[h]];
      end;
      put @4 j= @10 'i= ' i[*] +3 'c= ' c[*] +3 rc=;
   end;
run;
```

SAS writes the following output to the log:

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

See Also

CALL Routines:

- “CALL LEXCOMBI Routine” on page 180
- “CALL ALLCOMBI Routine” on page 153

LEXPERK Function

Generates all distinct permutations of the non-missing values of *n* variables taken *k* at a time in lexicographic order.

Category: Combinatorial

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

Syntax

LEXPERK(*count*, *k*, *variable-1*, …, *variable-n*)

Required Arguments

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

k
is a numeric constant, variable, or expression with an integer value between 1 and *n* inclusive.

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

Requirement Initialize these variables before you execute the LEXPERK function.

Tip After executing LEXPERK, the first *k* variables contain the values in one permutation.

Details

The Basics

Use the LEXPERK function in a loop where the first argument to LEXPERK takes each integral value from 1 to the number of distinct permutations of *k* non-missing values of the variables. In each execution of LEXPERK within this loop, *k* should have the same value.

Number of Permutations

When all of the variables have non-missing, unequal values, the number of permutations is PERM(*n*,*k*). If the number of variables that have missing values is *m*, and all the non-missing values are unequal, the LEXPERK function produces PERM(*n*-*m*,*k*) permutations because the missing values are omitted from the permutations. When some of the variables have equal values, the exact number of permutations is difficult to compute, but PERM(*n*,*k*) provides an upper bound. You do not need to compute the exact number of permutations, provided you exit the loop when the LEXPERK function returns a value that is less than zero.

LEXPERK Processing

On the first execution of the LEXPERK function, the following actions occur:

- The argument types and lengths are checked for consistency.

- The *m* missing values are assigned to the last *m* arguments.
- The *n-m* non-missing values are assigned in ascending order to the first *n-m* arguments following *count*.
- LEXPERK returns 1.

On subsequent executions, up to and including the last permutation, the following actions occur:

- The next distinct permutation of *k* non-missing values is generated in lexicographic order.
- If *variable-1* through *variable-i* did not change, but *variable-i* did change, where *j=i* +1, then LEXPERK returns *j*.

If you execute the LEXPERK function after generating all the distinct permutations, then LEXPERK returns –1.

If you execute the LEXPERK function with the first argument out of sequence, then the results are not useful. In particular, if you initialize the variables and then immediately execute the LEXPERK function with a first argument of *j*, you will not get the j^{th} permutation (except when *j* is 1). To get the j^{th} permutation, you must execute the LEXPERK function *j* times, with the first argument taking values from 1 through *j* in that exact order.

Comparisons

The LEXPERK function generates all distinct permutations of the non-missing values of *n* variables taken *k* at a time in lexicographic order. The LEXPERM function generates all distinct permutations of the non-missing values of *n* variables in lexicographic order. The ALLPERM function generates all permutations of the values of several variables in a minimal change order.

Example

The following is an example of the LEXPERK function.

```
data _null_;
   array x[5] $3 ('X' 'Y' 'Z' 'Z' 'Y');
   n=dim(x);
   k=3;
   nperm=perm(n,k);
   do j=1 to nperm+1;
      rc=lexperk(j, k, of x[*]);
      put j 5. +3 x1-x3 +3 rc=;
      if rc<0 then leave;
   end;
run;
```

SAS writes the following output to the log:

```
    1   X Y Y    rc=1
    2   X Y Z    rc=3
    3   X Z Y    rc=2
    4   X Z Z    rc=3
    5   Y X Y    rc=1
    6   Y X Z    rc=3
    7   Y Y X    rc=2
    8   Y Y Z    rc=3
    9   Y Z X    rc=2
```

```
10   Y Z Y    rc=3
11   Y Z Z    rc=3
12   Z X Y    rc=1
13   Z X Z    rc=3
14   Z Y X    rc=2
15   Z Y Y    rc=3
16   Z Y Z    rc=3
17   Z Z X    rc=2
18   Z Z Y    rc=3
19   Z Z Y    rc=-1
```

See Also

Functions:

- “ALLPERM Function” on page 97
- “LEXPERM Function” on page 629

CALL Routines:

- “CALL RANPERK Routine” on page 224
- “CALL RANPERM Routine” on page 226

LEXPERM Function

Generates all distinct permutations of the non-missing values of several variables in lexicographic order.

Category: Combinatorial

Syntax

LEXPERM(*count*, *variable-1* <, …, *variable-N*>)

Required Arguments

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

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

Requirement Initialize these variables before you execute the LEXPERM function.

Details

Determine the Number of Distinct Permutations

The following variables are defined for use in the equation that follows:

N
specifies the number of variables that are being permuted—that is, the number of arguments minus one.

M
specifies the number of missing values among the variables that are being permuted.

d
specifies the number of distinct non-missing values among the arguments.

N_i
for i=1, through i=d, N_i specifies the number of instances of the i*th* distinct value.

The number of distinct permutations of non-missing values of the arguments is expressed as follows:

$$P = \frac{(N_1 + N_2 + \ldots + N_d)!}{N_1 ! \, N_2 ! \ldots N_d!} <= N!$$

Note: The LEXPERM function cannot be executed with the %SYSFUNC macro.

LEXPERM Processing

Use the LEXPERM function in a loop where the argument *count* takes each integral value from 1 to P. You do not need to compute P provided you exit the loop when LEXPERM returns a value that is less than zero.

For 1=*count*<P, the following actions occur:

- The argument types and lengths are checked for consistency.
- The M missing values are assigned to the last M arguments.
- The N-M non-missing values are assigned in ascending order to the first N-M arguments following *count*.
- LEXPERM returns 1.

For 1<*count*<=P, the following actions occur:

- The next distinct permutation of the non-missing values is generated in lexicographic order.
- If *variable-1* through *variable-I* did not change, but *variable-J* did change, where J=I +1, then LEXPERM returns J.

For *count*>P, LEXPERM returns –1.

If the LEXPERM function is executed with the first argument out of sequence, the results might not be useful. In particular, if you initialize the variables and then immediately execute LEXPERM with a first argument of K, you will not get the K*th* permutation (except when K is 1). To get the K*th* permutation, you must execute LEXPERM K times, with the first argument accepting values from 1 through K in that exact order.

Comparisons

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

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

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

Example

The following is an example of the LEXPERM function.

```
data _null_;
   array x[6] $1 ('X' 'Y' 'Z' ' ' 'Z' 'Y');
   nfact=fact(dim(x));
   put +3 nfact=;
   do i=1 to nfact;
      rc=lexperm(i, of x[*]);
      put i 5. +2 rc= +2 x[*];
      if rc<0 then leave;
   end;
run;
```

SAS writes the following output to the log:

```
   nfact=720
    1  rc=1   X Y Y Z Z
    2  rc=3   X Y Z Y Z
    3  rc=4   X Y Z Z Y
    4  rc=2   X Z Y Y Z
    5  rc=4   X Z Y Z Y
    6  rc=3   X Z Z Y Y
    7  rc=1   Y X Y Z Z
    8  rc=3   Y X Z Y Z
    9  rc=4   Y X Z Z Y
   10  rc=2   Y Y X Z Z
   11  rc=3   Y Y Z X Z
   12  rc=4   Y Y Z Z X
   13  rc=2   Y Z X Y Z
   14  rc=4   Y Z X Z Y
   15  rc=3   Y Z Y X Z
   16  rc=4   Y Z Y Z X
   17  rc=3   Y Z Z X Y
   18  rc=4   Y Z Z Y X
   19  rc=1   Z X Y Y Z
   20  rc=4   Z X Y Z Y
   21  rc=3   Z X Z Y Y
   22  rc=2   Z Y X Y Z
   23  rc=4   Z Y X Z Y
   24  rc=3   Z Y Y X Z
   25  rc=4   Z Y Y Z X
   26  rc=3   Z Y Z X Y
   27  rc=4   Z Y Z Y X
   28  rc=2   Z Z X Y Y
   29  rc=3   Z Z Y X Y
   30  rc=4   Z Z Y Y X
   31  rc=-1   Z Z Y Y X
```

See Also

Functions:

- “ALLPERM Function” on page 97

CALL Routines:

- “CALL ALLPERM Routine” on page 156
- “CALL RANPERK Routine” on page 224
- “CALL RANPERM Routine” on page 226

LFACT Function

Computes the logarithm of the FACT (factorial) function.

Category: Combinatorial

Syntax

LFACT(*n*)

Required Argument

n
is an integer that represents the total number of elements from which the sample is chosen.

Details

The LFACT function computes the logarithm of the FACT function.

Example

The following SAS statements produce these results.

SAS Statements	Results
x=lfact(5000); put x;	37591.143509
y=lfact(100); put y;	363.73937556

See Also

Functions:

- “FACT Function” on page 399

LGAMMA Function

Returns the natural logarithm of the Gamma function.

Category: Mathematical

Syntax

LGAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=lgamma(2);`	`0`
`x=lgamma(1.5);`	`-0.120782238`

LIBNAME Function

Assigns or deassigns a libref for a SAS library.

Category: SAS File I/O

See: "LIBNAME Function: Windows" in *SAS Companion for Windows*

Syntax

LIBNAME(*libref*<,*SAS-library*<,*engine*<,*options*> > >)

Required Argument

libref
is a character constant, variable, or expression that specifies the libref that is assigned to a SAS library.

Tip The maximum length of *libref* is eight characters.

Optional Arguments

SAS-library
is a character constant, variable, or expression that specifies the physical name of the SAS library that is associated with the libref. Specify this name as required by the host operating environment. This argument can be null.

engine
is a character constant, variable, or expression that specifies the engine that is used to access SAS files opened in the data library. If you are specifying a SAS/SHARE server, then the value of engine should be REMOTE. This argument can be null.

options
is a character constant, variable, or expression that specifies one or more valid options for the specified engine, delimited with blanks. This argument can be null.

Details

Basic Information about Return Codes

The LIBNAME function assigns or deassigns a libref from a SAS library. When you use the LIBNAME function with two or more arguments, SAS attempts to assign the libref. When you use one argument, SAS attempts to deassign the libref. Return codes are generated depending on the value of the arguments that are used in the LIBNAME function and whether the libref is assigned.

When assigning a libref, the return code will be 0 if the libref is successfully assigned. If the return code is nonzero and the SYSMSG function returns a warning message or a note, then the assignment was successful. If the SYSMSG function returns an error, then the assignment was unsuccessful.

If a library is already assigned, and you attempt to assign a different name to the library, the libref is assigned, the LIBNAME function returns a nonzero return code, and the SYSMSG function returns a note.

When LIBNAME Has One Argument

When LIBNAME has one argument, the following rules apply:

- If the libref is not assigned, a nonzero return code is returned and the SYSMSG function returns a warning message.
- If the libref is successfully assigned, a 0 return code is returned and the SYSMSG function returns a blank value.

When LIBNAME Has Two Arguments

When LIBNAME has two arguments, the following rules apply:

- If the second argument is NULL, all blanks, or zero length, SAS attempts to deassign the libref.
- If the second argument is not NULL, not all blanks, and not zero length, SAS attempts to assign the specified path (the second argument) to the libref.
- If the libref is not assigned, a nonzero return code is returned and the SYSMSG function returns an error message.
- If the libref is successfully assigned, a 0 return code is returned and the SYSMSG function returns a blank value.

When LIBNAME Has Three or Four Arguments

- If the second argument is NULL, all blanks, or zero length, the results depend on your operating environment.
- If the second argument is NULL and the libref is not already assigned, then a nonzero return code is returned and the SYSMSG function returns an error message.
- If the second argument is NULL and the libref has already been assigned, then LIBNAME returns a value of 0 and the SYSMSG function returns a blank value.
- If at least one of the previous conditions is not met, then SAS attempts to assign the specified path (second argument) to the libref.
- If the engine is not a SAS engine (for example, ODBC), then the second argument, *SAS-library*, must be missing or empty and cannot contain " or TRIMN(). If the *SAS-library* argument does contain a value, then the libname is not assigned.

  These two examples use an ODBC engine to assign a libname:

```
rc3 = libname ('mylib3',  , 'ODBC', 'DSN=mySQLServer_11');
/**Argument2, SAS-library is missing**/

rc3 = libname ('mylib3',, 'ODBC', 'DSN=mySQLServer_11');
/** Argument2, SAS-library is missing**/
```

  This example uses an ODBC engine but does not assign a libname because TRIMN is used:

```
rc3 = libname ('mylib3',TRIMN(&variable.), 'ODBC', 'DSN=mySQLServer_11');
```

Note: In the DATA step, a character constant that consists of two consecutive quotation marks without any intervening spaces is interpreted as a single space, not as a string with a length of 0. To specify a string with a length of 0, use the TRIMN Function on page 926.

Operating Environment Information

Some systems allow a *SAS-library* value of " " (a space between the single quotation marks) to assign a libref to the current directory. Other systems disassociate the libref from the SAS library when the *SAS-library* value contains only blanks. The behavior of LIBNAME when a single space is specified for *SAS-library* is dependent on your operating environment.Under some operating environments, you can assign librefs by using system commands that are outside the SAS session.

Examples

Example 1: Assigning a Libref

This example attempts to assign the libref NEW to the SAS library MYLIB. If an error or warning occurs, the message is written to the SAS log. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%if (%sysfunc(libname(new,MYLIB))) %then
  %put %sysfunc(sysmsg());
```

Example 2: Deassigning a Libref

This example deassigns the libref NEW that was previously associated with the data library MYLIB in the preceding example. If an error or warning occurs, the message is written to the SAS log. In a macro statement, you do not enclose character strings in quotation marks.

```
%if (%sysfunc(libname(new))) %then
```

```
%put %sysfunc(sysmsg());
```

Example 3: Compressing a Library

This example assigns the libref NEW to the MYLIB data library and uses the COMPRESS option to compress the library. This example uses the default SAS engine. In a DATA step, you enclose character strings in quotation marks.

```
data test;
   rc=libname('new','MYLIB',,'compress=yes');
run;
```

See Also

Functions:

- "LIBREF Function" on page 636
- "SYSMSG Function" on page 905

LIBREF Function

Verifies that a libref has been assigned.

Category: SAS File I/O

Syntax

LIBREF(*libref*)

Required Argument

libref

specifies the libref to be verified. In a DATA step, *libref* can be a character expression, a string enclosed in quotation marks, or a DATA step variable whose value contains the libref. In a macro, *libref* can be any expression.

Range 1 to 8 characters

Details

The LIBREF function returns 0 if the libref has been assigned, or returns a nonzero value if the libref has not been assigned.

Example

This example verifies a libref. If an error or warning occurs, the message is written to the log. Under some operating environments, the user can assign librefs by using system commands outside the SAS session.

```
%if (%sysfunc(libref(sashelp))) %then
  %put %sysfunc(sysmsg());
```

See Also

Functions:

- "LIBNAME Function" on page 633

LOG Function

Returns the natural (base e) logarithm.

Category: Mathematical

Syntax

LOG(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=log(1.0);`	`0`
`x=log(10.0);`	`2.302585093`

LOG1PX Function

Returns the log of 1 plus the argument.

Category: Mathematical

Syntax

LOG1PX(*x*)

Required Argument

x
specifies a numeric variable, constant, or expression.

Details

The LOG1PX function computes the log of 1 plus the argument. The LOG1PX function is mathematically defined by the following equation, where –1 < x:

$$LOG1PX(x) = log(1 + x)$$

When x is close to 0, **LOG1PX(x)** can be more accurate than **LOG(1+x)**.

Examples

Example 1: Computing the Log with the LOG1PX Function

The following example computes the log of 1 plus the value 0.5.

```
data _null_;
   x=log1px(0.5);
   put x=;
run;
```

SAS writes the following output to the Log:

```
x=0.4054651081
```

Example 2: Comparing the LOG1PX Function with the LOG Function

In the following example, the value of X is computed by using the LOG1PX function. The value of Y is computed by using the LOG function.

```
   data _null_;
      x=log1px(1.e-5);
      put x= hex16.;
      y=log(1+1.e-5);
      put y= hex16.;
   run;
```

SAS writes the following output to the Log:

```
x=3EE4F8AEA9AE7317
y=3EE4F8AEA9AF0A25
```

See Also

Functions:

- "LOG Function" on page 637

LOG10 Function

Returns the logarithm to the base 10.

Category: Mathematical

Syntax

LOG10(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=log10(1.0);`	`0`
`x=log10(10.0);`	`1`
`x=log10(100.0);`	`2`

LOG2 Function

Returns the logarithm to the base 2.

Category: Mathematical

Syntax

LOG2(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Range must be positive.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=log2(2.0);`	`1`
`x=log2(0.5);`	`-1`

LOGBETA Function

Returns the logarithm of the beta function.

Category: Mathematical

Syntax

LOGBETA(*a*,*b*)

Required Arguments

a
is the first shape parameter, where $a>0$.

b
is the second shape parameter, where $b>0$.

Details

The LOGBETA function is mathematically given by the equation
$log(\beta(a, b)) = log(\Gamma(a)) + log(\Gamma(b)) - log(\Gamma(a + b))$

where $\Gamma(.)$ is the gamma function.

If the expression cannot be computed, LOGBETA returns a missing value.

Example

The following SAS statement produces this result.

SAS Statement	Result
`LOGBETA(5,3);`	`-4.653960350`

See Also

Functions:

- "BETA Function" on page 136

LOGCDF Function

Returns the logarithm of a left cumulative distribution function.

Category: Probability

See: "CDF Function" on page 277

Syntax

LOGCDF(*'dist'*,*quantile*<,*parm-1*,...,*parm-k*>)

Required Arguments

'dist'
is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	**'BERNOULLI'**
Beta	**'BETA'**
Binomial	**'BINOMIAL'**
Cauchy	**'CAUCHY'**
Chi-Square	**'CHISQUARE'**
Exponential	**'EXPONENTIAL'**
F	**'F'**
Gamma	**'GAMMA'**
Generalized Poisson	**'GENPOISSON'**
Geometric	**'GEOMETRIC'**
Hypergeometric	**'HYPERGEOMETRIC'**
Laplace	**'LAPLACE'**
Logistic	**'LOGISTIC'**
Lognormal	**'LOGNORMAL'**
Negative binomial	**'NEGBINOMIAL'**
Normal	**'NORMAL'** | **'GAUSS'**
Normal mixture	**'NORMALMIX'**
Pareto	**'PARETO'**
Poisson	**'POISSON'**
T	**'T'**

Distribution	Argument
Tweedie	`'TWEEDIE'`
Uniform	`'UNIFORM'`
Wald (inverse Gaussian)	`'WALD'` | `'IGAUSS'`
Weibull	`'WEIBULL'`

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric variable, constant, or expression that specifies the value of a random variable.

Optional Argument

parm-1,...,parm-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The LOGCDF function computes the logarithm of a left cumulative distribution function (logarithm of the left side) from various continuous and discrete distributions. For more information, see the "CDF Function" on page 277.

See Also

Functions:

- "CDF Function" on page 277
- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856
- "QUANTILE Function" on page 799
- "SQUANTILE Function" on page 881

LOGPDF Function

Returns the logarithm of a probability density (mass) function.

Category: Probability

Alias: LOGPMF

See: "PDF Function" on page 722

Syntax

LOGPDF(*'dist'*,*quantile*,*parm-1*,...,*parm-k*)

Required Arguments

'dist'

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	**'BERNOULLI'**
Beta	**'BETA'**
Binomial	**'BINOMIAL'**
Cauchy	**'CAUCHY'**
Chi-Square	**'CHISQUARE'**
Exponential	**'EXPONENTIAL'**
F	**'F'**
Gamma	**'GAMMA'**
Generalized Poisson	**'GENPOISSON'**
Geometric	**'GEOMETRIC'**
Hypergeometric	**'HYPERGEOMETRIC'**
Laplace	**'LAPLACE'**
Logistic	**'LOGISTIC'**
Lognormal	**'LOGNORMAL'**
Negative binomial	**'NEGBINOMIAL'**
Normal	**'NORMAL'** | **'GAUSS'**
Normal mixture	**'NORMALMIX'**
Pareto	**'PARETO'**
Poisson	**'POISSON'**

Distribution	Argument
T	`'T'`
Tweedie	`'TWEEDIE'`
Uniform	`'UNIFORM'`
Wald (inverse Gaussian)	`'WALD'` | `'IGAUSS'`
Weibull	`'WEIBULL'`

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of a random variable.

parm-1,...,parm-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The LOGPDF function computes the logarithm of the probability density (mass) function from various continuous and discrete distributions. For more information, see the "PDF Function" on page 722.

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856
- "QUANTILE Function" on page 799
- "SQUANTILE Function" on page 881

LOGSDF Function

Returns the logarithm of a survival function.

Category: Probability

See: "SDF Function" on page 856

Syntax

LOGSDF(*'dist'*,*quantile*,*parm-1*,...,*parm-k*)

Required Arguments

'dist'

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	`'BERNOULLI'`
Beta	`'BETA'`
Binomial	`'BINOMIAL'`
Cauchy	`'CAUCHY'`
Chi-Square	`'CHISQUARE'`
Exponential	`'EXPONENTIAL'`
F	`'F'`
Gamma	`'GAMMA'`
Generalized Poisson	`'GENPOISSON'`
Geometric	`'GEOMETRIC'`
Hypergeometric	`'HYPERGEOMETRIC'`
Laplace	`'LAPLACE'`
Logistic	`'LOGISTIC'`
Lognormal	`'LOGNORMAL'`
Negative binomial	`'NEGBINOMIAL'`
Normal	`'NORMAL'` | `'GAUSS'`
Normal mixture	`'NORMALMIX'`
Pareto	`'PARETO'`
Poisson	`'POISSON'`
T	`'T'`

Distribution	Argument
Tweedie	`'TWEEDIE'`
Uniform	`'UNIFORM'`
Wald (inverse Gaussian)	`'WALD'` | `'IGAUSS'`
Weibull	`'WEIBULL'`

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of a random variable.

parm-1,...,parm-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The LOGSDF function computes the logarithm of the survival function from various continuous and discrete distributions. For more information, see the "SDF Function" on page 856.

See Also

Functions:

- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642
- "CDF Function" on page 277
- "PDF Function" on page 722
- "SDF Function" on page 856
- "QUANTILE Function" on page 799
- "SQUANTILE Function" on page 881

LOWCASE Function

Converts all letters in an argument to lowercase.

Category: Character

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

Syntax

LOWCASE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the LOWCASE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

The LOWCASE function copies the character argument, converts all uppercase letters to lowercase letters, and returns the altered value as a result.

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

Example

The following SAS statements produce these results.

SAS Statement	Result

`x='INTRODUCTION';`
`y=lowcase(x);`
`put y;`
	`introduction`

See Also

Functions:

- "UPCASE Function" on page 928
- "PROPCASE Function" on page 773

LPERM Function

Computes the logarithm of the PERM function, which is the logarithm of the number of permutations of *n* objects, with the option of including *r* number of elements.

Category: Combinatorial

Syntax

LPERM(*n*<,*r*>)

Required Argument

n
is an integer that represents the total number of elements from which the sample is chosen.

Optional Argument

r

is an optional integer value that represents the number of chosen elements. If *r* is omitted, the function returns the factorial of *n*.

Restriction $r \leq n$

Details

The LPERM function computes the logarithm of the PERM function.

Example

The following SAS statements produce these results.

SAS Statement	Result
x=lperm(5000,500); put x;	4232.7715946
y=lperm(100,10); put y;	45.586735935

See Also

Functions:

- "PERM Function" on page 743

LPNORM Function

Returns the L*p* norm of the second argument and subsequent non-missing arguments.

Category: Descriptive Statistics

Syntax

LPNORM(*p*, *value-1* <,*value-2* …>)

Required Arguments

p

specifies a numeric constant, variable, or expression that is greater than or equal to 1, which is used as the power for computing the L_p norm.

value

specifies a numeric constant, variable, or expression.

Details

If all arguments have missing values, then the result is a missing value. Otherwise, the result is the L_p norm of the non-missing values of the second and subsequent arguments.

In the following example, *p* is the value of the first argument, and *x1*, *x2*, ..., *xn* are the values of the other non-missing arguments.

$$LPNORM(p, x1, x2, ..., xn) = \left(abs(x1)^p + abs(x2)^p + ... + abs(xn)^p\right)^{1/p}$$

Examples

Example 1: Calculating the Lp Norm

The following example returns the L_p norm of the second and subsequent non-missing arguments.

```
data _null_;
   x1 = lpnorm(1, ., 3, 0, .q, -4);
   x2 = lpnorm(2, ., 3, 0, .q, -4);
   x3 = lpnorm(3, ., 3, 0, .q, -4);
   x999 = lpnorm(999, ., 3, 0, .q, -4);
   put x1= / x2= / x3= / x999=;
run;
```

SAS writes the following output to the log:

```
x1=7
x2=5
x3=4.4979414453
x999=4
```

Example 2: Calculating the Lp Norm When You Use a Variable List

The following example uses a variable list and returns the L_p norm.

```
   data _null_;
      x1 = 1;
      x2 = 3;
      x3 = 4;
      x4 = 3;
      x5 = 1;
      x = lpnorm(of x1-x5);
      put x=;
   run;
```

SAS writes the following output to the log:

```
x=11
```

See Also

Functions:

- "SUMABS Function" on page 899 (L1 norm)
- "EUCLID Function" on page 395 (L2 norm)
- "MAX Function" on page 655 (Linfinity norm)

MAD Function

Returns the median absolute deviation from the median.

Category: Descriptive Statistics

Syntax

MAD(*value-1* <, *value-2*...>)

Required Argument

value
specifies a numeric constant, variable, or expression of which the median absolute deviation from the median is to be computed.

Details

If all arguments have missing values, the result is a missing value. Otherwise, the result is the median absolute deviation from the median of the non-missing 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*.

Example

The following SAS statements produce these results.

SAS Statement	Result
`mad=mad(2,4,1,3,5,999999);` `put mad;`	`1.5`

See Also

Functions:

- “IQR Function” on page 599
- “MEDIAN Function” on page 659
- “PCTL Function” on page 720

MARGRCLPRC Function

Calculates call prices for European options on stocks, based on the Margrabe model.

Category: Financial

Syntax

MARGRCLPRC(*X*$_1$, *t*, *X*$_2$, *sigma1*, *sigma2*, *rho12*)

Required Arguments

***X*$_1$**

is a nonmissing, positive value that specifies the price of the first asset.

Requirement Specify X_1 and X_2 in the same units.

t

is a nonmissing value that specifies the time to expiration.

***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.

sigma1

is a nonmissing, positive fraction that specifies the volatility of the first asset.

Requirement *sigma1* must be for the same time period as the unit of *t*.

sigma2

is a nonmissing, positive fraction that specifies the volatility of the second asset.

Requirement Specify a value for *sigma2* for the same time period as the unit of *t*.

rho12

specifies the correlation between the first and second assets, $\rho_{x_1 x_2}$.

Range between –1 and 1

Details

The MARGRCLPRC function calculates the call price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$CALL = X_1 N(d_1) - X_2 N(d_2)$$

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:

$$CALL = \max\left(\left(X_1 - X_2\right), 0\right)$$

Note: This function assumes that there are no dividends from the two assets.

For information about the basics of pricing, see "Using Pricing Functions" on page 8.

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 SAS statements produce these results.

SAS Statement	Result
	`----+----1----+---2--`
`a=margrclprc(500, .5, 950, 4, 5, 1);` `put a;`	`46.441283642`

SAS Statement	Result
`b=margrclprc(850, 1.2, 125, 5, 3, 1);` `put b;`	`777.67008185`
`c=margrclprc(7500, .9, 950, 3, 2, 1);` `put c;`	`6562.0354886`
`d=margrclprc(5000, -.5, 237, 3, 3, 1);` `put d;`	`0`

See Also

Functions:

- "MARGRPTPRC Function" on page 653

MARGRPTPRC Function

Calculates put prices for European options on stocks, based on the Margrabe model.

Category: Financial

Syntax

MARGRPTPRC(X_1, t, X_2, *sigma1*, *sigma2*, *rho12*)

Required Arguments

X_1

is a nonmissing, positive value that specifies the price of the first asset.

Requirement Specify X_1 and X_2 in the same units.

t

is a nonmissing value that specifies the time to expiration.

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.

sigma1

is a nonmissing, positive fraction that specifies the volatility of the first asset.

Requirement *sigma1* must be for the same time period as the unit of *t*.

sigma2

is a nonmissing, positive fraction that specifies the volatility of the second asset.

Requirement Specify a value for *sigma2* for the same time period as the unit of *t*.

rho12
specifies the correlation between the first and second assets, $\rho_{x_1 y_1}$.

Range between –1 and 1

Details

The MARGRPTPRC function calculates the put price for European options on stocks, based on the Margrabe model. The function is based on the following relationship:

$$PUT = X_2 N(pd_1) - X_1 N(pd_2)$$

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.

$\sigma_{x_1}^2$
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.

To view the corresponding CALL relationship, see the "MARGRCLPRC Function" on page 650.

For the special case of t=0, the following equation is true:

$$PUT = \max\left((X_2 - X_1), 0\right)$$

Note: This function assumes that there are no dividends from the two assets.

For basic information about pricing, see "Using Pricing Functions" on page 8.

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 SAS statements produce these results.

SAS Statement	Result
	`----+----1----+---2--`
`a=margrptprc(500, .5, 950, 4, 5, 1);` `put a;`	`496.44128364`
`b=margrptprc(850, 1.2, 125, 5, 3, 1);` `put b;`	`52.670081846`
`c=margrptprc(7500, .9, 950, 3, 2, 1);` `put c;`	`12.035488581`
`d=margrptprc(5000, -.5, 237, 3, 3, 1);` `put d;`	`0`

See Also

Functions:

- "MARGRCLPRC Function" on page 650

MAX Function

Returns the largest value.

Category: Descriptive Statistics

Syntax

MAX(*argument-1*,*argument-2*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two arguments are required. The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The MAX function returns a missing value (.) only if all arguments are missing.

The MAX operator (<>) returns a missing value only if both operands are missing. In this case, it returns the value of the operand that is higher in the sort order for missing values.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=max(8,3);`	8
`x1=max(2,6,.);`	6
`x2=max(2,-3,1,-1);`	2
`x3=max(3,.,-3);`	3
`x4=max(of x1-x3);`	6

MD5 Function

Returns the result of the message digest of a specified string.

Category: Character

Syntax

MD5(*string*)

Required Argument

string
specifies a character constant, variable, or expression.

Tip Enclose a literal string of characters in quotation marks.

Details

Length of Returned Variable

In a DATA step, if the MD5 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 MD5 function converts a string, based on the MD5 algorithm, into a 128-bit hash value. This hash value is referred to as a message digest (digital signature), which is nearly unique for each string that is passed to the function.

The MD5 function does not format its own output. You must specify a valid format (such as hex32. or binary128.) to view readable results.

Operating Environment Information

In the z/OS operating environment, the MD5 function produces output in EBCDIC rather than in ASCII. Therefore, the output will differ.

The Message Digest Algorithm

A message digest results from manipulating and compacting an arbitrarily long stream of binary data. An ideal message digest algorithm never generates the same result for two different sets of input. However, generating such a unique result would require a message digest as long as the input itself. Therefore, MD5 generates a message digest of modest size (16 bytes), created with an algorithm that is designed to make a nearly unique result.

Using the MD5 Function

You can use the MD5 function to track changes in your data sets. The MD5 function can generate a digest of a set of column values in a record in a table. This digest could be treated as the signature of the record, and be used to keep track of changes that are made to the record. If the digest from the new record matches the existing digest of a record in a table, then the two records are the same. If the digest is different, then a column value in the record has changed. The new changed record could then be added to the table along with a new surrogate key because it represents a change to an existing keyed value.

The MD5 function can be useful when developing shell scripts or Perl programs for software installation, for file comparison, and for detection of file corruption and tampering.

You can also use the MD5 function to create a unique identifier for observations to be used as the key of a hash object. For information about hash objects, see “Introduction to DATA Step Component Objects” in Chapter 22 of *SAS Language Reference: Concepts*.

Example

The following is an example of how to generate results that are returned by the MD5 function.

```
data _null_;
   y = md5('abc');
   z = md5('access method');
   put y= / y = hex32.;
   put z= / z = hex32.;
run;
```

The output from this program contains unprintable characters.

MDY Function

Returns a SAS date value from month, day, and year values.

Category: Date and Time

Syntax

MDY(*month*,*day*,*year*)

Required Arguments

month
specifies a numeric constant, variable, or expression that represents an integer from 1 through 12.

day
specifies a numeric constant, variable, or expression that represents an integer from 1 through 31.

year
specifies a numeric constant, variable, or expression with a value of a two-digit or four-digit integer that represents the year. The YEARCUTOFF= system option defines the year value for two-digit dates.

Example

The following SAS statements produce these results.

SAS Statement	Result

`birthday=mdy(8,27,90);`
`put birthday;`
`put birthday= worddate.;`
	`11196`
	`birthday=August 27, 1990`

`anniversary=mdy(7,11,2001);`
`put anniversary;`
`put anniversary=date9.;`
	`15167`
	`anniversary=11JUL2001`

See Also

Functions:

- “DAY Function” on page 360
- “MONTH Function” on page 669
- “YEAR Function” on page 984

MEAN Function

Returns the arithmetic mean (average).

Category: Descriptive Statistics

Syntax

MEAN(*argument-1*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one non-missing argument is required. Otherwise, the function returns a missing value.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

The GEOMEAN function returns the geometric mean, the HARMEAN function returns the harmonic mean, and the MEDIAN function returns the median of the non-missing values, whereas the MEAN function returns the arithmetic mean (average).

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=mean(2,.,.,6);`	4
`x2=mean(1,2,3,2);`	2
`x3=mean(of x1-x2);`	3

See Also

Functions:

- “GEOMEAN Function” on page 511
- “GEOMEANZ Function” on page 513
- “HARMEAN Function” on page 526
- “HARMEANZ Function” on page 527
- “MEDIAN Function” on page 659

MEDIAN Function

Returns the median value.

Category: Descriptive Statistics

Syntax

MEDIAN(*value1*<, *value2*, ...>)

Required Argument

value
is a numeric constant, variable, or expression.

Details

The MEDIAN function returns the median of the nonmissing values. If all arguments have missing values, the result is a missing value.

Note: The formula that is used in the MEDIAN function is the same as the formula that is used in PROC UNIVARIATE. For more information, see SAS Elementary Statistics Procedures.

Comparisons

The MEDIAN function returns the median of nonmissing values, whereas the MEAN function returns the arithmetic mean (average).

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=median(2,4,1,3);`	`2.5`
`y=median(5,8,0,3,4);`	`4`

See Also

Functions:

- "MEAN Function" on page 658

MIN Function

Returns the smallest value.

Category: Descriptive Statistics

Syntax

MIN(*argument-1*,*argument-2*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two arguments are required. The argument list can consist of a variable list, which is preceded by OF.

Details

The MIN function returns a missing value (.) only if all arguments are missing.

The MIN operator (><) returns a missing value if either operand is missing. In this case, it returns the value of the operand that is lower in the sort order for missing values.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=min(7,4);`	`4`
`x1=min(2,.,6);`	`2`
`x2=min(2,-3,1,-1);`	`-3`
`x3=min(0,4);`	`0`
`x4=min(of x1-x3);`	`-3`

MINUTE Function

Returns the minute from a SAS time or datetime value.

Category: Date and Time

Syntax

MINUTE(*time* | *datetime*)

Required Arguments

time
is a numeric constant, variable, or expression that specifies a SAS time value.

datetime
is a numeric constant, variable, or expression that specifies a SAS datetime value.

Details

The MINUTE function returns an integer that represents a specific minute of the hour. MINUTE always returns a positive number in the range of 0 through 59.

Example

The following SAS statements produce these results.

SAS Statement	Result

`time='3:19:24't;`
`m=minute(time);`
`put m;`
	`19`

See Also

Functions:

- "HOUR Function" on page 534
- "SECOND Function" on page 859

MISSING Function

Returns a numeric result that indicates whether the argument contains a missing value.

Categories: Descriptive Statistics
Character

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

Syntax

MISSING(*numeric-expression* | *character-expression*)

Required Arguments

numeric-expression
specifies a numeric constant, variable, or expression.

character-expression
specifies a character constant, variable, or expression.

Details

- The MISSING function checks a numeric or character expression for a missing value, and returns a numeric result. If the argument does not contain a missing value, SAS returns a value of 0. If the argument contains a missing value, SAS returns a value of 1.
- A numeric expression is considered missing if it evaluates to a numeric missing value: ., ._, .A, ..., .Z.
- A character expression is considered missing if it evaluates to a string that contains all blanks or has a length of zero.

Comparisons

The MISSING function can have only one argument. The CMISS function can have multiple arguments and returns a count of the missing values. The NMISS function requires numeric arguments and returns the number of missing values in the list of arguments.

Example

This example uses the MISSING function to check whether the input variables contain missing values.

```
data values;
   input @1 var1 3. @5 var2 3.;
   if missing(var1) then
      do;
         put 'Variable 1 is Missing.';
      end;
   else if missing(var2) then
      do;
         put 'Variable 2 is Missing.';
      end;
   datalines;
127
988 195
;
run;
```

SAS writes the following output to the log:

```
Variable 2 is Missing.
```

See Also

Functions:

- “CMISS Function” on page 303
- “NMISS Function” on page 683

CALL Routines:

- “CALL MISSING Routine” on page 191

MOD Function

Returns the remainder from the division of the first argument by the second argument, fuzzed to avoid most unexpected floating-point results.

Category: Mathematical

Syntax

MOD (*argument-1*,*argument-2*)

Required Arguments

argument-1
is a numeric constant, variable, or expression that specifies the dividend.

argument-2
is a numeric constant, variable, or expression that specifies the divisor.

Restriction cannot be 0

Details

The MOD function returns the remainder from the division of *argument-1* by *argument-2*. When the result is non-zero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MOD function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representations.

  To determine the largest integer for which the computation is exact, execute the following DATA step:

```
data _null_;
   exactint = constant('exactint');
   put exactint=;
run;
```

  Operating Environment Information
  For information about the largest integer, see the SAS documentation for your operating environment.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. In this case

- MOD returns zero if the remainder is very close to zero or very close to the value of the second argument.
- MOD returns a missing value if the remainder cannot be computed to a precision of approximately three digits or more. In this case, SAS also writes an error message to the log.

Note: Before SAS 9, the MOD function did not perform the adjustments to the remainder that were described in the previous paragraph. For this reason, the results of the MOD function in SAS 9 might differ from previous versions.

Comparisons

Here are some comparisons between the MOD and MODZ functions:

- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- The MODZ function performs no fuzzing.
- Both the MOD and MODZ functions return a missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

The following SAS statements produce results for MOD and MODZ.

SAS Statement	Result
x1=mod(10,3); put x1 9.4;	1.0000
xa=modz(10,3); put xa 9.4;	1.0000
x2=mod(.3,-.1); put x2 9.4;	0.0000
xb=modz(.3,-.1); put xb 9.4;	0.1000
x3=mod(1.7,.1); put x3 9.4;	0.0000
xc=modz(1.7,.1); put xc 9.4;	0.0000
x4=mod(.9,.3); put x4 24.20;	0.00000000000000000000
xd=modz(.9,.3); put xd 24.20;	0.00000000000000005551

See Also

Functions:

- “INT Function” on page 555
- “INTZ Function” on page 596
- “MODZ Function” on page 667

MODEXIST Function

Determines whether a software image exists in the version of SAS that you have installed.

Category: Numeric

Syntax

MODEXIST('*product-name*')

Required Argument

'*product-name*'
specifies a character constant, variable, or expression that is the name of the product image that you are checking.

Details

The MODEXIST function determines whether a software image exists in the version of SAS that you have installed. If an image exists, then MODEXIST returns a value of 1. If an image does not exist, then MODEXIST returns a value of 0.

Comparisons

The MODEXIST function determines whether a software image exists in the version of SAS that you have installed. The SYSPROD function determines whether a product is licensed.

Example

This example determines whether a product is licensed and the image is installed. The example returns a value of 1 if a SAS/GRAPH image is installed in your version of SAS, and returns a value of 0 if the image is not installed. The SYSPROD function determines whether the product is licensed.

```
data _null_;
   rc1 = sysprod('graph');
   rc2 = modexist('sasgplot');
   put rc1= rc2=;
run;
```

Log 2.14 *Output from MODEXIST*

```
rc1=1 rc2=1
```

MODULEC Function

Calls an external routine and returns a character value.

Category: External Routines

See: "CALL MODULE Routine" on page 192

Syntax

MODULEC(<*cntl-string*,> *module-name*<,*argument-1, ..., argument-n*>)

Details

For details about the MODULEC function, see "CALL MODULE Routine" on page 192.

See Also

Functions:

- "MODULEN Function" on page 667

CALL Routines:

- "CALL MODULE Routine" on page 192

MODULEN Function

Calls an external routine and returns a numeric value.

Category: External Routines

See: "CALL MODULE Routine" on page 192

Syntax

MODULEN(<*cntl-string*,> *module-name*<,*argument-1, ..., argument-n*>)

Details

For details about the MODULEN function, see "CALL MODULE Routine" on page 192.

See Also

Functions:

- "MODULEC Function" on page 666

CALL Routines:

- "CALL MODULE Routine" on page 192

MODZ Function

Returns the remainder from the division of the first argument by the second argument, using zero fuzzing.

Category: Mathematical

Syntax

MODZ (*argument-1*, *argument-2*)

Required Arguments

argument-1
is a numeric constant, variable, or expression that specifies the dividend.

argument-2
is a non-zero numeric constant, variable, or expression that specifies the divisor.

Details

The MODZ function returns the remainder from the division of *argument-1* by *argument-2*. When the result is non-zero, the result has the same sign as the first argument. The sign of the second argument is ignored.

The computation that is performed by the MODZ function is exact if both of the following conditions are true:

- Both arguments are exact integers.
- All integers that are less than either argument have exact 8-byte floating-point representation.

  To determine the largest integer for which the computation is exact, execute the following DATA step:

```
data _null_;
   exactint = constant('exactint');
   put exactint=;
run;
```

  Operating Environment Information
  For information about the largest integer, see the SAS documentation for your operating environment.

If either of the above conditions is not true, a small amount of numerical error can occur in the floating-point computation. For example, when you use exact arithmetic and the result is zero, MODZ might return a very small positive value or a value slightly less than the second argument.

Comparisons

Here are some comparisons between the MODZ and MOD functions:

- The MODZ function performs no fuzzing.
- The MOD function performs extra computations, called fuzzing, to return an exact zero when the result would otherwise differ from zero because of numerical error.
- Both the MODZ and MOD functions return a missing value if the remainder cannot be computed to a precision of approximately three digits or more.

Example

The following SAS statements produce these results for MOD and MODZ.

SAS Statement	Result
`x1=mod(10,3);` `put x1 9.4;`	`1.0000`
`xa=modz(10,3);` `put xa 9.4;`	`1.0000`
`x2=mod(.3,-.1);` `put x2 9.4;`	`0.0000`
`xb=modz(.3,-.1);` `put xb 9.4;`	`0.1000`
`x3=mod(1.7,.1);` `put x3 9.4;`	`0.0000`

SAS Statement	Result
`xc=modz(1.7,.1);` `put xc 9.4;`	`0.0000`
`x4=mod(.9,.3);` `put x4 24.20;`	`0.00000000000000000000`
`xd=modz(.9,.3);` `put xd 24.20;`	`0.00000000000000005551`

See Also

Functions:

- "INT Function" on page 555
- "INTZ Function" on page 596
- "MOD Function" on page 663

MONTH Function

Returns the month from a SAS date value.

Category: Date and Time

Syntax

MONTH(*date*)

Required Argument

date
specifies a numeric constant, variable, or expression that represents a SAS date value.

Details

The MONTH function returns a numeric value that represents the month from a SAS date value. Numeric values can range from 1 through 12.

Example

The following SAS statements produce this result.

SAS Statement	Result

`date='25jan94'd;`
`m=month(date);`
`put m;`
	`1`

See Also

Functions:

- "DAY Function" on page 360
- "YEAR Function" on page 984

MOPEN Function

Opens a file by directory ID and member name, and returns either the file identifier or a 0.

Category: External Files

See: "MOPEN Function: UNIX" in *SAS Companion for UNIX Environments*
"MOPEN Function: z/OS" in *SAS Companion for z/OS*

Syntax

MOPEN(*directory-id*,*member-name*<,*open-mode*<,*record-length*<,*record-format*> > >)

Required Arguments

directory-id
is a numeric variable that specifies the identifier that was assigned when the directory was opened, generally by the DOPEN function.

member-name
is a character constant, variable, or expression that specifies the member name in the directory.

Optional Arguments

open-mode
is a character constant, variable, or expression that specifies the type of access to the file:

A APPEND mode allows writing new records after the current end of the file.

I INPUT mode allows reading only (default).

O OUTPUT mode defaults to the OPEN mode specified in the operating environment option in the FILENAME statement or function. If no operating environment option is specified, it allows writing new records at the beginning of the file.

S Sequential input mode is used for pipes and other sequential devices such as hardware ports.

U UPDATE mode allows both reading and writing.

W Sequential update mode is used for pipes and other sequential devices such as ports.

Default I

record-length
is a numeric variable, constant, or expression that specifies a new logical record length for the file. To use the existing record length for the file, specify a length of 0, or do not provide a value here.

record-format
is a character constant, variable, or expression that specifies a new record format for the file. To use the existing record format, do not specify a value here. The following values are valid:

B specifies that data is to be interpreted as binary data.

D specifies the default record format.

E specifies the record format that you can edit.

F specifies that the file contains fixed-length records.

P specifies that the file contains printer carriage control in operating environment-dependent record format.

V specifies that the file contains variable-length records.

Note: If an argument is invalid, then MOPEN returns 0. You can obtain the text of the corresponding error message from the SYSMSG function. Invalid arguments do not produce a message in the SAS log and do not set the _ERROR_ automatic variable.

Details

MOPEN returns the identifier for the file, or 0 if the file could not be opened. You can use a *file-id* that is returned by the MOPEN function as you would use a *file-id* returned by the FOPEN function.

CAUTION:
Use OUTPUT mode with care. Opening an existing file for output might overwrite the current contents of the file without warning.

The member is identified by *directory-id* and *member-name* instead of by a fileref. You can also open a directory member by using FILENAME to assign a fileref to the member, followed by a call to FOPEN. However, when you use MOPEN, you do not have to use a separate fileref for each member.

If the file already exists, the output and update modes default to the operating environment option (append or replace) specified with the FILENAME statement or function. For example,

```
%let rc=%sysfunc(filename(file,physical-name,,mod));
%let did=%sysfunc(dopen(&file));
%let fid=%sysfunc(mopen(&did,member-name,o,0,d));
%let rc=%sysfunc(fput(&fid,This is a test.));
%let rc=%sysfunc(fwrite(&fid));
%let rc=%sysfunc(fclose(&fid));
```

If **'file'** already exists, FWRITE appends the new record instead of writing it at the beginning of the file. However, if no operating environment option is specified with the FILENAME function, the output mode implies that the record be replaced.

If the open fails, use SYSMSG to retrieve the message text.

Operating Environment Information
The term *directory* in this description refers to an aggregate grouping of files that are managed by the operating environment. Different host operating environments

identify such groupings with different names, such as directory, subdirectory, folder, MACLIB, or partitioned data set. For details, see the SAS documentation for your operating environment.Opening a directory member for output or append is not possible in some operating environments.

Example

This example assigns the fileref MYDIR to a directory. Then it opens the directory, determines the number of members, retrieves the name of the first member, and opens that member. The last three arguments to MOPEN are the defaults. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%let filrf=mydir;
%let rc=%sysfunc(filename(filrf,physical-name));
%let did=%sysfunc(dopen(&filrf));
%let frstname=' ';
%let memcount=%sysfunc(dnum(&did));
%if (&memcount > 0) %then
   %do;
      %let frstname =
         %sysfunc(dread(&did,1));
      %let fid =
         %sysfunc(mopen(&did,&frstname,i,0,d));
      macro statements to process the member
      %let rc=%sysfunc(fclose(&fid));
   %end;
%else
   %put %sysfunc(sysmsg());
%let rc=%sysfunc(dclose(&did));
```

See Also

Functions:

- “DCLOSE Function” on page 361
- “DNUM Function” on page 381
- “DOPEN Function” on page 382
- “DREAD Function” on page 386
- “FCLOSE Function” on page 402
- “FILENAME Function” on page 411
- “FOPEN Function” on page 485
- “FPUT Function” on page 494
- “FWRITE Function” on page 501
- “SYSMSG Function” on page 905

MORT Function

Returns amortization parameters.

Category: Financial

Syntax

MORT(*a*,*p*,*r*,*n*)

Required Arguments

a
is numeric, and specifies the initial amount.

p
is numeric, and specifies the periodic payment.

r
is numeric, and specifies the periodic interest rate that is expressed as a fraction.

n
is an integer, and specifies the number of compounding periods.

Range $n \geq 0$

Details

Calculating Results

The MORT function returns the missing argument in the list of four arguments from an amortization calculation with a fixed interest rate that is compounded each period. The arguments are related by the following equation:

$$p = \frac{ar(1 + r)^n}{(1 + r)^n - 1}$$

One missing argument must be provided. The value is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

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 statement, an amount of $50,000 is borrowed for 30 years at an annual interest rate of 10 percent compounded monthly. The monthly payment can be expressed as follows:

```
payment=mort(50000, . , .10/12,30*12);
```

The value that is returned is 438.79 (rounded). The second argument has been set to missing, which indicates that the periodic payment is to be calculated. The 10 percent

nominal annual rate has been converted to a monthly rate of 0.10/12. The rate is the fractional (not the percentage) interest rate per compounding period. The 30 years are converted to 360 months.

MSPLINT Function

Returns the ordinate of a monotonicity-preserving interpolating spline.

Category: Mathematical

Syntax

MSPLINT(*X*, *n*, X_1 <, X_2, …, X_n> , Y_1 <,Y_2, …, Y_n> <,D_1,D_n>)

Required Arguments

X
is a numeric constant, variable, or expression that specifies the abscissa for which the ordinate of the spline is to be computed.

n
is a numeric constant, variable, or expression that specifies the number of knots. *N* must be a positive integer.

X_1, …, X_n
are numeric constants, variables, or expressions that specify the abscissas of the knots. These values must be non-missing and listed in nondecreasing order. Otherwise, the result is undefined. MSPLINT does not check the order of the X_1 through X_n arguments.

Y_1, ..., Y_n
are numeric constants, variables, or expressions that specify the ordinates of the knots. The number of Y_1 through Y_n arguments must be the same as the number of X_1 throughX_n arguments.

Optional Argument

D_1, D_n
are optional numeric constants, variables, or expressions that specify the derivatives of the spline at X_1 and X_n. These derivatives affect only abscissas that are less than X_2 or greater than X_{n-1} .

Details

The MSPLINT function returns the ordinate of a monotonicity-preserving cubic interpolating spline for a single abscissa, *X*.

An interpolating spline is a function that passes through each point that is specified by the ordered pairs (X_1, Y_1), (X_2, Y_2), …, (X_n, Y_n). These points are called knots.

A spline preserves monotonicity if both of the following conditions are true:

- For any two or more consecutive knots with nondecreasing ordinates, all interpolated values within that interval are also nondecreasing.

- For any two or more consecutive knots with nonincreasing ordinates, all interpolated values within that interval are also nonincreasing.

However, if you specify values of D_1 or D_n with the wrong sign, monotonicity will not be preserved for values that are less than X_2 or greater than X_{n-1}.

If the arguments D_1 and D_n are omitted or missing, then the following actions occur:

- For n=1, MSPLINT returns Y_1.
- For n=2, MSPLINT uses linear interpolation or extrapolation.

If the arguments D_1 and D_n have non-missing values, or if $n >= 3$, then the following actions occur:

- If $X < X_1$ or $X > X_n$, MSPLINT uses linear extrapolation.
- If $X_1 <= X <= X_n$, MSPLINT uses cubic spline interpolation.

If two knots have equal abscissas but different ordinates, then the spline will be discontinuous at that abscissa. If two knots have equal abscissas and equal ordinates, then the spline will be continuous at that abscissa, but the first derivative will usually be discontinuous at that abscissa. Otherwise, the spline is continuous and has a continuous first derivative.

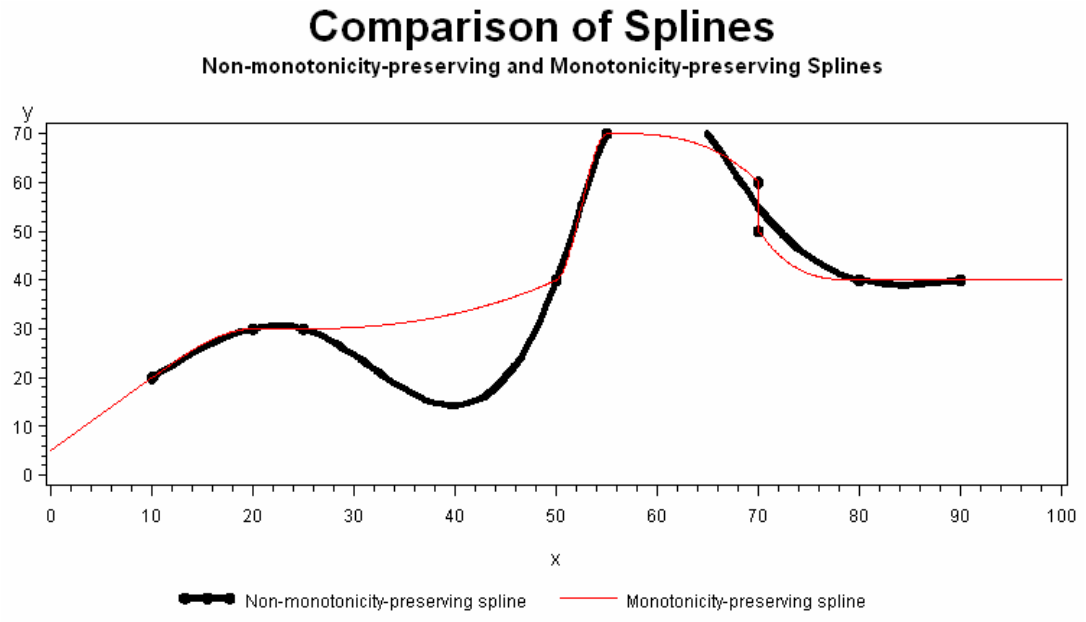
If X is missing, or if any other arguments required to compute the result are missing, then MSPLINT returns a missing value. MSPLINT does not check all of the arguments for missing values. Because the arguments D_1 and D_n are optional, and they are not required to compute the result, if one or both are missing and no errors occur, then MSPLINT returns a non-missing result.

Example

The following is an example of the MSPLINT function.

```
data msplint;
   do x=0 to 100 by .1;
      msplint=msplint(x, 9,
         10, 20, 25, 50, 55, 70, 70, 80, 90,
         20, 30, 30, 40, 70, 60, 50, 40, 40);
      output;
   end;
run;
data knots;
   input x y;
   datalines;
10 20
20 30
25 30
50 40
55 70
70 60
70 50
80 40
90 40
;
data plot;
   merge knots msplint;
   by x;
```

```
run;
title "Comparison of Splines";
title2 "Non-monotonicity-preserving and Monotonicity-preserving
        Splines";
legend1 value=('Non-monotonicity-preserving spline'
               'Monotonicity-preserving spline') label=none;
symbol1 value=dot  interpol=spline color=black width=5;
symbol2 value=none interpol=join   color=red;
proc gplot data=plot;
   plot y*x=1 msplint*x=2/overlay legend=legend1;
run;
quit;
```



References

Fritsch, F. N., and J. Butland. “A method for constructing local monotone piecewise cubic interpolants.” 1984. *Siam Journal of Scientific and Statistical Computing* 5:2: 300-304.

MVALID Function

Checks the validity of a character string for use as a SAS member name.

Category: Character

Syntax

MVALID(*libname*, *string*, *memtype*<,*validmemname*>)

Required Arguments

libname
specifies a character constant, variable, or expression that associates a SAS library with a libref. Leading and trailing blanks are ignored.

string
specifies a character constant, variable, or expression that is checked to determine whether its value can be used as a SAS member name. Leading and trailing blanks are ignored.

memtype
specifies a character constant, variable, or expression that is the member type of the member name that you are using. Leading and trailing blanks are ignored. The value of *memtype* is not validated. The following member types are available:

ACCESS	specifies access descriptor files that are created by SAS/ACCESS.
CATALOG	specifies SAS catalogs.
DATA	specifies SAS data files.
FDB	specifies a financial database.
ITEMSTOR	specifies a SAS data set that consists of pieces of information that can be accessed independently. The SAS Registry is an example of an item store.
MDDB	specifies a multidimensional database.
PROGRAM	specifies stored compiled SAS programs.
VIEW	specifies SAS views.

Optional Argument

validmemname
specifies a character constant, variable, or expression. The values for *validmemname* can be uppercase or lowercase. Leading and trailing blanks are ignored. The following list contains the values that you can use with *validmemname*:

COMPAT
COMPATIBLE
determines that *string* is a valid SAS member name when all three of the following conditions are true:

- The *string* argument begins with an English letter or an underscore.
- All subsequent characters are English letters, underscores, or digits.
- The length of *string* is 32 or fewer alphanumeric characters.

EXTEND
determines that *string* is a valid SAS member name when all of the following conditions are true:

- The length of *string* is 32 or fewer bytes.
- The *string* argument does not contain the characters / \ * ? " < > | : –

  Note: The SPD Engine additionally does not allow '$' as the first character. It also does not allow a period (.) in the member name.
- The *string* argument does not contain null bytes.

- The *string* argument does not begin with a blank or period (.).
- The *string* argument contains at least one character. A name that consists of all blanks is not valid.

Default VALIDMEMNAME= is set to COMPAT.

Note If no value is specified, the MVALUE function determines that *string* is a valid SAS member name based on the value of the VALIDMEMNAME= system option.

Details

The Basics

The MVALID function checks the value of *string* to determine whether it can be used as a SAS member name.

The MVALID function returns a value of 1 if *string* can be used as a SAS member name, and a value of 0 if *string* cannot be used as a SAS member name.

MVALID returns a missing value if one of the following conditions is true:

- The *libname* argument is not an assigned libref.
- The *memtype* argument is longer than nine characters.
- The *validmemname* argument does not have one of the following values: COMPATIBLE, COMPAT, or EXTEND, regardless of whether the value is uppercase or lowercase.

Requirements for Validation of a SAS Member Name

The *string* argument is evaluated to determine whether it is a valid SAS member name. An engine name with its associated library, as well as member type, affect the validation of *string*. Of the member types, only DATA, ITEMSTOR, and VIEW allow names with extended characters. When *string* is evaluated, the EXTEND value of the optional *validmemname* argument is taken into account. Not all engines support *validmemname* processing. For the engines that do not, *string* is validated based on the rules for that engine.

The following example shows you how to use the MVALID function to determine whether *string* is a valid SAS member name, based on engine name, DATA member type, and the EXTEND value for *validmemname*:

```
libname V9eng V9 'mypath';
data _null_;
   rc=MVALID('V9eng', 'my name', 'data', 'extend');
   put rc=;
run;
```

The following items apply to the preceding example:

- The example returns a value of 1, indicating that '*my name*' is a valid member name for the V9 engine when member type equals DATA and *validmemname* equals EXTEND.
- If you use the V6 engine in the example, the program returns a value of 0, indicating that '*my name*' is not valid when member type equals DATA and *validmemname* equals EXTEND. The V6 engine does not support *validmemname* processing.

In the following example, CATALOG is used instead of DATA for member type:

```
libname V9eng V9 'mypath';
data _null_;
   rc=MVALID('V9eng', 'my name', 'catalog', 'extend');
   put rc=;
run;
```

The following items apply to the preceding example:

- If you use CATALOG in the example instead of DATA, the program returns a value of 0, indicating that '*my name*' is not valid when member type equals CATALOG and *validmemname* equals EXTEND. The member type CATALOG does not support extended names, and therefore the EXTEND value for *validmemname* is not valid.
- If you use COMPAT in the example instead of EXTEND, the program returns a value of 0, indicating that '*my name*' is not valid when member type equals CATALOG and *validmemname* equals COMPAT. The COMPAT value of *validmemname* does not allow spaces in member names.

N Function

Returns the number of nonmissing numeric values.

Category: Descriptive Statistics

Syntax

N(*argument-1*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one argument is required. The argument list can consist of a variable list, which is preceded by OF.

Comparisons

The N function counts nonmissing values, whereas the NMISS and the CMISS functions count missing values. N requires numeric arguments, whereas CMISS works with both numeric and character values.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=n(1,0,.,2,5,.);`	4
`x2=n(1,2);`	2
`x3=n(of x1-x2);`	2

NETPV Function

Returns the net present value as a percent.

Category: Financial

Syntax

NETPV(*r*,*freq*,*c0*,*c1*,...,*cn*)

Required Arguments

r

is numeric, the interest rate over a specified base period of time expressed as a fraction.

Range $r >= 0$

freq

is numeric, the number of payments during the base period of time that is specified with the rate *r*.

Range $freq > 0$

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.

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:

$$NETPV(r,\ freq,\ c_0,\ c_1,\ ...,\ c_n) = \sum_{i=0}^{n} c_i x^i$$

The following relationship applies to the preceding equation:

$$x = \begin{cases} \dfrac{1}{(1+r)^{(1/\,freq)}} & freq > 0 \\ \varepsilon^{-r} & freq = 0 \end{cases}$$

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 percent 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 percent 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 percent, the net present value of the investment can be expressed as follows:

```
value=netpv(.10,.5,-500,200,300,400);
```

The value returned is 95.98.

NLITERAL Function

Converts a character string that you specify to a SAS name literal.

Category: Character

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

Syntax

NLITERAL(*string*)

Required Argument

string
specifies a character constant, variable, or expression that is to be converted to a SAS name literal.

Restriction If the string is a valid SAS variable name, it is not changed.

Tip Enclose a literal string of characters in quotation marks.

Details

Length of Returned Variable

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

The Basics

String will be converted to a name literal, unless it qualifies under the default rules for a SAS variable name. These default rules are in effect when the SAS system option VALIDVARNAME=V7:

- It begins with an English letter or an underscore.
- All subsequent characters are English letters, underscores, or digits.
- The length is 32 or fewer alphanumeric characters.

String qualifies as a SAS variable name, when all of these rules are true.

The NLITERAL function encloses the value of *string* in single or double quotation marks, based on the contents of *string*.

Value in *string*	Result
an ampersand (&)	enclosed in single quotation marks
a percent sign (%)	enclosed in single quotation marks
more double quotation marks than single quotation marks	enclosed in single quotation marks
none of the above	enclosed in double quotation marks

If insufficient space is available for the resulting n-literal, NLITERAL returns a blank string, prints an error message, and sets _ERROR_ to 1.

Example

This example demonstrates multiple uses of NLITERAL.

```
data test;
   input string $32.;
   length result $ 67;
   result = nliteral(string);
   datalines;
abc_123
This and That
cats & dogs
Company's profits (%)
"Double Quotes"
'Single Quotes'
;
proc print;
title 'Strings Converted to N-Literals or Returned Unchanged';
run;
```

Display 2.43 *Converting Strings to Name Literals with NLITERAL*

Strings Converted to N-Literals or Returned Unchanged

Obs	string	result
1	abc_123	abc_123
2	This and That	"This and That"N
3	cats & dogs	'cats & dogs'N
4	Company's profits (%)	'Company''s profits (%)'N
5	"Double Quotes"	'"Double Quotes"'N
6	'Single Quotes'	"'Single Quotes'"N

See Also

Functions:

- "COMPARE Function" on page 311
- "DEQUOTE Function" on page 368
- "NVALID Function" on page 711

System Options:

- "VALIDVARNAME= System Option" in *SAS System Options: Reference*

Other References:

- "Words in the SAS Language" in Chapter 3 of *SAS Language Reference: Concepts*

NMISS Function

Returns the number of missing numeric values.

Category: Descriptive Statistics

Syntax

NMISS(*argument-1*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one argument is required. The argument list can consist of a variable list, which is preceded by OF.

Details

The NMISS function returns the number of missing values, whereas the N function returns the number of nonmissing values. NMISS requires numeric values, whereas CMISS works with both numeric and character values. NMISS works with multiple numeric values, whereas MISSING works with only one value that can be either numeric or character.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=nmiss(1,0,.,2,5,.);`	2
`x2=nmiss(1,0);`	0
`x3=nmiss(of x1-x2);`	0

NOMRATE Function

Returns the nominal annual interest rate.

Category: Financial

Syntax

NOMRATE(*compounding-interval*, *rate*)

Required Arguments

compounding-interval
is a SAS interval. This value represents how often the returned value is compounded.

rate
is numeric. *rate* is the effective annual interest rate (expressed as a percentage) that is compounded at each interval.

Details

The NOMRATE function returns the nominal annual interest rate. NOMRATE computes the nominal annual interest rate that corresponds to an effective annual interest rate.

The following details apply to the NOMRATE function:

- The values for rates must be at least –99.
- In considering an effective interest rate and a compounding interval, if *compounding-interval* is 'CONTINUOUS', then the value that is returned by NOMRATE equals $\log_e(1+[rate/100])$.

If *compounding-interval* is not 'CONTINUOUS', and *m* intervals occur in a year, the value that is returned by NOMRATE equals the following:

$$m\left(\left(1+\frac{rate}{100}\right)^{\frac{1}{m}}-1\right)$$

- 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);
```

NORMAL Function

Returns a random variate from a normal, or Gaussian, distribution.

Category: Random Number

Alias: RANNOR

See: “RANNOR Function” on page 821

NOTALNUM Function

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

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTALNUM function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase letter. If such a character is found, NOTALNUM returns the position in the string of that character. If no such character is found, NOTALNUM returns a value of 0.

If you use only one argument, NOTALNUM begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALNUM returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALNUM function searches a character string for a non-alphanumeric character. The ANYALNUM function searches a character string for an alphanumeric character.

Example

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

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

```
            put +3 j= c=;
          end;
        end;
    run;
```

The following lines are written to the SAS log:

```
j=5 c=
j=6 c==
j=7 c=
j=12 c=
j=13 c=+
j=14 c=
j=16 c=;
That's all
```

See Also

Functions:

- "ANYALNUM Function" on page 99

NOTALPHA Function

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

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTALPHA function searches a string for the first occurrence of any character that is not an uppercase or lowercase letter. If such a character is found, NOTALPHA returns

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

If you use only one argument, NOTALPHA begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTALPHA returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTALPHA function searches a character string for a nonalphabetic character. The ANYALPHA function searches a character string for an alphabetic character.

Examples

Example 1: Searching a String for Nonalphabetic Characters

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

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

The following lines are written to the SAS log:

```
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
```

```
   j=18 c=;
   That's all
```

Example 2: Identifying Control Characters by Using the NOTALPHA Function

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

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

See Also

Functions:

- "ANYALPHA Function" on page 101

NOTCNTRL Function

Searches a character string for a character that is not a control character, and returns the first position at which that character is found.

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTCNTRL function searches a string for the first occurrence of a character that is not a control character. If such a character is found, NOTCNTRL returns the position in the string of that character. If no such character is found, NOTCNTRL returns a value of 0.

If you use only one argument, NOTCNTRL begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTCNTRL returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTCNTRL function searches a character string for a character that is not a control character. The ANYCNTRL function searches a character string for a control character.

Example

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

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

See Also

Functions:

- "ANYCNTRL Function" on page 103

NOTDIGIT Function

Searches a character string for any character that is not a digit, and returns the first position at which that character is found.

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTDIGIT function searches a string for the first occurrence of any character that is not a digit. If such a character is found, NOTDIGIT returns the position in the string of that character. If no such character is found, NOTDIGIT returns a value of 0.

If you use only one argument, NOTDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTDIGIT function searches a character string for any character that is not a digit. The ANYDIGIT function searches a character string for a digit.

Example

The following example uses the NOTDIGIT function to search for a character that is not a digit.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
```

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

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=16 c=E
j=18 c=;
That's all
```

See Also

Functions:

- "ANYDIGIT Function" on page 105

NOTE Function

Returns an observation ID for the current observation of a SAS data set.

Category: SAS File I/O

Syntax

NOTE(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

Details

You can use the observation ID value to return to the current observation by using POINT. Observations can be marked by using NOTE and then returned to later by using POINT. Each observation ID is a unique numeric value.

To free the memory that is associated with an observation ID, use DROPNOTE.

Example

This example calls CUROBS to display the observation number, calls NOTE to mark the observation, and calls POINT to point to the observation that corresponds to NOTEID.

```
%let dsid=%sysfunc(open(sasuser.fitness,i));
  /* Go to observation 10 in data set */
%let rc=%sysfunc(fetchobs(&dsid,10));
%if %sysfunc(abs(&rc)) %then
  %put FETCHOBS FAILED;
%else
  %do;
      /* Display observation number     */
      /* in the Log                     */
    %let cur=%sysfunc(curobs(&dsid));
    %put CUROBS=&cur;
      /* Mark observation 10 using NOTE */
    %let noteid=%sysfunc(note(&dsid));
      /* Rewind pointer to beginning    */
      /* of data                        */
      /* set using REWIND               */
    %let rc=%sysfunc(rewind(&dsid));
      /* FETCH first observation into DDV */
    %let rc=%sysfunc(fetch(&dsid));
      /* Display first observation number */
    %let cur=%sysfunc(curobs(&dsid));
    %put CUROBS=&cur;
      /* POINT to observation 10 marked  */
      /* earlier by NOTE                 */
    %let rc=%sysfunc(point(&dsid,&noteid));
      /* FETCH observation into DDV */
    %let rc=%sysfunc(fetch(&dsid));
      /* Display observation number 10   */
      /* marked by NOTE                  */
    %let cur=%sysfunc(curobs(&dsid));
    %put CUROBS=&cur;
  %end;
%if (&dsid > 0) %then
  %let rc=%sysfunc(close(&dsid));
```

The following lines are written to the SAS log:

```
CUROBS=10
CUROBS=1
CUROBS=10
```

See Also

Functions:

- "DROPNOTE Function" on page 387
- "OPEN Function" on page 716
- "POINT Function" on page 746
- "REWIND Function" on page 831

NOTFIRST Function

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

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. These characters are any except the underscore (_) and uppercase or lowercase English letters. If such a character is found, NOTFIRST returns the position in the string of that character. If no such character is found, NOTFIRST returns a value of 0.

If you use only one argument, NOTFIRST begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTFIRST returns a value of zero when one of the following is true:

- The character that you are searching for is not found.

- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTFIRST function searches a string for the first occurrence of any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7. The ANYFIRST function searches a string for the first occurrence of any character that is valid as the first character in a SAS variable name under VALIDVARNAME=V7.

Example

The following example uses the NOTFIRST function to search a string for any character that is not valid as the first character in a SAS variable name under VALIDVARNAME=V7.

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

The following lines are written to the SAS log:

```
j=5 c=
j=6 c==
j=7 c=
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
j=18 c=;
That's all
```

See Also

Functions:

- “ANYFIRST Function” on page 106

NOTGRAPH Function

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

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTGRAPH function searches a string for the first occurrence of a non-graphical character. A graphical character is defined as any printable character other than white space. If such a character is found, NOTGRAPH returns the position in the string of that character. If no such character is found, NOTGRAPH returns a value of 0.

If you use only one argument, NOTGRAPH begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTGRAPH returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTGRAPH function searches a character string for a non-graphical character. The ANYGRAPH function searches a character string for a graphical character.

Examples

Example 1: Searching a String for Non-Graphical Characters

The following example uses the NOTGRAPH function to search a string for a non-graphical character.

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

The following lines are written to the SAS log:

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

Example 2: Identifying Control Characters by Using the NOTGRAPH Function

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

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

See Also

Functions:

- “ANYGRAPH Function” on page 108

NOTLOWER Function

Searches a character string for a character that is not a lowercase letter, and returns the first position at which that character is found.

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTLOWER function searches a string for the first occurrence of any character that is not a lowercase letter. If such a character is found, NOTLOWER returns the position in the string of that character. If no such character is found, NOTLOWER returns a value of 0.

If you use only one argument, NOTLOWER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTLOWER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTLOWER function searches a character string for a character that is not a lowercase letter. The ANYLOWER function searches a character string for a lowercase letter.

Example

The following example uses the NOTLOWER function to search a string for any character that is not a lowercase letter.

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

The following lines are written to the SAS log:

```
j=1 c=N
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
That's all
```

See Also

Functions:

- "ANYLOWER Function" on page 110

NOTNAME Function

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

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. These characters are any except underscore (_), digits, and uppercase or lowercase English letters. If such a character is found, NOTNAME returns the position in the string of that character. If no such character is found, NOTNAME returns a value of 0.

If you use only one argument, NOTNAME begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTNAME returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTNAME function searches a string for the first occurrence of any character that is not valid in a SAS variable name under VALIDVARNAME=V7. The ANYNAME function searches a string for the first occurrence of any character that is valid in a SAS variable name under VALIDVARNAME=V7.

Example

The following example uses the NOTNAME function to search a string for any character that is not valid in a SAS variable name under VALIDVARNAME=V7.

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

```
run;
```

The following lines are written to the SAS log:

```
j=5 c=
j=6 c==
j=7 c=
j=11 c=
j=12 c=+
j=13 c=
j=18 c=;
That's all
```

See Also

Functions:

- "ANYNAME Function" on page 112

NOTPRINT Function

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

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

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

If you use only one argument, NOTPRINT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*,

specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTPRINT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTPRINT function searches a character string for a non-printable character. The ANYPRINT function searches a character string for a printable character.

Example

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

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

See Also

Functions:

- "ANYPRINT Function" on page 113

NOTPUNCT Function

Searches a character string for a character that is not a punctuation character, and returns the first position at which that character is found.

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTPUNCT function searches a string for the first occurrence of a character that is not a punctuation character. If such a character is found, NOTPUNCT returns the position in the string of that character. If no such character is found, NOTPUNCT returns a value of 0.

If you use only one argument, NOTPUNCT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTPUNCT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

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

Examples

Example 1: Searching a String for Characters That Are Not Punctuation Characters

The following example uses the NOTPUNCT function to search a string for characters that are not punctuation characters.

```
data _null_;
   string='Next = _n_ + 12E3;';
   j=0;
   do until(j=0);
```

```
         j=notpunct(string,j+1);
         if j=0 then put +3 "That's all";
         else do;
            c=substr(string,j,1);
            put +3 j= c=;
         end;
      end;
   run;
```

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=7 c=
j=9 c=n
j=11 c=
j=13 c=
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
That's all
```

Example 2: Identifying Control Characters by Using the NOTPUNCT Function

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

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

See Also

Functions:

- “ANYPUNCT Function” on page 116

NOTSPACE Function

Searches a character string for a character that is not a white-space character (blank, horizontal and vertical tab, carriage return, line feed, and form feed), and returns the first position at which that character is found.

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTSPACE function searches a string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. If such a character is found, NOTSPACE returns the position in the string of that character. If no such character is found, NOTSPACE returns a value of 0.

If you use only one argument, NOTSPACE begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTSPACE returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTSPACE function searches a character string for the first occurrence of a character that is not a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed. The ANYSPACE function searches a character string for the first occurrence of a character that is a blank, horizontal tab, vertical tab, carriage return, line feed, or form feed.

Examples

Example 1: Searching a String for a Character That Is Not a White-Space Character

The following example uses the NOTSPACE function to search a string for a character that is not a white-space character.

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

The following lines are written to the SAS log:

```
j=1 c=N
j=2 c=e
j=3 c=x
j=4 c=t
j=6 c==
j=8 c=_
j=9 c=n
j=10 c=_
j=12 c=+
j=14 c=1
j=15 c=2
j=16 c=E
j=17 c=3
j=18 c=;
That's all
```

Example 2: Identifying Control Characters by Using the NOTSPACE Function

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

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

See Also

Functions:

- “ANYSPACE Function” on page 118

NOTUPPER Function

Searches a character string for a character that is not an uppercase letter, and returns the first position at which that character is found.

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

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

The NOTUPPER function searches a string for the first occurrence of a character that is not an uppercase letter. If such a character is found, NOTUPPER returns the position in the string of that character. If no such character is found, NOTUPPER returns a value of 0.

If you use only one argument, NOTUPPER begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTUPPER returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTUPPER function searches a character string for a character that is not an uppercase letter. The ANYUPPER function searches a character string for an uppercase letter.

Example

The following example uses the NOTUPPER function to search a string for any character that is not an uppercase letter.

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

The following lines are written to the SAS log:

```
j=2 c=e
j=3 c=x
j=4 c=t
j=5 c=
j=6 c==
j=7 c=
j=8 c=_
j=9 c=n
j=10 c=_
j=11 c=
j=12 c=+
j=13 c=
j=14 c=1
j=15 c=2
j=17 c=3
j=18 c=;
That's all
```

See Also

Functions:

- “ANYUPPER Function” on page 120

NOTXDIGIT Function

Searches a character string for a character that is not a hexadecimal character, and returns the first position at which that character is found.

Category: Character

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

Syntax

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

Required Argument

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

Optional Argument

start
is an optional numeric constant, variable, or expression with an integer value that specifies the position at which the search should start and the direction in which to search.

Details

The NOTXDIGIT function searches a string for the first occurrence of any character that is not a digit or an uppercase or lowercase A, B, C, D, E, or F. If such a character is found, NOTXDIGIT returns the position in the string of that character. If no such character is found, NOTXDIGIT returns a value of 0.

If you use only one argument, NOTXDIGIT begins the search at the beginning of the string. If you use two arguments, the absolute value of the second argument, *start*, specifies the position at which to begin the search. The direction in which to search is determined in the following way:

- If the value of *start* is positive, the search proceeds to the right.
- If the value of *start* is negative, the search proceeds to the left.
- If the value of *start* is less than the negative length of the string, the search begins at the end of the string.

NOTXDIGIT returns a value of zero when one of the following is true:

- The character that you are searching for is not found.
- The value of *start* is greater than the length of the string.
- The value of *start* = 0.

Comparisons

The NOTXDIGIT function searches a character string for a character that is not a hexadecimal character. The ANYXDIGIT function searches a character string for a character that is a hexadecimal character.

Example

The following example uses the NOTXDIGIT function to search a string for a character that is not a hexadecimal character.

```
data _null_;
   string='Next = _n_ + 12E3;';
```

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

The following lines are written to the SAS log:

```
   j=1 c=N
   j=3 c=x
   j=4 c=t
   j=5 c=
   j=6 c==
   j=7 c=
   j=8 c=_
   j=9 c=n
   j=10 c=_
   j=11 c=
   j=12 c=+
   j=13 c=
   j=18 c=;
   That's all
```

See Also

Functions:

- "ANYXDIGIT Function" on page 121

NPV Function

Returns the net present value with the rate expressed as a percentage.

Category: Financial

Syntax

NPV(*r*,*freq*,*c0*,*c1*,…,*cn*)

Required Arguments

r

is numeric, the interest rate over a specified base period of time expressed as a percentage.

freq

is numeric, the number of payments during the base period of time specified with the rate *r*.

Range *freq* > 0

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.

Comparisons

The NPV function is identical to NETPV, except that the *r* argument is provided as a percentage.

NVALID Function

Checks the validity of a character string for use as a SAS variable name.

Category: Character

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

Syntax

NVALID(*string*<,*validvarname*>)

Required Argument

string
specifies a character constant, variable, or expression which will be checked to determine whether its value can be used as a SAS variable name.

Note: Trailing blanks are ignored.

Tip Enclose a literal string of characters in quotation marks.

Optional Argument

validvarname
is a character constant, variable, or expression that specifies one of the following values:

V7
determines that *string* is a valid SAS variable name when all three of the following are true:

- *string* begins with an English letter or an underscore
- All subsequent characters are English letters, underscores, or digits
- The length is 32 or fewer alphanumeric characters

ANY
determines that *string* is a valid SAS variable name if it contains 32 or fewer characters of any type, including blanks.

NLITERAL
determines that *string* is a valid SAS variable name if it is in the form of a SAS name literal ('name'N) or if it is a valid SAS variable name when VALIDVARNAME=V7.

See V7 above in this same list.

Default If no value is specified, the NVALID function determines that *string* is a valid SAS variable name based on the value of the SAS system option VALIDVARNAME=.

Details

The NVALID function checks the value of *string* to determine whether it can be used as a SAS variable name.

The NVALID function returns a value of 1 or 0.

Condition	Returned Value
string can be used as a SAS variable name	1
string cannot be used as a SAS variable name	0

Example

This example determines the validity of specified strings as SAS variable names. The value that is returned by the NVALID function varies with the validvarname argument. The value of 1 is returned when the string is determined to be a valid SAS variable name under the rules for the specified validvarname argument. Otherwise, the value of 0 is returned.

```
options validvarname=v7 ls=64;
data string;
   input string $char40.;
   v7=nvalid(string,'v7');
   any=nvalid(string,'any');
   nliteral=nvalid(string,'nliteral');
   default=nvalid(string);
   datalines;
Tooooooooooooooooooooooooo Long
OK
Very_Long_But_Still_OK_for_V7
1st_char_is_a_digit
Embedded blank
!@#$%^&*
"Very Loooong N-Literal with """N
'No closing quotation mark
;

proc print noobs;
title1 'NLITERAL and Validvarname Arguments Determine';
title2 'Invalid (0) and Valid (1) SAS Variable Names';
run;
```

Display 2.44 *Determining the Validity of SAS Variable Names with NLITERAL*

NLITERAL and Validvarname Arguments Determine Invalid (0) and Valid (1) SAS Variable Names

string	v7	any	nliteral	default
Toooooooooooooooooooooooooo Long	0	0	0	0
OK	1	1	1	1
Very_Long_But_Still_OK_for_V7	1	1	1	1
1st_char_is_a_digit	0	1	1	0
Embedded blank	0	1	1	0
!@#$%^&*	0	1	1	0
"Very Loooong N-Literal with """N	0	0	1	0
'No closing quotation mark	0	1	0	0

See Also

Functions:

- "COMPARE Function" on page 311
- "NLITERAL Function" on page 681

System Options:

- "VALIDVARNAME= System Option" in *SAS System Options: Reference*

Other References:

- "Rules for Words and Names in the SAS Language" in Chapter 3 of *SAS Language Reference: Concepts*

NWKDOM Function

Returns the date for the *n*th occurrence of a weekday for the specified month and year.

Category: Date and Time

Syntax

NWKDOM(*n*, *weekday*, *month*, *year*)

Required Arguments

n
specifies the numeric week of the month that contains the specified day.

Range 1–5

Tip *N*=5 indicates that the specified day occurs in the last week of that month. Sometimes *n*=4 and *n*=5 produce the same results.

weekday
specifies the number that corresponds to the day of the week.

Range 1–7

Tip Sunday is considered the first day of the week and has a *weekday* value of 1.

month
specifies the number that corresponds to the month of the year.

Range 1–12

year
specifies a four-digit calendar year.

Details

The NWKDOM function returns a SAS date value for the *n*th weekday of the month and year that you specify. Use any valid SAS date format, such as the DATE9. format, to display a calendar date. You can specify *n*=5 for the last occurrence of a particular weekday in the month.

Sometimes *n*=5 and *n*=4 produce the same result. These results occur when there are only four occurrences of the requested weekday in the month. For example, if the month of January begins on a Sunday, there will be five occurrences of Sunday, Monday, and Tuesday, but only four occurrences of Wednesday, Thursday, Friday, and Saturday. In this case, specifying *n*=5 or *n*=4 for Wednesday, Thursday, Friday, or Saturday will produce the same result.

If the year is not a leap year, February has 28 days and there are four occurrences of each day of the week. In this case, *n*=5 and *n*=4 produce the same results for every day.

Comparisons

In the NWKDOM function, the value for *weekday* corresponds to the numeric day of the week beginning on Sunday. This value is the same value that is used in the WEEKDAY function, where Sunday =1, and so on. The value for *month* corresponds to the numeric month of the year beginning in January. This value is the same value that is used in the MONTH function, where January =1, and so on.

You can use the NWKDOM function to calculate events that are not defined by the HOLIDAY function. For example, if a university always schedules graduation on the first Saturday in June, then you can use the following statement to calculate the date:

```
UnivGrad = nwkdom(1, 7, 6, year);
```

Examples

Example 1: Returning Date Values

The following example uses the NWKDOM function and returns the date for specific occurrences of a weekday for a specified month and year.

```
data _null_;
      /* Return the date of the third Monday in May 2000. */
   a=nwkdom(3, 2, 5, 2000);
      /* Return the date of the fourth Wednesday in November 2007. */
   b=nwkdom(4, 4, 11, 2007);
      /* Return the date of the fourth Saturday in November 2007. */
   c=nwkdom(4, 7, 11, 2007);
      /* Return the date of the first Sunday in January 2007. */
   d=nwkdom(1, 1, 1, 2007);
      /* Return the date of the second Tuesday in September 2007. */
   e=nwkdom(2, 3, 9, 2007);
      /* Return the date of the fifth Thursday in December 2007. */
   f=nwkdom(5, 5, 12, 2007);
   put a= weekdatx.;
   put b= weekdatx.;
   put c= weekdatx.;
   put d= weekdatx.;
   put e= weekdatx.;
   put f= weekdatx.;
run;
```

Log 2.15 *Output from Returning Date Values*

```
a=Monday, 15 May 2000
b=Wednesday, 28 November 2007
c=Saturday, 24 November 2007
d=Sunday, 7 January 2007
e=Tuesday, 11 September 2007
f=Thursday, 27 December 2007
```

Example 2: Returning the Date of the Last Monday in May

The following example returns the date that corresponds to the last Monday in the month of May in the year 2007.

```
data _null_;
      /* The last Monday in May. */
   x=nwkdom(5,2,5,2007);
   put x date9.;
run;
```

The following output is written to the SAS log:

Log 2.16 *Output from Calculating the Date of the Last Monday in May*

```
28MAY2007
```

See Also

Functions:

- "HOLIDAY Function" on page 531
- "INTNX Function" on page 580
- "MONTH Function" on page 669
- "WEEKDAY Function" on page 981

OPEN Function

Opens a SAS data set.

Category: SAS File I/O

Syntax

OPEN(<*data-set-name* <,*mode* <,*generation-number* <,*type*> > > >)

Optional Arguments

data-set-name
is a character constant, variable, or expression that specifies the name of the SAS data set or SAS SQL view to be opened. The value of this character string should be of the form

<*libref*.> *member-name*<(*data-set-options*)>

Default The default value for *data-set-name* is _LAST_.

Restriction If you specify the FIRSTOBS= and OBS= data set options, they are ignored. All other data set options are valid.

mode
is a character constant, variable, or expression that specifies the type of access to the data set:

I opens the data set in INPUT mode (default). Values can be read but not modified. **'I'** uses the strongest access mode available in the engine. That is, if the engine supports random access, OPEN defaults to random access. Otherwise, the file is opened in **'IN'** mode automatically. Files are opened with sequential access and a system level warning is set.

IN opens the data set in INPUT mode. Observations are read sequentially, and you are allowed to revisit an observation.

IS opens the data set in INPUT mode. Observations are read sequentially, but you are not allowed to revisit an observation.

Default I

generation-number
specifies a consistently increasing number that identifies one of the historical versions in a generation group.

Tip The *generation-number* argument is ignored if *type* = F.

type
is a character constant and can be one of the following values:

D
specifies that the first argument, *data-set-name*, is a one-level or two-level data set name. The following example shows how the D *type* value can be used:

```
rc = open('lib.mydata', , , 'D');
```

Tip D is the default if there is no fourth argument.

F
specifies that the first argument, *data-set-name*, is a filename, a physical path to a file.The following examples show how the F *type* value can be used:

```
rc = open('c:\data\mydata.sas7bdat', , , 'F');
rc = open('c:\data\mydata', , , 'F');
```

Tip If you use the F value, then the third argument, *generation-number*, is ignored.

Note If an argument is invalid, OPEN returns 0. You can obtain the text of the corresponding error message from the SYSMSG function. Invalid arguments do not produce a message in the SAS log and do not set the _ERROR_ automatic variable.

Details

The OPEN function opens a SAS data set, DATA step, or a SAS SQL view and returns a unique numeric data set identifier, which is used in most other data set access functions. OPEN returns 0 if the data set could not be opened.

If you call the OPEN function from a macro, then the result of the call is valid only when the result is passed to functions in a macro. If you call the OPEN function from the DATA step, then the result is valid only when the result is passed to functions in the same DATA step.

By default, a SAS data set is opened with a control level of RECORD. For more information, see “CNTLLEV= Data Set Option” in *SAS Data Set Options: Reference* .An open SAS data set should be closed when it is no longer needed. If you open a data set within a DATA step, it will be closed automatically when the DATA step ends.

OPEN defaults to the strongest access mode available in the engine. That is, if the engine supports random access, OPEN defaults to random access. Otherwise, data sets are opened with sequential access, and a system-level warning is set.

Example

- This example opens the data set PRICES in the library MASTER using INPUT mode. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%let dsid=%sysfunc(open(master.prices,i));
%if (&dsid = 0) %then
     %put %sysfunc(sysmsg());
```

```
%else
     %put PRICES data set has been opened;
```

- This example passes values from macro or DATA step variables to be used on data set options. It opens the data set SASUSER.HOUSES, and uses the WHERE= data set option to apply a permanent WHERE clause. Note that in a macro statement you do not enclose character strings in quotation marks.

```
%let choice = style="RANCH";
%let dsid=%sysfunc(open(sasuser.houses
                       (where=(&choice)),i));
```

- This example shows how to check the returned value for errors and to write an error message from the SYSMSG function.

```
data _null_;
   d=open('bad','?');
   if not d then do;
      m=sysmsg();
      put m;
      abort;
   end;
   ... more SAS statements ...;
run;
```

See Also

Functions:

- “CLOSE Function” on page 303
- “SYSMSG Function” on page 905

ORDINAL Function

Returns the k*th* smallest of the missing and nonmissing values.

Category: Descriptive Statistics

Syntax

ORDINAL(*k*,*argument-1*,*argument-2*<,…*argument-n*>)

Required Arguments

k
is a numeric constant, variable, or expression with an integer value that is less than or equal to the number of subsequent elements in the list of arguments.

argument
specifies a numeric constant, variable, or expression. At least two arguments are required. An argument can consist of a variable list, preceded by OF.

Details

The ORDINAL function returns the k*th* smallest value, either missing or nonmissing, among the second through the last arguments.

Comparisons

The ORDINAL function counts both missing and nonmissing values, whereas the SMALLEST function counts only nonmissing values.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x1=ordinal(4,1,2,3,-4,5,6,7);`	`3`

PATHNAME Function

Returns the physical name of an external file or a SAS library, or returns a blank.

Categories: SAS File I/O
External Files

See: "PATHNAME Function: UNIX" in *SAS Companion for UNIX Environments*
"PATHNAME Function: z/OS" in *SAS Companion for z/OS*

Syntax

PATHNAME((*fileref* | *libref*) <,*search-ref*>)

Required Arguments

fileref
is a character constant, variable, or expression that specifies the fileref that is assigned to an external file.

libref
is a character constant, variable, or expression that specifies the libref that is assigned to a SAS library.

Optional Argument

search-ref
is a character constant, variable, or expression that specifies whether to search for a fileref or a libref.

F specifies a search for a fileref.

L specifies a search for a libref.

Details

PATHNAME returns the physical name of an external file or SAS library, or blank if *fileref* or *libref* is invalid.

If the name of a fileref is identical to the name of a libref, you can use the *search-ref* argument to choose which reference you want to search. If you specify a value of F, SAS searches for a fileref. If you specify a value of L, SAS searches for a libref.

If you do not specify a *search-ref* argument, and the name of a fileref is identical to the name of a libref, PATHNAME searches first for a libref. If a libref does not exist, PATHNAME then searches for a fileref.

The default length of the target variable in the DATA step is 200 characters.

You can assign a fileref to an external file by using the FILENAME statement or the FILENAME function.

You can assign a libref to a SAS library using the LIBNAME statement or the LIBNAME function. Some operating environments allow you to assign a libref using system commands.

Windows Specifics

Under some operating environments, filerefs can also be assigned by using system commands. For details, see the SAS documentation for your operating environment.

Example

This example uses the FILEREF function to verify that the fileref MYFILE is associated with an external file. Then it uses PATHNAME to retrieve the actual name of the external file:

```
data _null_;
   length fname $ 100;
   rc=fileref('myfile');
   if (rc=0) then
   do;
      fname=pathname('myfile');
      put fname=;
   end;
run;
```

See Also

Functions:

- “FEXIST Function” on page 408
- “FILEEXIST Function” on page 410
- “FILENAME Function” on page 411
- “FILEREF Function” on page 414

Statements:

- “LIBNAME Statement” in *SAS Statements: Reference*
- “FILENAME Statement” in *SAS Statements: Reference*

PCTL Function

Returns the percentile that corresponds to the percentage.

Category: Descriptive Statistics

Syntax

PCTL<*n*> (*percentage*, *value1*<,*value2*,…>)

Required Arguments

percentage

is a numeric constant, variable, or expression that specifies the percentile to be computed.

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

value

is a numeric variable, constant, or expression.

Optional Argument

n

is a digit from 1 to 5 which specifies the definition of the percentile to be computed.

Default definition 5

Details

The PCTL function returns the percentile of the nonmissing values corresponding to the percentage. If *percentage* is missing, less than zero, or greater than 100, the PCTL function generates an error message.

Note: The formula that is used in the PCTL function is the same formula that used in PROC UNIVARIATE. For more information, see "SAS Elementary Statistics Procedures" in Chapter 1 of *Base SAS Procedures Guide*.

Example

The following SAS statements produce these results.

SAS Statement	Result
`lower_quartile=PCTL(25,2,4,1,3);` `put lower_quartile;`	`1.5`
`percentile_def2=PCTL2(25,2,4,1,3);` `put percentile_def2;`	`1`
`lower_tertile=PCTL(100/3,2,4,1,3);` `put lower_tertile;`	`2`
`percentile_def3=PCTL3(100/3,2,4,1,3);` `put percentile_def3;`	`2`

SAS Statement	Result
`median=PCTL(50,2,4,1,3);` `put median;`	`2.5`
`upper_tertile=PCTL(200/3,2,4,1,3);` `put upper_tertile;`	`3`
`upper_quartile=PCTL(75,2,4,1,3);` `put upper_quartile;`	`3.5`

PDF Function

Returns a value from a probability density (mass) distribution.

Category: Probability

Alias: PMF

Syntax

PDF (*dist*,*quantile*<,*parm-1, ... ,parm-k*>)

Required Arguments

dist

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	**`Bernoulli`**
Beta	**`BETA`**
Binomial	**`BINOMIAL`**
Cauchy	**`CAUCHY`**
Chi-Square	**`CHISQUARE`**
Exponential	**`EXPONENTIAL`**
F	**`F`**
Gamma	**`GAMMA`**
Generalized Poisson	**`GENPOISSON`**
Geometric	**`GEOMETRIC`**

Distribution	Argument
Hypergeometric	**HYPERGEOMETRIC**
Laplace	**LAPLACE**
Logistic	**LOGISTIC**
Lognormal	**LOGNORMAL**
Negative binomial	**NEGBINOMIAL**
Normal	**NORMAL** | **GAUSS**
Normal mixture	**NORMALMIX**
Pareto	**PARETO**
Poisson	**POISSON**
T	**T**
Tweedie	**TWEEDIE**
Uniform	**UNIFORM**
Wald (inverse Gaussian)	**WALD** | **IGAUSS**
Weibull	**WEIBULL**

Note Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable, or expression that specifies the value of the random variable.

Optional Argument

parm-1,...,parm-k
are optional numeric constants, variables, or expressions that specify the values of *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

See "Details" on page 723 for complete information about these parameters.

Details

Bernoulli Distribution

PDF('BERNOULLI',*x*,*p*)

Arguments

x
is a numeric random variable.

p
is a numeric probability of success.

Range $0 \le p \le 1$

Details

The PDF function for the Bernoulli distribution returns the probability density function of a Bernoulli distribution, with probability of success equal to *p*. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('BERN', x, p) = \begin{cases} 0 & x < 0 \\ 1 - p & x = 0 \\ 0 & 0 < x < 1 \\ p & x = 1 \\ 0 & x > 1 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Beta Distribution

PDF('BETA',*x*,*a*,*b*<,*l*,*r*>)

Arguments

x
is a numeric random variable.

a
is a numeric shape parameter.

Range $a > 0$

b
is a numeric shape parameter.

Range $b > 0$

l
is the numeric left location parameter.

Default 0

r
is the right location parameter.

Default 1

Range $r > l$

Details

The PDF function for the beta distribution returns the probability density function of a beta distribution, with shape parameters *a* and *b*. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('BETA', x, a, b, l, r) = \begin{cases} 0 & x < l \\ \frac{1}{\beta(a, b)} \frac{(x - l)^{a-1}(r - x)^{b-1}}{(r - l)^{a+b-1}} & l \le x \le r \\ 0 & x > r \end{cases}$$

Note: The quantity $\frac{x - l}{r - l}$ is forced to be $\varepsilon \le \frac{x - l}{r - l} \le 1 - 2\varepsilon$.

Binomial Distribution

PDF('BINOMIAL',*m*,*p*,*n*)

Arguments

m

is an integer random variable that counts the number of successes.

Range $m = 0, 1, ...$

p

is a numeric probability of success.

Range $0 \le p \le 1$

n

is an integer parameter that counts the number of independent Bernoulli trials.

Range $n = 0, 1, ...$

Details

The PDF function for the binomial distribution returns the probability density function of a binomial distribution, with parameters *p* and *n*, which is evaluated at the value *m*. The equation follows:

$$PDF('BINOM', m, p, n) = \begin{cases} 0 & m < 0 \\ \binom{n}{m} p^m (1 - p)^{n-m} & 0 \le m \le n \\ 0 & m > n \end{cases}$$

Note: There are no *location* or *scale* parameters for the binomial distribution.

Cauchy Distribution

PDF('CAUCHY',*x*<,*θ*,*λ*>)

Arguments

x

is a numeric random variable.

θ

is a numeric location parameter.

Default 0

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the Cauchy distribution returns the probability density function of a Cauchy distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('CAUCHY', x, \theta, \lambda) = \frac{1}{\pi}\left(\frac{\lambda}{\lambda^2 + (x - \theta)^2}\right)$$

Chi-Square Distribution

PDF('CHISQUARE',*x*,*df* <,*nc*>)

Arguments

x
is a numeric random variable.

df
is a numeric degrees of freedom parameter.

Range $df > 0$

nc
is an optional numeric non-centrality parameter.

Range $nc \geq 0$

Details

The PDF function for the chi-square distribution returns the probability density function of a chi-square distribution, with *df* degrees of freedom and non-centrality parameter *nc*. The PDF function is evaluated at the value *x*. This function accepts non-integer degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central chi-square distribution. The following equation describes the PDF function for the chi-square distribution:

$$PDF('CHISQ', x, v, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} p_c(x, v + 2j) & x \geq 0 \end{cases}$$

In the equation, $p_c(.,.)$ denotes the density from the central chi-square distribution:

$$p_c(x, a) = \frac{1}{2} p_g\left(\frac{x}{2}, \frac{a}{2}\right)$$

In the equation, $p_g(y,b)$ is the density from the gamma distribution, which is given by the following equation:

$$p_g(y, b) = \frac{1}{\Gamma(b)} e^{-y} y^{b-1}$$

Exponential Distribution

PDF('EXPONENTIAL',*x* <,*λ*>)

Arguments

x
is a numeric random variable.

λ
is a scale parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the exponential distribution returns the probability density function of an exponential distribution, with the scale parameter λ. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('EXPO', x, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

F Distribution

PDF('F',*x*,*ndf*,*ddf*<,*nc*>)

Arguments

x
is a numeric random variable.

ndf
is a numeric numerator degrees of freedom parameter.

Range $ndf > 0$

ddf
is a numeric denominator degrees of freedom parameter.

Range $ddf > 0$

nc
is a numeric non-centrality parameter.

Range $nc \geq 0$

Details

The PDF function for the *F* distribution returns the probability density function of an *F* distribution, with *ndf* numerator degrees of freedom, *ddf* denominator degrees of freedom, and the non-centrality parameter *nc*. The PDF function is evaluated at the value *x*. This PDF function accepts non-integer degrees of freedom for *ndf* and *ddf*. If *nc* is omitted or equal to zero, the value returned is from a central *F* distribution. In the following equation, let $v_1 = ndf$, let $v_2 = ddf$, and let $\lambda = nc$. The equation describes the PDF function for the *F* distribution:

$$PDF('F', x, v_1, v_2, \lambda) = \begin{cases} 0 & x < 0 \\ \sum_{j=0}^{\infty} e^{-\frac{\lambda}{2}} \frac{\left(\frac{\lambda}{2}\right)^j}{j!} p_f(f, v_1 + 2j, v_2) & x \geq 0 \end{cases}$$

In the equation, $p_f(f,u_1,u_2)$ is the density from the central *F* distribution:

$$p_f(f, u_1, u_2) = p_B\left(\frac{u_1 f}{uf + u_2}, \frac{u_1}{2}, \frac{u_2}{2}\right)\frac{u_1 u_2}{(u_2 + u_1 f)^2}$$

In the equation $p_B(x,a,b)$ is the density from the standard beta distribution.

Note: There are no *location* or *scale* parameters for the *F* distribution.

Gamma Distribution

PDF('GAMMA',*x*,*a*<,*λ*>)

Arguments

x

is a numeric random variable.

a

is a numeric shape parameter.

Range $a > 0$

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the gamma distribution returns the probability density function of a gamma distribution, with the shape parameter *a* and the scale parameter λ. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('GAMMA', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ \frac{1}{\lambda^a \Gamma(a)} x^{a-1} exp\left(-\frac{x}{\lambda}\right) & x \geq 0 \end{cases}$$

Generalized Poisson Distribution

PDF('GENPOISSON',*x*, *θ*, *η*)

Arguments

x

is an integer random variable.

θ

specifies a shape parameter.

Range $<10^5$ and >0

η

specifies a shape parameter.

Range ≥ 0 and <0.95

Tip When η =0, the distribution is the Poisson distribution with a mean and variance of θ. When η>0, the mean is $\theta \div (1 - \eta)$ and the variance is $\theta \div (1 - \eta)^3$.

Details

The probability mass function for the generalized Poisson distribution follows:

$$f(x;\ \theta,\ \eta) = \theta(\theta + \eta x)^{x-1} e^{-\theta-\eta x} / x!\ ,\quad x = 0,\ 1,\ 2,\ ...,\qquad \theta > 0,\ 0 \le \eta < 1$$

Geometric Distribution

PDF('GEOMETRIC',*m*,*p*)

Arguments

m

is a numeric random variable that denotes the number of failures before the first success.

Range $m \ge 0$

p

is a numeric probability of success.

Range $0 \le p \le 1$

Details

The PDF function for the geometric distribution returns the probability density function of a geometric distribution, with parameter *p*. The PDF function is evaluated at the value *m*. The equation follows:

$$PDF('GEOM',\ m,\ p) = \begin{cases} 0 & m < 0 \\ p(1 - p)^m & m \ge 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for this distribution.

Hypergeometric Distribution

PDF('HYPER',*x*,*N*,*R*,*n*<,*o*>)

Arguments

x

is an integer random variable.

N

is an integer population size parameter.

Range $N = 1, 2, ...$

R

is an integer number of items in the category of interest.

Range $R = 0, 1, ..., N$

n

is an integer sample size parameter.

Range $n = 1, 2, ..., N$

o

is an optional numeric odds ratio parameter.

Range $o > 0$

Details

The PDF function for the hypergeometric distribution returns the probability density function of an extended hypergeometric distribution, with population size *N*, number of items *R*, sample size *n*, and odds ratio *o*. The PDF function is evaluated at the value *x*. If *o* is omitted or equal to 1, the value returned is from the usual hypergeometric distribution. The equation follows:

$$PDF\left('HYPER', x, N, R, n, o\right) = \begin{cases} 0 & x < max(0,\ R + n - N) \\ \dfrac{\binom{R}{x}\binom{N-R}{n-x}o^{x}}{\sum_{j=max(0,R+n-N)}^{min(R,n)}\binom{R}{j}\binom{N-R}{n-j}o^{j}} & max(0,\ R + n - N) \le x \le min(R,\ n) \\ 0 & x > min(R,\ n) \end{cases}$$

Laplace Distribution

PDF('LAPLACE',*x*<,*θ*,*λ*>)

Arguments

x

is a numeric random variable.

θ

is a numeric location parameter.

Default 0

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the Laplace distribution returns the probability density function of the Laplace distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF\left('LAPLACE', x, \theta, \lambda\right) = \frac{1}{2\lambda} exp\left(-\frac{|\ x - \theta\ |}{\lambda}\right)$$

Logistic Distribution

PDF('LOGISTIC',*x*<,*θ*,*λ*>)

Arguments

x

is a numeric random variable.

θ

is a numeric location parameter.

Default 0

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the logistic distribution returns the probability density function of a logistic distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('LOGISTIC', x, \theta, \lambda) = \frac{exp\left(-\frac{x-\theta}{\lambda}\right)}{\lambda\left(1+exp\left(-\frac{x-\theta}{\lambda}\right)\right)^2}$$

Lognormal Distribution

PDF('LOGNORMAL',*x*<,*θ*,*λ*>)

Arguments

x

is a numeric random variable.

θ

specifies a numeric log scale parameter. (exp(θ) is a scale parameter.)

Default 0

λ

specifies a numeric shape parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the lognormal distribution returns the probability density function of a lognormal distribution, with the log scale parameter θ and the shape parameter λ. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('LOGN', x, \theta, \lambda) = \begin{cases} 0 & x \le 0 \\ \frac{1}{x\lambda\sqrt{2\pi}}\ exp\left(-\frac{(log(x)-\theta)^2}{2\lambda^2}\right) & x > 0 \end{cases}$$

Negative Binomial Distribution

PDF('NEGBINOMIAL',*m*,*p*,*n*)

Arguments

m

is a positive integer random variable that counts the number of failures.

Range $m = 0, 1, \ldots$

p

is a numeric probability of success.

Range $0 \le p \le 1$

n
is a numeric value that counts the number of successes.

Range $n>0$

Details

The PDF function for the negative binomial distribution returns the probability density function of a negative binomial distribution, with probability of success p and number of successes n. The PDF function is evaluated at the value m. The equation follows:

$$PDF('NEGB', m, p, n) = \begin{cases} 0 & m < 0 \\ \binom{n+m-1}{n-1} p^n (1-p)^m & m \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the negative binomial distribution.

Normal Distribution

PDF('NORMAL',*x*<,*θ*,*λ*>)

Arguments

x
is a numeric random variable.

θ
is a numeric location parameter.

Default 0

λ
is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the normal distribution returns the probability density function of a normal distribution, with the location parameter θ and the scale parameter λ. The PDF function is evaluated at the value x. The equation follows:

$$PDF('NORMAL', x, \theta, \lambda) = \frac{1}{\lambda\sqrt{2\pi}} \exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

Normal Mixture Distribution

PDF('NORMALMIX',*x*,*n*,*p*,*m*,*s*)

Arguments

x
is a numeric random variable.

n
is the integer number of mixtures.

Range $n = 1, 2, \ldots$

p

is the n proportions, $p_1, p_2, \ldots, p_n$, where $\sum_{i=1}^{i=n} p_i = 1$.

Range $p = 0, 1, \ldots$

m

is the *n* means $m_1, m_2, \ldots, m_n$.

s

is the *n* standard deviations $s_1, s_2, \ldots, s_n$.

Range $s > 0$

Details

The PDF function for the normal mixture distribution returns the probability that an observation from a mixture of normal distribution is less than or equal to *x*. The equation follows:

$$PDF('NORMALMIX', x, n, p, m, s) = \sum_{i=1}^{i=n} p_i \, PDF('NORMAL', x, m_i, s_i)$$

Note: There are no *location* or *scale* parameters for the normal mixture distribution.

Pareto Distribution

PDF('PARETO',*x*,*a*<,*k*>)

Arguments

x

is a numeric random variable.

a

is a numeric shape parameter.

Range $a > 0$

k

is a numeric scale parameter.

Default 1

Range $k > 0$

Details

The PDF function for the Pareto distribution returns the probability density function of a Pareto distribution, with the shape parameter *a* and the scale parameter *k*. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('PARETO', x, a, k) = \begin{cases} 0 & x < k \\ \frac{a}{k}\left(\frac{k}{x}\right)^{a+1} & x \geq k \end{cases}$$

Poisson Distribution

PDF('POISSON',*n*,*m*)

Arguments

n
is an integer random variable.

Range $n = 0, 1, \ldots$

m
is a numeric mean parameter.

Range $m > 0$

Details

The PDF function for the Poisson distribution returns the probability density function of a Poisson distribution, with mean *m*. The PDF function is evaluated at the value *n*. The equation follows:

$$PDF('POISSON', n, m) = \begin{cases} 0 & n < 0 \\ e^{-m}\frac{m^n}{n!} & n \geq 0 \end{cases}$$

Note: There are no *location* or *scale* parameters for the Poisson distribution.

T Distribution

PDF('T',*t*,*df*<,*nc*>)

Arguments

t
is a numeric random variable.

df
is a numeric degrees of freedom parameter.

Range $df > 0$

nc
is an optional numeric non-centrality parameter.

Details

The PDF function for the *T* distribution returns the probability density function of a *T* distribution, with degrees of freedom *df* and the non-centrality parameter *nc*. The PDF function is evaluated at the value *x*. This PDF function accepts non-integer degrees of freedom. If *nc* is omitted or equal to zero, the value returned is from the central *T* distribution. In the following equation, let $\nu = df$ and let $\delta = nc$.

$$PDF('T', t, \nu, \delta) = \frac{1}{2^{\frac{\nu}{2}-1}\Gamma\left(\frac{\nu}{2}\right)} \int_0^{\infty} x^{\nu-1} e^{-\frac{1}{2}x^2} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{tx}{\sqrt{\nu}}-\delta\right)^2} \frac{x}{\sqrt{\nu}} dx$$

Note: There are no *location* or *scale* parameters for the *T* distribution.

Tweedie Distribution

PDF('TWEEDIE', *y*, *p*<,*μ*, *φ*>)

Arguments

y
is a random variable.

Range $y \geq 0$

Notes This argument is required.

When $y>1$, y is numeric. When $p=1$, y is an integer.

p

is the power parameter.

Range $p \geq 1$

Note This argument is required.

μ

is the mean.

Default 1

Range $\mu > 0$

φ

is the dispersion parameter.

Default 1

Range $\varphi > 0$

Details

The PDF function for the Tweedie distribution returns an exponential dispersion model with variance and mean related by the equation *variance* = $\phi * \mu^{p}$.

The equation follows:

$$\frac{1}{y}\sum_{j=1}^{\infty}\left(\frac{y^{-j\alpha}(p-1)^{j\alpha}}{\phi^{j(1-\alpha)}(2-p)^{j}\,j!\,\Gamma(-j\alpha)}\right)\exp\left(\frac{1}{\phi}\left(y\frac{\mu^{1-p}-1}{1-p}-\frac{\mu^{2-p}-1}{2-p}\right)\right)$$

The following relationship applies to the preceding equation:

$$\alpha = \frac{2-p}{1-p}$$

Note: The accuracy of computed Tweedie probabilities is highly dependent on the location in parameter space. Ten digits of accuracy are usually available except when p is near 2 or ϕ is near 0, in which case the accuracy might be as low as six digits.

Uniform Distribution

PDF('UNIFORM',*x*<,*l*,*r*>)

Arguments

x

is a numeric random variable.

l

is the numeric left location parameter.

Default 0

r
is the numeric right location parameter.

Default 1

Range $r > l$

Details

The PDF function for the uniform distribution returns the probability density function of a uniform distribution, with the left location parameter *l* and the right location parameter *r*. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('UNIFORM', x, l, r) = \begin{cases} 0 & x < l \\ \frac{1}{r - l} & l \le x \le r \\ 0 & x > r \end{cases}$$

Wald (Inverse Gaussian) Distribution

PDF('WALD',*x*,*λ*<,*μ*>)

PDF('IGAUSS',*x*,*λ*<,*μ*>)

Arguments

x
is a numeric random variable.

λ
is a numeric shape parameter.

Range $\lambda > 0$

μ
is the mean.

Default 1

Range $\mu > 0$

Details

The PDF function for the Wald distribution returns the probability density function of a Wald distribution, with shape parameter *λ*, which is evaluated at the value *x*. The equation follows:

$$fx(x) = \left[\frac{\lambda}{2\pi x^3}\right]^{1/2} \exp\left\{-\frac{\lambda}{2\mu^2 x}(x-\mu)^2\right\}, \qquad x > 0$$

Weibull Distribution

PDF('WEIBULL',*x*,*a*<,*λ*>)

Arguments

x
is a numeric random variable.

a
is a numeric shape parameter.

Range $a > 0$

λ

is a numeric scale parameter.

Default 1

Range $\lambda > 0$

Details

The PDF function for the Weibull distribution returns the probability density function of a Weibull distribution, with the shape parameter *a* and the scale parameter λ. The PDF function is evaluated at the value *x*. The equation follows:

$$PDF('WEIBULL', x, a, \lambda) = \begin{cases} 0 & x < 0 \\ exp\left(-\left(\frac{x}{\lambda}\right)^a\right)\frac{a}{\lambda}\left(\frac{x}{\lambda}\right)^{a-1} & x \geq 0 \end{cases}$$

Example

The following SAS statements produce these results.

SAS Statement	Result
`y=pdf('BERN',0,.25);`	`0.75`
`y=pdf('BERN',1,.25);`	`0.25`
`y=pdf('BETA',0.2,3,4);`	`1.2288`
`y=pdf('BINOM',4,.5,10);`	`0.20508`
`y=pdf('CAUCHY',2);`	`0.063662`
`y=pdf('CHISQ',11.264,11);`	`0.081686`
`y=pdf('EXPO',1);`	`0.36788`
`y=pdf('F',3.32,2,3);`	`0.054027`
`y=pdf('GAMMA',1,3);`	`0.18394`
`y=pdf('GENPOISSON',9,1,.7);`	`0.0150130915`
`y=pdf('HYPER',2,200,50,10);`	`0.28685`
`y=pdf('LAPLACE',1);`	`0.18394`
`y=pdf('LOGISTIC',1);`	`0.19661`
`y=pdf('LOGNORMAL',1);`	`0.39894`
`y=pdf('NEGB',1,.5,2);`	`0.25`
`y=pdf('NORMAL',1.96);`	`0.058441`

SAS Statement	Result
y=pdf('NORMALMIX',2.3,3,.33,.33,.34, .5,1.5,2.5,.79,1.6,4.3);	0.1166
y=pdf('PARETO',1,1);	1
y=pdf('POISSON',2,1);	0.18394
y=pdf('T',.9,5);	0.24194
y=pdf('TWEEDIE',.8,5);	0.7422908236
y=pdf('UNIFORM',0.25);	1
y=pdf('WALD',1,2);	0.56419
y=pdf('WEIBULL',1,2);	0.73576

See Also

Functions:

- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “CDF Function” on page 277
- “SDF Function” on page 856
- “QUANTILE Function” on page 799
- “SQUANTILE Function” on page 881

PEEK Function

Stores the contents of a memory address in a numeric variable on a 32-bit platform.

Category: Special

Restriction: Use on 32-bit platforms only.

Syntax

PEEK(*address*<,*length*>)

Required Argument

address
is a numeric constant, variable, or expression that specifies the memory address.

Optional Argument

length
is a numeric constant, variable, or expression that specifies the data length.

Default	a 4-byte address pointer
Range	2–8

Details

If you do not have access to the memory storage location that you are requesting, the PEEK function returns an "Invalid argument" error.

You cannot use the PEEK function on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use PEEK, change the applications and use PEEKLONG instead. You can use PEEKLONG on both 32–bit and 64–bit platforms.

Comparisons

The PEEK function stores the contents of a memory address into a *numeric* variable. The PEEKC function stores the contents of a memory address into a *character* variable.

Note: SAS recommends that you use PEEKLONG instead of PEEK because PEEKLONG can be used on both 32–bit and 64–bit platforms.

Example

The following example, specific to the z/OS operating environment, returns a numeric value that represents the address of the Communication Vector Table (CVT).

```
data _null_;
      /* 16 is the location of the CVT address */
   y=16;
   x=peek(y);
   put 'x= ' x hex8.;
run;
```

See Also

Functions:

- “ADDR Function” on page 92
- “PEEKC Function” on page 739

CALL Routines:

- “CALL POKE Routine” on page 195

PEEKC Function

Stores the contents of a memory address in a character variable on a 32-bit platform.

Category: Special

Restriction: Use on 32-bit platforms only.

Syntax

PEEKC(*address*<,*length*>)

Required Argument

address
is a numeric constant, variable, or expression that specifies the memory address.

Optional Argument

length
is a numeric constant, variable, or expression that specifies the data length.

Default 8, unless the variable length has already been set (by the LENGTH statement, for example)

Range 1–32,767

Details

If you do not have access to the memory storage location that you are requesting, the PEEKC function returns an "Invalid argument" error.

You cannot use the PEEKC function on 64-bit platforms. If you attempt to use it, SAS writes a message to the log stating that this restriction applies. If you have legacy applications that use PEEKC, change the applications and use PEEKCLONG instead. You can use PEEKCLONG on both 32–bit and 64–bit platforms.

Comparisons

The PEEKC function stores the contents of a memory address into a *character* variable. The PEEK function stores the contents of a memory address into a *numeric* variable.

Note: SAS recommends that you use PEEKCLONG instead of PEEKC because PEEKCLONG can be used on both 32–bit and 64–bit platforms.

Example: Listing ASCB Bytes

The following example, specific to the z/OS operating environment, uses both PEEK and PEEKC, and prints the first four bytes of the Address Space Control Block (ASCB).

```
data _null_;
   length y $4;
      /* 220x is the location of the ASCB pointer */
   x=220x;
   y=peekc(peek(x));
   put 'y= ' y;
run;
```

See Also

Functions:

- "ADDR Function" on page 92
- "PEEK Function" on page 738

CALL Routines:

- "CALL POKE Routine" on page 195

PEEKCLONG Function

Stores the contents of a memory address in a character variable on 32-bit and 64-bit platforms.

Category: Special

See: "PEEKCLONG Function: z/OS" in *SAS Companion for z/OS*

Syntax

PEEKCLONG(*address*<,*length*>)

Required Argument

address
specifies a character constant, variable, or expression that contains the binary pointer address.

Optional Argument

length
is a numeric constant, variable, or expression that specifies the length of the character data.

Default 8

Range 1 to 32,767

Details

If you do not have access to the memory storage location that you are requesting, the PEEKCLONG function returns an "Invalid argument" error.

Comparisons

The PEEKCLONG function stores the contents of a memory address in a *character* variable.

The PEEKLONG function stores the contents of a memory address in a *numeric* variable. It assumes that the input address refers to an integer in memory.

Examples

Example 1: Example for a 32-bit Platform

The following example returns the pointer address for the character variable Z.

```
data _null_;
   x='ABCDE';
   y=addrlong(x);
   z=peekclong(y,2);
   put z=;
run;
```

The output from the SAS log is: **z=AB**

Example 2: Example for a 64-bit Platform

The following example, specific to the z/OS operating environment, returns the pointer address for the character variable Y.

```
data _null_;
   length y $4;
   x220addr=put(220x,pib4.);
   ascb=peeklong(x220addr);
   ascbaddr=put(ascb,pib4.);
   y=peekclong(ascbaddr);
run;
```

The output from the SAS log is: **y='ASCB'**

See Also

Functions:

- "PEEKLONG Function" on page 742

PEEKLONG Function

Stores the contents of a memory address in a numeric variable on 32-bit and 64-bit platforms.

Category: Special

See: "PEEKLONG Function: Windows" in *SAS Companion for Windows*
"PEEKLONG Function: UNIX" in *SAS Companion for UNIX Environments*
"PEEKLONG Function: z/OS" in *SAS Companion for z/OS*

Syntax

PEEKLONG(*address*<,*length*>)

Required Argument

address
specifies a character constant, variable, or expression that contains the binary pointer address.

Optional Argument

length
is a numeric constant, variable, or expression that specifies the length of the character data.

Default	4 on 32-bit computers; 8 on 64-bit computers.
Range	1-4 on 32-bit computers; 1-8 on 64-bit computers.

Details

If you do not have access to the memory storage location that you are requesting, the PEEKLONG function returns an “Invalid argument” error.

Comparisons

The PEEKLONG function stores the contents of a memory address in a *numeric* variable. It assumes that the input address refers to an integer in memory.

The PEEKCLONG function stores the contents of a memory address in a *character* variable. It assumes that the input address refers to character data.

Examples

Example 1: Example for a 32-bit Platform

The following example returns the pointer address for the numeric variable Z.

```
data _null_;
   length y $4;
   y=put(1,IB4.);
   addry=addrlong(y);
   z=peeklong(addry,4);
   put z=;
run;
```

SAS writes the following output to the log: **z=1**

Example 2: Example for a 64-bit Platform

The following example, specific to the z/OS operating environment, returns the pointer address for the numeric variable X.

```
   data _null_;
      x=peeklong(put(16,pib4.));
      put x=hex8.;
   run;
```

SAS writes the following output to the log: **x=00FCFCB0**

See Also

Functions:

- “PEEKCLONG Function” on page 741

PERM Function

Computes the number of permutations of *n* items that are taken *r* at a time.

Category: Combinatorial

Syntax

PERM(*n*<,*r*>)

Required Argument

n

is an integer that represents the total number of elements from which the sample is chosen.

Optional Argument

r

is an integer value that represents the number of chosen elements. If *r* is omitted, the function returns the factorial of *n*.

Restriction $r \leq n$

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 SAS statements produce these results.

SAS Statement	Result
`x=perm(5,1);`	`5`
`x=perm(5);`	`120`
`x=perm(5,2)`	`20`

See Also

Functions:

- “COMB Function” on page 310
- “FACT Function” on page 399
- “LPERM Function” on page 647

PMT Function

Returns the periodic payment for a constant payment loan or the periodic savings for a future balance.

Category: Financial

Syntax

PMT (*rate*, *number-of-periods*, *principal-amount*, <*future-amount*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

number-of-periods
specifies the number of payment periods. *number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

future-amount
specifies the future amount. *future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a missing value is specified.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Example

- 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 1037.03.

- 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 1030.16.

- 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, 5, -5000);
```

This computation returns a value of –1030.20.

- The payment for monthly periodic savings that accrue over 18 years at a 6% nominal annual interest rate, and which accumulates $50,000 at the end of the 18 years, is computed as follows:

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

This computation returns a value of 129.081.

POINT Function

Locates an observation that is identified by the NOTE function.

Category: SAS File I/O

Syntax

POINT(*data-set-id*,*note-id*)

Required Arguments

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

note-id
is a numeric variable that specifies the identifier assigned to the observation by the NOTE function.

Details

POINT returns 0 if the operation was successful, ≠0 if it was not successful. POINT prepares the program to read from the SAS data set. The Data Set Data Vector is not updated until a read is done using FETCH or FETCHOBS.

Example

This example calls NOTE to obtain an observation ID for the most recently read observation of the SAS data set MYDATA. It calls POINT to point to that observation, and calls FETCH to return the observation marked by the pointer.

```
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetch(&dsid));
%let noteid=%sysfunc(note(&dsid));
   ...more macro statements...
%let rc=%sysfunc(point(&dsid,&noteid));
%let rc=%sysfunc(fetch(&dsid));
   ...more macro statements...
%let rc=%sysfunc(close(&dsid));
```

See Also

Functions:

- “DROPNOTE Function” on page 387

- "NOTE Function" on page 692
- "OPEN Function" on page 716

POISSON Function

Returns the probability from a Poisson distribution.

Category: Probability

See: "CDF Function" on page 277 , "PDF Function" on page 722

Syntax

POISSON(*m*,*n*)

Required Arguments

m

is a numeric mean parameter.

Range $m \geq 0$

n

is an integer random variable.

Range $n \geq 0$

Details

The POISSON function returns the probability that an observation from a Poisson distribution, with mean *m*, is less than or equal to *n*. To compute the probability that an observation is equal to a given value, *n*, compute the difference of two probabilities from the Poisson distribution for *n* and *n*−1.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=poisson(1,2);`	`0.9196986029`

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644

- "PDF Function" on page 722
- "SDF Function" on page 856

PPMT Function

Returns the principal payment for a given period for a constant payment loan or the periodic savings for a future balance.

Category: Financial

Syntax

PPMT (*rate*, *period*, *number-of-periods*, *principal-amount*, <*future-amount*>, <*type*>)

Required Arguments

rate
specifies the interest rate per payment period.

period
specifies the payment period for which the principal payment is computed. *period* must be a positive integer value that is less than or equal to the value of *number-of-periods*.

number-of-periods
specifies the number of payment periods. *number-of-periods* must be a positive integer value.

principal-amount
specifies the principal amount of the loan. Zero is assumed if a missing value is specified.

Optional Arguments

future-amount
specifies the future amount. *future-amount* can be the outstanding balance of a loan after the specified number of payment periods, or the future balance of periodic savings. Zero is assumed if *future-amount* is omitted or if a missing value is specified.

type
specifies whether the payments occur at the beginning or end of a period. 0 represents the end-of-period payments, and 1 represents the beginning-of-period payments. 0 is assumed if *type* is omitted or if a missing value is specified.

Example

- 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 75.62.

- 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 640.10 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 389.914 as the principal that was paid with the first payment.

PROBBETA Function

Returns the probability from a beta distribution.

Category: Probability

See: “CDF Function” on page 277, “PDF Function” on page 722

Syntax

PROBBETA(*x*,*a*,*b*)

Required Arguments

x

is a numeric random variable.

Range $0 \leq x \leq 1$

a

is a numeric shape parameter.

Range $a > 0$

b

is a numeric shape parameter.

Range $b > 0$

Details

The PROBBETA function returns the probability that an observation from a beta distribution, with shape parameters *a* and *b*, is less than or equal to *x*.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=probbeta(.2,3,4);`	`0.09888`

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856

PROBBNML Function

Returns the probability from a binomial distribution.

Category: Probability

See: "CDF Function" on page 277 , "PDF Function" on page 722

Syntax

PROBBNML(*p*,*n*,*m*)

Required Arguments

p

is a numeric probability of success parameter.

Range $0 \le p \le 1$

n

is an integer number of independent Bernoulli trials parameter.

Range $n > 0$

m

is an integer number of successes random variable.

Range $0 \le m \le n$

Details

The PROBBNML function returns the probability that an observation from a binomial distribution, with probability of success p, 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 SAS statement produces this result.

SAS Statement	Result
`x=probbnml(0.5,10,4);`	`0.376953125`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “SDF Function” on page 856

PROBBNRM Function

Returns a probability from a bivariate normal distribution.

Category: Probability

Syntax

PROBBNRM(*x*,*y*,*r*)

Required Arguments

x

specifies a numeric constant, variable, or expression.

y

specifies a numeric constant, variable, or expression.

r

is a numeric correlation coefficient.

Range $-1 \le r \le 1$

Details

The PROBBNRM function returns the probability that an observation (X, Y) from a standardized bivariate normal distribution with mean 0, variance 1, and a correlation coefficient *r*, is less than or equal to (*x*, *y*). That is, it returns the probability that X≤*x* and Y≤*y*. The following equation describes the PROBBNRM function, where *u* and *v* represent the random variables *x* and *y*, respectively:

$$PROBBNRM(x,\ y,\ r) = \frac{1}{2\pi\sqrt{1 - r^2}} \int_{-\infty}^{x} \int_{-\infty}^{y} \exp\left[- \frac{u^2 - 2ruv + v^2}{2(1 - r^2)} \right] dv\, du$$

Example

The following SAS statements produce this result.

SAS Statement	Result
`p=probbnrm(.4, -.3, .2);` `put p;`	`0.2783183345`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “SDF Function” on page 856

PROBCHI Function

Returns the probability from a chi-square distribution.

Category: Probability

See: “CDF Function” on page 277, “PDF Function” on page 722

Syntax

PROBCHI(*x*,*df*<,*nc*>)

Required Arguments

x

is a numeric random variable.

Range $x \geq 0$

df

is a numeric degrees of freedom parameter.

Range $df > 0$

Optional Argument

nc

is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Details

The PROBCHI function returns the probability that an observation from a chi-square distribution, with degrees of freedom *df* and noncentrality parameter *nc*, is less than or equal to *x*. This function accepts a noninteger degrees of freedom parameter *df*. If the optional parameter *nc* is not specified or has the value 0, the value returned is from the central chi-square distribution.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=probchi(11.264,11);`	`0.5785813293`

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856

PROBF Function

Returns the probability from an *F* distribution.

Category: Probability

See: "CDF Function" on page 277, "PDF Function" on page 722

Syntax

PROBF(*x*,*ndf*,*ddf*<,*nc*>)

Required Arguments

x

is a numeric random variable.

Range $x \geq 0$

ndf

is a numeric numerator degrees of freedom parameter.

Range $ndf > 0$

ddf
is a numeric denominator degrees of freedom parameter.

Range $ddf > 0$

Optional Argument

nc
is an optional numeric noncentrality parameter.

Range $nc \geq 0$

Details

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

The significance level for an *F* test statistic is given by

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

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=probf(3.32,2,3);`	`0.8263933602`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “SDF Function” on page 856

PROBGAM Function

Returns the probability from a gamma distribution.

Category: Probability

See: “CDF Function” on page 277, “PDF Function” on page 722

Syntax

PROBGAM(*x*,*a*)

Required Arguments

x

is a numeric random variable.

Range $x \geq 0$

a

is a numeric shape parameter.

Range $a > 0$

Details

The PROBGAM function returns the probability that an observation from a gamma distribution, with shape parameter *a*, is less than or equal to *x*.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=probgam(1,3);`	`0.0803013971`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “SDF Function” on page 856

PROBHYPR Function

Returns the probability from a hypergeometric distribution.

Category: Probability

See: “CDF Function” on page 277, “PDF Function” on page 722

Syntax

PROBHYPR(*N*,*K*,*n*,*x*<,*r*>)

Required Arguments

N
is an integer population size parameter.

Range $N \geq 1$

K
is an integer number of items in the category of interest parameter.

Range $0 \leq K \leq N$

n
is an integer sample size parameter.

Range $0 \leq n \leq N$

x
is an integer random variable.

Range $max(0, K + n-N) \leq x \leq min(K,n)$

Optional Argument

r
is an optional numeric odds ratio parameter.

Range $r \geq 0$

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 SAS statement produces this result.

SAS Statement	Result
`x=probhypr(200,50,10,2);`	`0.5236734081`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640

- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856

PROBIT Function

Returns a quantile from the standard normal distribution.

Category: Quantile

Syntax

PROBIT(*p*)

Required Argument

p
is a numeric probability.

Range $0 < p < 1$

Details

The PROBIT function returns the p^{th} quantile from the standard normal distribution. The probability that an observation from the standard normal distribution is less than or equal to the returned quantile is *p*.

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

Note: PROBIT is the inverse of the PROBNORM function.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=probit(.025);`	`-1.959963985`
`x=probit(1.e-7);`	`-5.199337582`

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642

- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856

PROBMC Function

Returns a probability or a quantile from various distributions for multiple comparisons of means.

Category: Probability

Syntax

PROBMC(*distribution*,*q*,*prob*,*df*,*nparms*<, *parameters*>)

Required Arguments

distribution

is a character constant, variable, or expression that identifies the distribution. The following are valid distributions:

Distribution	Argument
Analysis of Means	ANOM
One-sided Dunnett	DUNNETT1
Two-sided Dunnett	DUNNETT2
Maximum Modulus	MAXMOD
Partitioned Range	PARTRANGE
Studentized Range	RANGE
Williams	WILLIAMS

q

is the quantile from the distribution.

Restriction Either *q* or *prob* can be specified, but not both.

prob

is the left probability from the distribution.

Restriction Either *prob* or *q* can be specified, but not both.

df

is the degrees of freedom.

Note: A missing value is interpreted as an infinite value.

nparms
is the number of treatments.

Note: For DUNNETT1 and DUNNETT2, the control group is not counted.

Optional Argument

parameters
is an optional set of *nparms* parameters that must be specified to handle 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.

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*.

$$d\mu_v(x) = \frac{v^{\frac{v}{2}}}{\Gamma\left(\frac{v}{2}\right)2^{\frac{v}{2}-1}} x^{v-1}\varepsilon^{-\frac{vx^2}{2}}\,dx$$

$$\phi(x) = \frac{1}{\sqrt{2\pi}}\varepsilon^{-\frac{x^2}{2}}$$

$$\Phi(x) = \int_{-\infty}^{x}\phi(u)\,du$$

Computing the Analysis of Means

Analysis of Means (ANOM) applies to data that is organized as k (Gaussian) samples, the i^{th} sample being of size n_i. Let $I = \sqrt{-1}$. The distribution function [1, 2, 3, 4, 5] is the CDF for the maximum absolute of a k-dimensional multivariate $\mathbb{T}$ vector, with v degrees of freedom, and an associated correlation matrix $\rho_{ij} = -\alpha_i\alpha_j$. This equation can be written as

$$prob = r(|t_1| < h, |t_2| < h, ..., |t_k| < h)$$
$$= \int_0^\infty \left\{ \int_0^\infty \prod_{j=0}^{j=k} g(sh, y, \alpha_j)\phi(y)dy \right\} d\mu_v(s)$$

The following relationship applies to the preceding equation:

$$g(sh, y, \alpha_j) = \Phi\left(\frac{sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-sh - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

where $\Gamma(.)$, $\phi(.)$, and $\Phi(.)$, are the gamma function, the density, and the CDF from the standard normal distribution, respectively.

For $v = \infty$, the distribution reduces to:

$$r(|t_1| < h, |t_2| < h, ..., |t_k| < h) = \int_0^\infty \prod_{j=0}^{j=k} g(h, y, \alpha_j)\phi(y)dy$$

The following relationship applies to the preceding equation:

$$g(h, y, \alpha_j) = \Phi\left(\frac{h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right) - \Phi\left(\frac{-h - y\alpha_j}{\sqrt{1+\alpha_j^2}}\right)$$

For the balanced case, the distribution reduces to the following:

$$r(|t_1| < h, |t_2| < h, ..., |t_n| < h) = \int_0^\infty f(h, y, \rho)^n \phi(y)dy$$

The following relationship applies to the preceding equation:

$$f(h, y, \rho) = \Phi\left(\frac{h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right) - \Phi\left(\frac{-h - y\sqrt{\rho}}{\sqrt{1+\rho}}\right)$$

and $\rho = \frac{1}{n-1}$

Here is the syntax for this distribution:

x=*probmc*('*anom*', *q*,*p*,*nu*,*n*,<*alpha*$_1$,..., *alpha*$_n$>);

Arguments

x
is a numeric value with the returned result.

q
is a numeric value that denotes the quantile.

p
is a numeric value that denotes the probability. One of *p* and *q* must be missing.

nu
is a numeric value that denotes the degrees of freedom.

n
is a numeric value that denotes the number of samples.

alpha$_i$, *i*=1,...,*k*
are optional numeric values denoting the alpha values from the first equation of this distribution (see "Computing the Analysis of Means" on page 759).

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

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\lambda_1, ..., \lambda_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_{0}^{\infty}\int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \Phi\left(\frac{\lambda_i y + qx}{\sqrt{1 - \lambda_i^2}}\right) dy\, du_v(x)$$

- 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:

$$prob = \int_{0}^{\infty}\int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2qx})]^k dy\, du_v(x)$$

- 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:

$$prob = \int_{0}^{\infty}\int_{-\infty}^{\infty} \phi(y) \prod_{i=1}^{k} \left[\Phi\left(\frac{\lambda_i y + qx}{\sqrt{1 - \lambda_i^2}}\right) - \Phi\left(\frac{\lambda_i y - qx}{\sqrt{1 - \lambda_i^2}}\right)\right] dy\, du_v(x)$$

- 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:

$$prob = \int_{0}^{\infty}\int_{-\infty}^{\infty} \phi(y)[\Phi(y + \sqrt{2qx}) - \Phi(y - \sqrt{2qx})]^k dy\, du_v(x)$$

- 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 $n_1 + ... + n_k = n$. Then the partitioned range is the maximum of the studentized ranges in the respective subsets, with the studentization factor being the same in all cases.

$$prob = \int_0^{\infty} \prod_{i=1}^{i=k} \left(\int_{-\infty}^{\infty} k\phi(y)(\Phi(y) - \Phi(y-qx))^{k-1} dy \right)^{n_i} d\mu_v(x)$$

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 762 .

The Studentized Range

Note: The studentized range is not computed for finite degrees of freedom and unequal sample sizes.

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with finite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty}\int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y-qx)]^{k-1} dy\, du_v(x)$$

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with infinite degrees of freedom. The *parameters* are σ_1, ..., σ_k, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} \sum_{j=1}^{k} \left\{ \prod_{i=1}^{k} \left[\Phi\left(\frac{y}{\sigma_i}\right) - \Phi\left(\frac{y-q}{\sigma_i}\right) \right] \right\} \phi\left(\frac{y}{\sigma_j}\right) \frac{1}{\sigma_j} dy$$

- This case relates the probability, *prob*, and the quantile, *q*, for the equal case with infinite degrees of freedom. No *parameters* are passed, the value of *nparms* is set to *k*, and the value of *df* is set to missing. The equation follows:

$$prob = \int_{-\infty}^{\infty} k\phi(y)[\Phi(y) - \Phi(y - q)]^{k-1} dy$$

The Studentized Maximum Modulus

- This case relates the probability, *prob*, and the quantile, *q*, for the unequal case with finite degrees of freedom. The *parameters* are $\sigma_1, ..., \sigma_k$, the value of *nparms* is set to *k*, and the value of *df* is set to *v*. The equation follows:

$$prob = \int_0^{\infty} \prod_{i=1}^{k} \left[2\Phi\left(\frac{qx}{\sigma_i}\right) - 1 \right] d\mu_v(x)$$

- 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:

$$prob = \int_0^{\infty} [2\Phi(qx) - 1]^k d\mu_v(x)$$

- 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

PROBMC computes the probabilities or quantiles from the distribution defined in Williams (1971, 1972) (See "References" on page 1001). It arises when you compare the dose treatment means with a control mean to determine the lowest effective dose of treatment.

Note: Williams' Test is computed only for equal sample sizes.

Let $X_1, X_2, ..., X_k$ be identical independent N(0,1) random variables. Let Y_k denote their average given by

$$Y_k = \frac{X_1 + X_2 + ... + X_k}{k}$$

It is required to compute the distribution of

$$(Y_k - Z) / S$$

Arguments

Y_k
is as defined previously

Z
is an N(0,1) independent random variable

S
is such that $\frac{1}{2}\nu S^2$ is a χ^2 variable with ν degrees of freedom.

As described in Williams (1971) (See "References" on page 1001), the full computation is extremely lengthy and is carried out in three stages.

1. Compute the distribution of Y_k. It is the fundamental (expensive) part of this operation and it can be used to find both the density and the probability of Y_k. Let U_i be defined as

$$U_i = \frac{X_1 + X_2 + \ldots + X_i}{i}, \quad i = 1, 2, \ldots, k$$

You can write a recursive expression for the probability of $Y_k > d$, with d being 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

$$\Pr((Y_k - Z) > d) = \int_{-\infty}^{\infty} \Pr(Y_k > \sqrt{2}d + y)\phi(y)\,dy$$

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

Examples

Example 1: Computing Probabilities by Using PROBMC

This example shows how to compute probabilities.

```
data probs;
   array par{5};
      par{1}=.5;
      par{2}=.51;
      par{3}=.55;
      par{4}=.45;
      par{5}=.2;
   df=40;
   q=1;
   do test="dunnett1","dunnett2", "maxmod";
      prob=probmc(test, q, ., df, 5, of par1-par5);
      put test $10. df q e18.13 prob e18.13;
   end;
run;
```

SAS writes the following results to the log:

Log 2.17 *Probabilities from PROBMC*

```
DUNNETT1  40  1.00000000000E+00 4.82992196083E-01
DUNNETT2  40  1.00000000000E+00 1.64023105316E-01
MAXMOD    40  1.00000000000E+00 8.02784203408E-01
```

Example 2: Computing the Analysis of Means

```
data _null_;
   q1=probmc('anom',.,0.9,.,20);                      put q1=;
   q2=probmc('anom',.,0.9,20,5,0.1,0.1,0.1,0.1,0.1); put q2=;
   q3=probmc('anom',.,0.9,20,5,0.5,0.5,0.5,0.5,0.5); put q3=;
   q4=probmc('anom',.,0.9,20,5,0.1,0.2,0.3,0.4,0.5); put q4=;
run;
```

SAS writes the following output to the log:

Log 2.18 *Output from Analysis of Means*

```
q1=2.7895061016
q2=2.4549961967
q3=2.4549961967
q4=2.4532319994
```

Example 3: Comparing Means

This example shows how to compare group means to find where the significant differences lie. The data for this example is taken from a paper by Duncan (1955), and can also be found in Hochberg and Tamhane (1987) (See the References section at the end of this function.)

The following values are the group means:

- 49.6
- 71.2
- 67.6
- 61.5
- 71.3
- 58.1
- 61.0

For this data, the mean square error is $s^2 = 79.64$ ($s = 8.924$) with $\nu = 30$.

```
data duncan;
   array tr{7}$;
   array mu{7};
   n=7;
   do i=1 to n;
      input tr{i} $1. mu{i};
   end;
   input df s alpha;
   prob= 1-alpha;
      /* compute the interval */
   x = probmc("RANGE", ., prob, df, 7);
   w = x * s / sqrt(6);
      /* compare the means */
   do i = 1 to n;
      do j = i + 1 to n;
         dmean = abs(mu{i} - mu{j});
         if dmean >= w then do;
            put tr{i} tr{j} dmean;
         end;
      end;
   end;
   datalines;
A 49.6
B 71.2
C 67.6
D 61.5
E 71.3
F 58.1
G 61.0
 30 8.924 .05
;
run;
```

SAS writes the following output to the log:

Log 2.19 *Group Differences*

```
A B 21.6
A C 18
A E 21.7
```

Example 4: Computing the Partitioned Range

```
data _null_;
   q1=probmc('partrange',.,0.9,.,4,3,4,5,6);  put q1=;
   q2=probmc('partrange',.,0.9,12,4,3,4,5,6); put q2=;
run;
```

SAS writes the following output to the log:

Log 2.20 *Output from the Partitioned Range*

```
q1=4.1022397989
q2=4.7888626338
```

Example 5: Computing Confidence Intervals

This example shows how to compute 95% one-sided and two-sided confidence intervals of Dunnett's test. This example and the data come from Dunnett (1955), and can also be found in Hochberg and Tamhane (1987) (See the References section at the end of this function.) The data are blood count measurements on three groups of animals. As shown in the following table, the third group serves as the control, while the first two groups were treated with different drugs. The numbers of animals in these three groups are unequal.

Treatment Group:	Drug A	Drug B	Control
	9.76	12.80	7.40
	8.80	9.68	8.50
	7.68	12.16	7.20
	9.36	9.20	8.24
		10.55	9.84
			8.32
Group Mean	8.90	10.88	8.25
n	4	5	6

The mean square error $s^2 = 1.3805$ ($s = 1.175$) with $\nu = 12$.

```
data a;
   array drug{3}$;
   array count{3};
   array mu{3};
   array lambda{2};
   array delta{2};
   array left{2};
   array right{2};
      /* input the table */
   do i = 1 to 3;
      input drug{i} count{i} mu{i};
   end;
```

```
      /* input the alpha level,    */
      /* the degrees of freedom,   */
      /* and the mean square error */
   input alpha df s;

      /* from the sample size, */
      /* compute the lambdas   */
   do i = 1 to 2;
      lambda{i} = sqrt(count{i}/
        (count{i} + count{3}));
   end;
      /* run the one-sided Dunnett's test */
   test="dunnett1";
      x = probmc(test, ., 1 - alpha, df,
                 2, of lambda1-lambda2);
      do i = 1 to 2;
         delta{i} = x * s *
            sqrt(1/count{i} + 1/count{3});
         left{i} = mu{i} - mu{3} - delta{i};
      end;
   put test $10. x left{1} left{2};
      /* run the two-sided Dunnett's test */
   test="dunnett2";
      x = probmc(test, ., 1 - alpha, df,
                 2, of lambda1-lambda2);
      do i=1 to 2;
         delta{i} = x * s *
            sqrt(1/count{i} + 1/count{3});
         left{i} = mu{i} - mu{3} - delta{i};
         right{i} = mu{i} - mu{3} + delta{i};
      end;
   put test $10. left{1} right{1};
   put test $10. left{2} right{2};
   datalines;
A 4 8.90
B 5 10.88
C 6 8.25
0.05 12 1.175
;
run;
```

SAS writes the following output to the log:

Log 2.21 *Confidence Intervals*

```
DUNNETT1  2.1210448226 -0.958726041 1.1208812046
DUNNETT2  -1.256408109 2.5564081095
DUNNETT2  0.8416306717 4.4183693283
```

Example 6: Computing Williams' Test

In the following example, a substance has been tested at seven levels in a randomized block design of eight blocks. The observed treatment means are as follows:

Treatment	Mean
X_0	10.4
X_1	9.9
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.

SAS Statement	Result
`prob=probmc("williams",2.6,.,42,6);`	`0.9924466872`
`quant5=probmc("williams",.,.95,42,6);`	`1.806562536`
`quant1=probmc("williams",.,.99,42,6);`	`2.490908273`

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856

References

Guirguis, G. H., and R. D. Tobias. "On the computation of the distribution for the analysis of means." 2004. *Communications in Statistics: Simulation and Computation* 33: 861–887.

Nelson, P. R. "Numerical evaluation of an equicorrelated multivariate non-central t distribution." 1981. *Communications in Statistics: Part B - Simulation and Computation* 10: 41–50.

Nelson, P. R. "Exact critical points for the analysis of means." 1982. *Communications in Statistics: Part A - Theory and Methods* 11: 699–709.

Nelson, P. R. "An Approximation for the Complex Normal Probability Integral." 1982a. *BIT* 22(1): 94–100.

Nelson, P. R. "Application of the analysis of means." 1988. *Proceedings of the SAS Users Group International Conference* 13: 225–230.

Nelson, P. R. "Numerical evaluation of multivariate normal integrals with correlations." 1991. *The Frontiers of Statistical Scientific Theory and Industrial Applications* 2: 97–114.

Nelson, P. R. "Additional Uses for the Analysis of Means and Extended Tables of Critical Values." 1993. *Technometrics* 35: 61–71.

PROBNEGB Function

Returns the probability from a negative binomial distribution.

Category: Probability

See: "CDF Function" on page 277

Syntax

PROBNEGB(*p*,*n*,*m*)

Required Arguments

p
is a numeric probability of success parameter.

Range $0 \le p \le 1$

n
is an integer number of successes parameter.

Range $n \ge 1$

m
is a positive integer random variable, the number of failures.

Range $m \ge 0$

Details

The PROBNEGB function returns the probability that an observation from a negative binomial distribution, with probability of success p and number of successes n, is less than or equal to m.

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 SAS statement produces this result.

SAS Statement	Result
`x=probnegb(0.5,2,1);`	`0.5`

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "SDF Function" on page 856

PROBNORM Function

Returns the probability from the standard normal distribution.

Category: Probability

See: "CDF Function" on page 277

Syntax

PROBNORM(*x*)

Required Argument

x
is a numeric random variable.

Details

The PROBNORM function returns the probability that an observation from the standard normal distribution is less than or equal to *x*.

Note: PROBNORM is the inverse of the PROBIT function.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=probnorm(1.96);`	`0.9750021049`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “SDF Function” on page 856

PROBT Function

Returns the probability from a *t* distribution.

Category: Probability

See: “CDF Function” on page 277 , “PDF Function” on page 722

Syntax

PROBT(*x*,*df*<,*nc*>)

Required Arguments

x
is a numeric random variable.

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Optional Argument

nc
is an optional numeric noncentrality parameter.

Details

The PROBT function returns the probability that an observation from a Student's *t* distribution, with degrees of freedom *df* and noncentrality parameter *nc*, is less than or equal to *x*. This function accepts a noninteger degree of freedom parameter *df*. If the optional parameter, *nc*, is not specified or has the value 0, the value that is returned is from the central Student's *t* distribution.

The significance level of a two-tailed *t* test is given by

```
p=(1-probt(abs(x),df))*2;
```

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=probt(0.9,5);`	`0.7953143998`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “SDF Function” on page 856

PROPCASE Function

Converts all words in an argument to proper case.

Category: Character

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

Syntax

PROPCASE(*argument* <,*delimiters*>)

Required Argument

argument
specifies a character constant, variable, or expression.

Optional Argument

delimiter
specifies one or more delimiters that are enclosed in quotation marks. The default delimiters are blank, forward slash, hyphen, open parenthesis, period, and tab.

Tip If you use this argument, then the default delimiters, including the blank, are no longer in effect.

Details

Length of Returned Variable

In a DATA step, if the PROPCASE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument that is passed to PROPCASE.

The Basics

The PROPCASE function copies a character argument and converts all uppercase letters to lowercase letters. It then converts to uppercase the first character of a word that is preceded by a blank, forward slash, hyphen, open parenthesis, period, or tab. PROPCASE returns the value that is altered.

If you use the second argument, then the default delimiters are no longer in effect.

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

Examples

Example 1: Changing the Case of Words

The following example shows how PROPCASE handles the case of words:

```
data _null_;
   input place $ 1-40;
   name=propcase(place);
   put name;
   datalines;
INTRODUCTION TO THE SCIENCE OF ASTRONOMY
VIRGIN ISLANDS (U.S.)
SAINT KITTS/NEVIS
WINSTON-SALEM, N.C.
;
```

```
run;
```

SAS writes the following output to the log:

```
Introduction To The Science Of Astronomy
Virgin Islands (U.S.)
Saint Kitts/Nevis
Winston-Salem, N.C.
```

Example 2: Using a Second Argument with PROPCASE

The following example uses a blank, a hyphen and a single quotation mark as the second argument so that names such as O'Keeffe and Burne-Jones are written correctly.

```
data names;
   infile datalines dlm='#';
   input CommonName : $20. CapsName : $20.;
   PropcaseName=propcase(capsname, " -'");
   datalines;
Delacroix, Eugene# EUGENE DELACROIX
O'Keeffe, Georgia# GEORGIA O'KEEFFE
Rockwell, Norman# NORMAN ROCKWELL
Burne-Jones, Edward# EDWARD BURNE-JONES
;
proc print data=names noobs;
   title 'Names of Artists';
run;
```

Display 2.45 *Output Showing the Results of Using PROPCASE with a Second Argument*

Names of Artists

CommonName	CapsName	PropcaseName
Delacroix, Eugene	EUGENE DELACROIX	Eugene Delacroix
O'Keeffe, Georgia	GEORGIA O'KEEFFE	Georgia O'Keeffe
Rockwell, Norman	NORMAN ROCKWELL	Norman Rockwell
Burne-Jones, Edward	EDWARD BURNE-JONES	Edward Burne-Jones

See Also

Functions:

- “UPCASE Function” on page 928
- “LOWCASE Function” on page 646

PRXCHANGE Function

Performs a pattern-matching replacement.

Category: Character String Matching

Syntax

PRXCHANGE(*perl-regular-expressionregular-expression-id*, *times*, *source*)

Required Arguments

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned from the PRXPARSE function.

Restriction If you use this argument, you must also use the PRXPARSE function.

times
is a numeric constant, variable, or expression that specifies the number of times to search for a match and replace a matching pattern.

Tip If the value of *times* is –1, then matching patterns continue to be replaced until the end of *source* is reached.

source
specifies a character constant, variable, or expression that you want to search.

Details

The Basics

If you use *regular-expression-id*, the PRXCHANGE function searches the variable *source* with the *regular-expression-id* that is returned by PRXPARSE. It returns the value in *source* with the changes that were specified by the regular expression. If there is no match, PRXCHANGE returns the unchanged value in *source*.

If you use *perl-regular-expression*, PRXCHANGE searches the variable *source* with the *perl-regular-expression*, and you do not need to call PRXPARSE. You can use PRXCHANGE with a *perl-regular-expression* in a WHERE clause and in PROC SQL.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 42.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant or if it uses the /o option, then the Perl regular expression is compiled once and each use of PRXCHANGE reuses the compiled expression. If *perl-regular-expression* is not a constant and if it does not use the /o option, then the Perl regular expression is recompiled for each call to PRXCHANGE.

Note: The compile-once behavior occurs when you use PRXCHANGE in a DATA step, in a WHERE clause, or in PROC SQL. For all other uses, the *perl-regular-expression* is recompiled for each call to PRXCHANGE.

Performing a Match

Perl regular expressions consist of characters and special characters that are called metacharacters. When performing a match, SAS searches a source string for a substring that matches the Perl regular expression that you specify.

To view a short list of Perl regular expression metacharacters that you can use when you build your code, see the table "Tables of Perl Regular Expression (PRX) Metacharacters" on page 1003. You can find a complete list of metacharacters on the Perl Web site.

Comparisons

The PRXCHANGE function is similar to the CALL PRXCHANGE routine except that the function returns the value of the pattern-matching replacement as a return argument instead of as one of its parameters.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Examples

Example 1: Changing the Order of First and Last Names by Using the DATA Step

The following example uses the DATA step to change the order of first and last names.

```
   /* Create a data set that contains a list of names. */
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;
   /* Reverse last and first names with a DATA step. */
data names;
   set ReversedNames;
   name = prxchange('s/(\w+), (\w+)/$2 $1/', -1, name);
run;
proc print data=names;
run;
```

Display 2.46 *Output from the DATA Step*

The SAS System

Obs	name
1	Fred Jones
2	Kate Kavich
3	Ron Turley
4	Yolanda Dulix

Example 2: Changing the Order of First and Last Names by Using PROC SQL

The following example uses PROC SQL to change the order of first and last names.

```
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;
proc sql;
   create table names as
   select prxchange('s/(\w+), (\w+)/$2 $1/', -1, name) as name
   from ReversedNames;
quit;
proc print data=names;
run;
```

Display 2.47 *Output from PROC SQL*

The SAS System

Obs	name
1	Fred Jones
2	Kate Kavich
3	Ron Turley
4	Yolanda Dulix

Example 3: Matching Rows That Have the Same Name

The following example compares the names in two data sets, and writes those names that are common to both data sets.

```
data names;
   input name & $32.;
   datalines;
Ron Turley
Judy Donnelly
Kate Kavich
Tully Sanchez
;
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
```

```
;
proc sql;
   create table NewNames as
   select a.name from names as a, ReversedNames as b
   where a.name = prxchange('s/(\w+), (\w+)/$2 $1/', -1, b.name);
quit;
proc print data=NewNames;
run;
```

Display 2.48 *Output from Matching Rows That Have the Same Names*

The SAS System

Obs	name
1	Ron Turley
2	Kate Kavich

Example 4: Changing Lowercase Text to Uppercase

The following example uses the \U, \L and \E metacharacters to change the case of a string of text. Case modifications do not nest. In this example, note that "bear" does not convert to uppercase letters because the \E metacharacter ends all case modifications.

```
data _null_;
   length txt $32;
   txt = prxchange ('s/(big)(black)(bear)/\U$1\L$2\E$3/', 1, 'bigblackbear');
   put txt=;
run;
```

SAS returns the following output to the log:

```
txt=BIGblackbear
```

Example 5: Changing a Matched Pattern to a Fixed Value

This example locates a pattern in a variable and replaces the variable with a predefined value. The example uses a DATA step to find phone numbers and replace them with an informational message.

```
   /* Create data set that contains confidential information. */
data a;
   input text $80.;
   datalines;
The phone number for Ed is (801)443-9876 but not until tonight.
He can be reached at (910)998-8762 tomorrow for testing purposes.
;
run;
   /* Locate confidential phone numbers and replace them with message */
   /* indicating that they have been removed.                         */
data b;
   set a;
   text = prxchange('s/\([2-9]\d\d\) ?[2-9]\d\d-\d\d\d\d/*PHONE NUMBER
      REMOVED*/', -1, text);
```

```
   put text=;
run;
proc print data = b;
run;
```

Display 2.49 *Output from Changing a Matched Pattern to a Fixed Value*

The SAS System

Obs	text
1	The phone number for Ed is *PHONE NUMBER REMOVED* but not until tonight.
2	He can be reached at *PHONE NUMBER REMOVED* tomorrow for testing purposes.

See Also

Functions:

- “PRXMATCH Function” on page 780
- “PRXPAREN Function” on page 784
- “PRXPARSE Function” on page 786
- “PRXPOSN Function” on page 788

CALL Routines:

- “CALL PRXCHANGE Routine” on page 198
- “CALL PRXDEBUG Routine” on page 200
- “CALL PRXFREE Routine” on page 202
- “CALL PRXNEXT Routine” on page 203
- “CALL PRXPOSN Routine” on page 205
- “CALL PRXSUBSTR Routine” on page 208

PRXMATCH Function

Searches for a pattern match and returns the position at which the pattern is found.

Category: Character String Matching

Syntax

PRXMATCH (*regular-expression-id* | *perl-regular-expression*, *source*)

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned from the PRXPARSE function.

Restriction If you use this argument, you must also use the PRXPARSE function.

perl-regular-expression
specifies a character constant, variable, or expression with a value that is a Perl regular expression.

source
specifies a character constant, variable, or expression that you want to search.

Details

The Basics

If you use *regular-expression-id*, then the PRXMATCH function searches *source* with the *regular-expression-id* that is returned by PRXPARSE, and returns the position at which the string begins. If there is no match, PRXMATCH returns a zero.

If you use *perl-regular-expression*, PRXMATCH searches *source* with the *perl-regular-expression*, and you do not need to call PRXPARSE.

You can use PRXMATCH with a Perl regular expression in a WHERE clause and in PROC SQL. For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 42.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant or if it uses the /o option, then the Perl regular expression is compiled once and each use of PRXMATCH reuses the compiled expression. If *perl-regular-expression* is not a constant and if it does not use the /o option, then the Perl regular expression is recompiled for each call to PRXMATCH.

Note: The compile-once behavior occurs when you use PRXMATCH in a DATA step, in a WHERE clause, or in PROC SQL. For all other uses, the *perl-regular-expression* is recompiled for each call to PRXMATCH.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Examples

Example 1: Finding the Position of a Substring by Using PRXPARSE

The following example searches a string for a substring, and returns its position in the string.

```
/* For 9.0: the following example makes a call to PRXPARSE. */
  /* For 9.1, no call is required. */

     data _null_;
         /* Use PRXPARSE to compile the Perl regular expression.    */
      patternID = prxparse('/world/');
         /* Use PRXMATCH to find the position of the pattern match. */
      position=prxmatch(patternID, 'Hello world!');
      put position=;
   run;
```

SAS writes the following line to the log:

```
position=7
```

Example 2: Finding the Position of a Substring by Using a Perl Regular Expression

The following example uses a Perl regular expression to search a string (Hello world) for a substring (world) and to return the position of the substring in the string.

```
data _null_;
      /* Use PRXMATCH to find the position of the pattern match. */
   position=prxmatch('/world/', 'Hello world!');
   put position=;
run;
```

SAS writes the following line to the log:

```
position=7
```

Example 3: Finding the Position of a Substring in a String: A Complex Example

The following example uses several Perl regular expression functions and a CALL routine to find the position of a substring in a string.

```
data _null_;
   if _N_ = 1 then
   do;
      retain PerlExpression;
      pattern = "/(\d+):(\d\d)(?:\.(\d+))?/";
      PerlExpression = prxparse(pattern);
   end;

   array match[3] $ 8;
   input minsec $80.;
   position = prxmatch(PerlExpression, minsec);
   if position ^= 0 then
   do;
      do i = 1 to prxparen(PerlExpression);
         call prxposn(PerlExpression, i, start, length);
         if start ^= 0 then
            match[i] = substr(minsec, start, length);
      end;
      put match[1] "minutes, " match[2] "seconds" @;
      if ^missing(match[3]) then
         put ", " match[3] "milliseconds";
   end;
   datalines;
14:56.456
45:32
;
run;
```

The following lines are written to the SAS log:

```
14 minutes, 56 seconds, 456 milliseconds
45 minutes, 32 seconds
```

Example 4: Extracting a ZIP Code by Using the DATA Step

The following example uses a DATA step to search each observation in a data set for a nine-digit ZIP code, and writes those observations to the data set ZipPlus4.

```
data ZipCodes;
   input name: $16. zip:$10.;
   datalines;
Johnathan 32523-2343
Seth 85030
Kim 39204
Samuel 93849-3843
;
   /* Extract ZIP+4 ZIP codes with the DATA step. */
data ZipPlus4;
   set ZipCodes;
   where prxmatch('/\d{5}-\d{4}/', zip);
run;
proc print data=ZipPlus4;
run;
```

Display 2.50 *ZIP Code Output from the DATA Step*

The SAS System

Obs	name	zip
1	Johnathan	32523-2343
2	Samuel	93849-3843

Example 5: Extracting a ZIP Code by Using PROC SQL

The following example searches each observation in a data set for a nine-digit ZIP code, and writes those observations to the data set ZipPlus4.

```
data ZipCodes;
   input name: $16. zip:$10.;
   datalines;
Johnathan 32523-2343
Seth 85030
Kim 39204
Samuel 93849-3843
;
   /* Extract ZIP+4 ZIP codes with PROC SQL. */
proc sql;
   create table ZipPlus4 as
   select * from ZipCodes
   where prxmatch('/\d{5}-\d{4}/', zip);
run;
proc print data=ZipPlus4;
run;
```

Display 2.51 *ZIP Code Output from PROC SQL*

The SAS System

Obs	name	zip
1	Johnathan	32523-2343
2	Samuel	93849-3843

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXPAREN Function" on page 784
- "PRXPARSE Function" on page 786
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202
- "CALL PRXNEXT Routine" on page 203
- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208
- "CALL PRXCHANGE Routine" on page 198

PRXPAREN Function

Returns the last bracket match for which there is a match in a pattern.

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Syntax

PRXPAREN (*regular-expression-id*)

Required Argument

regular-expression-id
specifies a numeric variable with a value that is an identification number that is returned by the PRXPARSE function.

Details

The PRXPAREN function is useful in finding the largest capture-buffer number that can be passed to the CALL PRXPOSN routine, or in identifying which part of a pattern matched.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 42 .

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following example uses Perl regular expressions and writes the results to the SAS log.

```
data _null_;
   ExpressionID = prxparse('/(magazine)|(book)|(newspaper)/');
   position = prxmatch(ExpressionID, 'find book here');
   if position then paren = prxparen(ExpressionID);
   put 'Matched paren ' paren;
   position = prxmatch(ExpressionID, 'find magazine here');
   if position then paren = prxparen(ExpressionID);
   put 'Matched paren ' paren;
   position = prxmatch(ExpressionID, 'find newspaper here');
   if position then paren = prxparen(ExpressionID);
   put 'Matched paren ' paren;
run;
```

The following lines are written to the SAS log:

```
Matched paren 2
Matched paren 1
Matched paren 3
```

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXMATCH Function" on page 780
- "PRXPARSE Function" on page 786
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202
- "CALL PRXNEXT Routine" on page 203

- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208
- "CALL PRXCHANGE Routine" on page 198

PRXPARSE Function

Compiles a Perl regular expression (PRX) that can be used for pattern matching of a character value.

Category: Character String Matching

Restriction: Use with other Perl regular expressions.

Syntax

regular-expression-id=**PRXPARSE** (*perl-regular-expression*)

Required Arguments

regular-expression-id
is a numeric pattern identifier that is returned by the PRXPARSE function.

perl-regular-expression
specifies a character value that is a Perl regular expression.

Details

The Basics

The PRXPARSE function returns a pattern identifier number that is used by other Perl functions and CALL routines to match patterns. If an error occurs in parsing the regular expression, SAS returns a missing value.

PRXPARSE uses metacharacters in constructing a Perl regular expression. To view a table of common metacharacters, see "Tables of Perl Regular Expression (PRX) Metacharacters" on page 1003.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 42.

Compiling a Perl Regular Expression

If *perl-regular-expression* is a constant or if it uses the /o option, the Perl regular expression is compiled only once. Successive calls to PRXPARSE 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.

Note: If you have a Perl regular expression that is a constant, or if the regular expression uses the /o option, then calling PRXFREE to free the memory allocation results in the need to recompile the regular expression the next time that it is called by PRXPARSE.The compile-once behavior occurs when you use PRXPARSE in a DATA step. For all other uses, the *perl-regular-expression* is recompiled for each call to PRXPARSE.

Comparisons

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following example uses metacharacters and regular characters to construct a Perl regular expression. The example parses addresses and writes formatted results to the SAS log.

```
data _null_;
   if _N_ = 1 then
   do;
      retain patternID;
         /* The i option specifies a case insensitive search. */
      pattern = "/ave|avenue|dr|drive|rd|road/i";
      patternID = prxparse(pattern);
   end;
   input street $80.;
   call prxsubstr(patternID, street, position, length);
   if position ^= 0 then
   do;
      match = substr(street, position, length);
      put match:$QUOTE. "found in " street:$QUOTE.;
   end;
   datalines;
153 First Street
6789 64th Ave
4 Moritz Road
7493 Wilkes Place
;
```

The following lines are written to the SAS log:

```
"Ave" found in "6789 64th Ave"
"Road" found in "4 Moritz Road"
```

See Also

Functions:

- "PRXCHANGE Function" on page 775
- "PRXPAREN Function" on page 784
- "PRXMATCH Function" on page 780
- "PRXPOSN Function" on page 788

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202

- "CALL PRXNEXT Routine" on page 203
- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208
- "CALL PRXCHANGE Routine" on page 198

PRXPOSN Function

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

Category: Character String Matching

Restriction: Use with the PRXPARSE function.

Syntax

PRXPOSN(*regular-expression-id*, *capture-buffer*, *source*)

Required Arguments

regular-expression-id
specifies a numeric variable with a value that is a pattern identifier that is returned by the PRXPARSE function.

capture-buffer
is a numeric constant, variable, or expression that identifies the capture buffer for which to retrieve a value:

- If the value of *capture-buffer* is zero, PRXPOSN returns the entire match.
- If the value of *capture-buffer* is between 1 and the number of open parentheses in the regular expression, then PRXPOSN returns the value for that capture buffer.
- If the value of *capture-buffer* is greater than the number of open parentheses, then PRXPOSN returns a missing value.

source
specifies the text from which to extract capture buffers.

Details

The PRXPOSN function uses the results of PRXMATCH, PRXSUBSTR, PRXCHANGE, or PRXNEXT to return a capture buffer. A match must be found by one of these functions for PRXPOSN to return meaningful information.

A capture buffer is part of a match, enclosed in parentheses, that is specified in a regular expression. This function simplifies using capture buffers by returning the text for the capture buffer directly, and by not requiring a call to SUBSTR as in the case of CALL PRXPOSN.

For more information about pattern matching, see "Pattern Matching Using Perl Regular Expressions (PRX) " on page 42 .

Comparisons

The PRXPOSN function is similar to the CALL PRXPOSN routine, except that it returns the capture buffer itself rather than the position and length of the capture buffer.

The Perl regular expression (PRX) functions and CALL routines work together to manipulate strings that match patterns. To see a list and short description of these functions and CALL routines, see the Character String Matching category in "SAS Functions and CALL Routines by Category" on page 65.

Examples

Example 1: Extracting First and Last Names

The following example uses PRXPOSN to extract first and last names from a data set.

```
data ReversedNames;
   input name & $32.;
   datalines;
Jones, Fred
Kavich, Kate
Turley, Ron
Dulix, Yolanda
;
data FirstLastNames;
   length first last $ 16;
   keep first last;
   retain re;
   if _N_ = 1 then
      re = prxparse('/(\w+), (\w+)/');
   set ReversedNames;
   if prxmatch(re, name) then
      do;
         last = prxposn(re, 1, name);
         first = prxposn(re, 2, name);
      end;
run;
proc print data = FirstLastNames;
run;
```

Display 2.52 *Output from PRXPOSN: First and Last Names*

The SAS System

Obs	first	last
1	Fred	Jones
2	Kate	Kavich
3	Ron	Turley
4	Yolanda	Dulix

Example 2: Extracting Names When Some Names Are Invalid

The following example creates a data set that contains a list of names. Observations that have only a first name or only a last name are invalid. PRXPOSN extracts the valid names from the data set, and writes the names to the data set NEW.

```
data old;
   input name $60.;
   datalines;
Judith S Reaveley
Ralph F. Morgan
Jess Ennis
Carol Echols
Kelly Hansen Huff
Judith
Nick
Jones
;
data new;
   length first middle last $ 40;
   keep first middle last;
   re = prxparse('/(\S+)\s+([^\s]+\s+)?(\S+)/o');
   set old;
   if prxmatch(re, name) then
      do;
         first = prxposn(re, 1, name);
         middle = prxposn(re, 2, name);
         last = prxposn(re, 3, name);
         output;
      end;
run;
proc print data = new;
run;
```

Display 2.53 *Output of Valid Names*

The SAS System

Obs	first	middle	last
1	Judith	S	Reaveley
2	Ralph	F.	Morgan
3	Jess		Ennis
4	Carol		Echols
5	Kelly	Hansen	Huff

See Also

Functions:

- “PRXCHANGE Function” on page 775
- “PRXMATCH Function” on page 780
- “PRXPAREN Function” on page 784
- “PRXPARSE Function” on page 786

CALL Routines:

- "CALL PRXCHANGE Routine" on page 198
- "CALL PRXDEBUG Routine" on page 200
- "CALL PRXFREE Routine" on page 202
- "CALL PRXNEXT Routine" on page 203
- "CALL PRXPOSN Routine" on page 205
- "CALL PRXSUBSTR Routine" on page 208
- "CALL PRXCHANGE Routine" on page 198

PTRLONGADD Function

Returns the pointer address as a character variable on 32-bit and 64-bit platforms.

Category: Special

Syntax

PTRLONGADD(*pointer*<,*amount*>)

Required Arguments

pointer
is a character constant, variable, or expression that specifies the pointer address.

amount
is a numeric constant, variable, or expression that specifies the amount to add to the address.

Tip *amount* can be a negative number.

Details

The PTRLONGADD function performs pointer arithmetic and returns a pointer address as a character string.

Example

The following example returns the pointer address for the variable Z.

```
data _null_;
   x='ABCDE';
   y=ptrlongadd(addrlong(x),2);
   z=peekclong(y,1);
   put z=;
run;
```

The output from the SAS log is: **z=C**

PUT Function

Returns a value using a specified format.

Category: Special

Syntax

PUT(*source*, *format*.)

Required Arguments

source
identifies the constant, variable, or expression whose value you want to reformat. The *source* argument can be character or numeric.

***format*.**
contains the SAS format that you want applied to the value that is specified in the source. This argument must be the name of a format with a period and optional width and decimal specifications, not a character constant, variable, or expression. By default, if the source is numeric, the resulting string is right aligned, and if the source is character, the result is left aligned. To override the default alignment, you can add an alignment specification to a format:

-L left aligns the value.

-C centers the value.

-R right aligns the value.

Restriction The *format*. must be of the same type as the source, either character or numeric. That is, if the source is character, the format name must begin with a dollar sign, but if the source is numeric, the format name must not begin with a dollar sign.

Details

If the PUT function returns a value to a variable that has not yet been assigned a length, by default the variable length is determined by the width of the format.

Use the PUT function to convert a numeric value to a character value. The PUT function has no effect on which formats are used in PUT statements or which formats are assigned to variables in data sets. You cannot use the PUT function to directly change the type of variable in a data set from numeric to character. However, you can create a new character variable as the result of the PUT function. Then, if needed, use the DROP statement to drop the original numeric variable, followed by the RENAME statement to rename the new variable back to the original variable name.

Comparisons

The PUT statement and the PUT function are similar. The PUT function returns a value using a specified format. You must use an assignment statement to store the value in a variable. The PUT statement writes a value to an external destination (either the SAS log or a destination you specify).

Examples

Example 1: Converting Numeric Values to Character Value

In this example, the first statement converts the values of *cc*, a numeric variable, into the four-character hexadecimal format, and the second statement writes the same value that the PUT function returns.

```
cchex=put(cc,hex4.);
put cc hex4.;
```

If you need to keep the original variable name of *cc*, but as a character variable, then use the DROP and RENAME statements following the PUT function.

```
cchex=put(cc,hex4.);
drop cc;
rename cchex=cc;
```

The new *cchex* variable is created as a character variable from the numeric value of the *cc* variable. The DROP statement prevents the numeric *cc* variable from being written to the data set, and the RENAME statement renames the new character *cchex* variable back to the name of *cc*.

Example 2: Using PUT and INPUT Functions

In this example, the PUT function returns a numeric value as a character string. The value 122591 is assigned to the CHARDATE variable. The INPUT function returns the value of the character string as a SAS date value using a SAS date informat. The value 11681 is stored in the SASDATE variable.

```
numdate=122591;
chardate=put(numdate,z6.);
sasdate=input(chardate,mmddyy6.);
```

See Also

Functions:

- “INPUT Function” on page 550
- “INPUTC Function” on page 552
- “INPUTN Function” on page 554
- “PUTC Function” on page 793
- “PUTN Function” on page 795

Statements:

- “PUT Statement” in *SAS Statements: Reference*

PUTC Function

Enables you to specify a character format at run time.

Category: Special

Syntax

PUTC(*source*, *format*.<,*w*>)

Required Arguments

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

format.
is a character constant, variable, or expression with a value that is the character format you want to apply to *source*.

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, by default 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 lets you specify a format at compile time rather than at run time.

Example

The PROC FORMAT step in this example creates a format, TYPEFMT., that formats the variable values 1, 2, and 3 with the name of one of the three other formats that this step creates. These three formats output responses of "positive," "negative," and "neutral" as different words, depending on the type of question. After PROC FORMAT creates the formats, the DATA step creates a SAS data set from raw data consisting of a number identifying the type of question and a response. After reading a record, the DATA step uses the value of TYPE to create a variable, RESPFMT, that contains the value of the appropriate format for the current type of question. The DATA step also creates another variable, WORD, whose value is the appropriate word for a response. The PUTC function assigns the value of WORD based on the type of question and the appropriate format.

```
proc format;
   value typefmt 1='$groupx'
                 2='$groupy'
                 3='$groupz';
   value $groupx 'positive'='agree'
                 'negative'='disagree'
                 'neutral'='notsure ';
   value $groupy 'positive'='accept'
```

```
                'negative'='reject'
                'neutral'='possible';
   value $groupz 'positive'='pass    '
                'negative'='fail'
                'neutral'='retest';
run;
data answers;
   length word $ 8;
   input type response $;
   respfmt = put(type, typefmt.);
   word = putc(response, respfmt);
   datalines;
1 positive
1 negative
1 neutral
2 positive
2 negative
2 neutral
3 positive
3 negative
3 neutral
;

proc print data=answers;
run;
SAS log:
Obs    word        type    response    respfmt
 2     disagree      1     negative    $groupx
 3     notsure       1     neutral     $groupx
 4     accept        2     positive    $groupy
 5     reject        2     negative    $groupy
 6     possible      2     neutral     $groupy
 7     pass          3     positive    $groupz
 8     fail          3     negative    $groupz
 9     retest        3     neutral     $groupz
```

The value of the variable WORD is **agree** for the first observation. The value of the variable WORD is **retest** for the last observation.

See Also

Functions:

- “INPUT Function” on page 550
- “INPUTC Function” on page 552
- “INPUTN Function” on page 554
- “PUT Function” on page 791
- “PUTN Function” on page 795

PUTN Function

Enables you to specify a numeric format at run time.

Category: Special

Syntax

PUTN(*source*, *format*.<,*w*<,*d*> >)

Required Arguments

source
specifies a numeric constant, variable, or expression to which you want to apply the format.

format.
is a character constant, variable, or expression with a value that is the numeric format you want to apply to *source*.

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.

Comparisons

The PUTC function enables you to specify a character format at run time.

The PUT function is faster than PUTN because PUT lets you specify a format at compile time rather than at run time.

Example

The PROC FORMAT step in this example creates a format, WRITFMT., that formats the variable values 1 and 2 with the name of a SAS date format. The DATA step creates a SAS data set from raw data consisting of a number and a key. After reading a record, the DATA step uses the value of KEY to create a variable, DATEFMT, that contains the value of the appropriate date format. The DATA step also creates a new variable, DATE, whose value is the formatted value of the date. PUTN assigns the value of DATE based on the value of NUMBER and the appropriate format.

```
proc format;
   value writfmt 1='date9.'
```

```
                    2='mmddyy10.';
run;
data dates;
   input number key;
   datefmt=put(key,writfmt.);
   date=putn(number,datefmt);
   datalines;
15756 1
14552 2
;
```

See Also

Functions:

- "INPUT Function" on page 550
- "INPUTC Function" on page 552
- "INPUTN Function" on page 554
- "PUT Function" on page 791
- "PUTC Function" on page 793

PVP Function

Returns the present value for a periodic cash flow stream (such as a bond), with repayment of principal at maturity.

Category: Financial

Syntax

PVP(A,c,n,K,k_0,y)

Required Arguments

A

specifies the par value.

Range: $A > 0$

c

specifies the nominal per-year coupon rate, expressed as a fraction.

Range: $0 \le c < 1$

n

specifies the number of coupons per year.

Range: $n > 0$ and is an integer

K

specifies the number of remaining coupons.

Range: $K > 0$ and is an integer

k_0

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

Range: $0 < k_0 \leq \frac{1}{n}$

y

specifies the nominal per-year yield-to-maturity, expressed as a fraction.

Range: $y > 0$

Details

The PVP function is based on the relationship

$$P = \sum_{k=1}^{K} \frac{c(k)}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

The following relationships apply to the preceding equation:

- $t_k = nk_0 + k - 1$
- $c(k) = \frac{c}{n}A \quad for\ k = 1, \ \ldots, \ K - 1$
- $c(K) = \left(1 + \frac{c}{n}\right)A$

Example

```
data _null_;
p=pvp(1000,.01,4,14,.33/2,.10);
put p;
run;
```

The value that is returned is 743.168.

QTR Function

Returns the quarter of the year from a SAS date value.

Category: Date and Time

Syntax

QTR(*date*)

Required Argument

date

specifies a numeric constant, variable, or expression that represents a SAS date value.

Details

The QTR function returns a value of 1, 2, 3, or 4 from a SAS date value to indicate the quarter of the year in which a date value falls.

Example

The following SAS statements produce these results.

SAS Statement	Result

`x='20jan94'd;`
`y=qtr(x);`
`put y=;`
	`y=1`

See Also

Functions:

- "YYQ Function" on page 989

QUANTILE Function

Returns the quantile from a distribution when you specify the left probability (CDF).

Category: Quantile

See: "CDF Function" on page 277

Syntax

QUANTILE(*dist*,*probability*,*parm-1*,...,*parm-k*)

Required Arguments

dist

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	**BERNOULLI**
Beta	**BETA**
Binomial	**BINOMIAL**
Cauchy	**CAUCHY**
Chi-Square	**CHISQUARE**

Distribution	Argument
Exponential	**EXPONENTIAL**
F	**F**
Gamma	**GAMMA**
Generalized Poisson	**GENPOISSON**
Geometric	**GEOMETRIC**
Hypergeometric	**HYPERGEOMETRIC**
Laplace	**LAPLACE**
Logistic	**LOGISTIC**
Lognormal	**LOGNORMAL**
Negative binomial	**NEGBINOMIAL**
Normal	**NORMAL**|**GAUSS**
Normal mixture	**NORMALMIX**
Pareto	**PARETO**
Poisson	**POISSON**
T	**T**
Tweedie	**TWEEDIE**
Uniform	**UNIFORM**
Wald (inverse Gaussian)	**WALD**|**IGAUSS**
Weibull	**WEIBULL**

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

probability
is a numeric constant, variable, or expression that specifies the value of a random variable.

parm-1,…,parm-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The QUANTILE function computes the probability from various continuous and discrete distributions. For more information, see the "Details" on page 279 section of the CDF function.

Example

The following SAS statements produce these results.

SAS Statement	Result
y=quantile('BERN',.75,.25);	0
y=quantile('BETA',0.1,3,4);	0.2009088789
y=quantile('BINOM',.4,.5,10);	5
y=quantile('CAUCHY',.85);	1.9626105055
y=quantile('CHISQ',.6,11);	11.529833841
y=quantile('EXPO',.6);	0.9162907319
y=quantile('F',.8,2,3);	2.8860266073
y=quantile('GAMMA',.4,3);	2.285076904
y=quantile('GENPOISSON',.9,1,.7);	9
y=quantile('HYPER',.5,200,50,10);	2
y=quantile('LAPLACE',.8);	0.9162907319
y=quantile('LOGISTIC',.7);	0.8472978604
y=quantile('LOGNORMAL',.5);	1
y=quantile('NEGB',.5,.5,2);	1
y=quantile('NORMAL',.975);	1.9599639845
y=quantile('PARETO',.01,1);	1.0101010101
y=quantile('POISSON',.9,1);	2
y=quantile('T',.8,5);	0.9195437802
y=quantile('TWEEDIE',.8,5);	1.261087383
y=quantile('UNIFORM',0.25);	0.25

SAS Statement	Result
`y=quantile('WALD',.6,2);`	`0.9526209927`
`y=quantile('WEIBULL',.6,2);`	`0.9572307621`

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “SDF Function” on page 856
- “SQUANTILE Function” on page 881

QUOTE Function

Adds double quotation marks to a character value.

Category: Character

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

Syntax

QUOTE(*argument-1*, *argument-2*)

Required Arguments

argument-1
specifies a character constant, variable, or expression.

argument-2
specifies a quoting character, which is a single or double quotation mark. Other characters are ignored and the double quotation mark is used. The double quotation mark is the default.

Details

Length of Returned Variable

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

The Basics

The QUOTE function adds double quotation marks, the default character, to a character value. If double quotation marks are found within the argument, they are doubled in the output.

The length of the receiving variable must be long enough to contain the argument (including trailing blanks), leading and trailing quotation marks and any embedded quotation marks that are doubled. For example, if the argument is ABC followed by three trailing blanks, then the receiving variable must have a length of at least eight to hold “ABC###”. (The character # represents a blank space.) If the receiving field is not long enough, the QUOTE function returns a blank string, and writes an invalid argument note to the log.

Example

The following SAS statements produce these results.

SAS Statement	Result

`x='A"B';`
`y=quote(x);`
`put y;`
	`"A""B"`

`x='A''B';`
`y=quote(x);`
`put y;`
	`"A'B"`

`x='Paul''s';`
`y=quote(x);`
`put y;`
	`"Paul's"`

`x='Catering Service Center  ';`
`y=quote(x);`
`put y;`
	`"Catering Service Center  "`

`x='Paul''s Catering Service  ';`
`y=quote(trim(x));`
`puty;`
	`"Paul's Catering Service:`

`x=quote('abc');` `put x=;`	`"abc"`

`x=quote('abc',"'");`
`put x=;`
Note: The second argument contains a single quotation mark. In order to be passed to the function in the DATA step, the argument is specified in the DATA step as double quotation mark, single quotation mark, double quotation mark. The argument could have also been specified as four single quotation marks (that is, a quoted string that uses single quotation marks). The quoted value is an escaped single quotation mark represented as two single quotation marks.
	`'abc'`

RANBIN Function

Returns a random variate from a binomial distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANBIN routine instead of the RANBIN function.

Syntax

RANBIN(*seed*,*n*,*p*)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Seed Values" on page 11 for more information about seed values

n

is a numeric constant, variable, or expression with an integer value that specifies the number of independent Bernoulli trials parameter.

Range $n > 0$

p

is a numeric constant, variable, or expression that specifies the probability of success.

Range $0 < p < 1$

Details

The RANBIN function returns a variate that is generated from a binomial distribution with mean *np* and variance $np(1-p)$. If $n \leq 50$, $np \leq 5$, or $n(1-p) \leq 5$, an inverse transform method applied to a RANUNI uniform variate is used. If $n > 50$, $np > 5$, and $n(1-p) > 5$, the normal approximation to the binomial distribution is used. In that case, the Box-Muller transformation of RANUNI uniform variates is used.

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "Generating Multiple Variables from One Seed in Random-Number Functions" on page 22 .

Comparisons

The CALL RANBIN routine, an alternative to the RANBIN function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- "CALL RANBIN Routine" on page 210

RANCAU Function

Returns a random variate from a Cauchy distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANCAU routine instead of the RANCAU function.

Syntax

RANCAU(*seed*)

Required Argument

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31} - 1$

See "Seed Values" on page 11 for more information about seed values

Details

The RANCAU function returns a variate that is generated from a Cauchy distribution with location parameter 0 and scale parameter 1. An acceptance-rejection procedure applied to RANUNI uniform variates is used. If u and v are independent uniform (−1/2, 1/2) variables and $u^2 + v^2 \le 1/4$, then u/v is a Cauchy variate. A Cauchy variate X with location parameter ALPHA and scale parameter BETA can be generated:

```
x=alpha+beta*rancau(seed);
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "Generating Multiple Variables from One Seed in Random-Number Functions" on page 22.

Comparisons

The CALL RANCAU routine, an alternative to the RANCAU function, gives greater control of the seed and random number streams.

See Also

Functions:

- “RAND Function” on page 806

CALL Routines:

- “CALL RANCAU Routine” on page 212

RAND Function

Generates random numbers from a distribution that you specify.

Category: Random Number

Syntax

RAND (*dist*, *parm-1*,…,*parm-k*)

Required Arguments

dist

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	`BERNOULLI`
Beta	`BETA`
Binomial	`BINOMIAL`
Cauchy	`CAUCHY`
Chi-Square	`CHISQUARE`
Erlang	`ERLANG`
Exponential	`EXPONENTIAL`
F	`F`
Gamma	`GAMMA`
Geometric	`GEOMETRIC`
Hypergeometric	`HYPERGEOMETRIC`
Lognormal	`LOGNORMAL`

Distribution	Argument
Negative Binomial	**NEGBINOMIAL**
Normal	**NORMAL** | **GAUSSIAN**
Poisson	**POISSON**
T	**T**
Tabled	**TABLE**
Triangular	**TRIANGLE**
Uniform	**UNIFORM**
Weibull	**WEIBULL**

Note: Except for T and F, you can minimally identify any distribution by its first four characters.

parm-1,…,parm-k
are *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

See "Details" on page 807

Details

Generating Random Numbers

The RAND function generates random numbers from various continuous and discrete distributions. Wherever possible, the simplest form of the distribution is used.

The RAND function uses the Mersenne-Twister random number generator (RNG) that was developed by Matsumoto and Nishimura (1998). The random number generator has a very long period ($2^{19937} - 1$) and very good statistical properties. The period is a Mersenne prime, which contributes to the naming of the RNG. The algorithm is a twisted generalized feedback shift register (TGFSR) that explains the latter part of the name. The TGFSR gives the RNG a very high order of equidistribution (623-dimensional with 32-bit accuracy), which means that there is a very small correlation between successive vectors of 623 pseudo-random numbers.

The RAND function is started with a single seed. However, the state of the process cannot be captured by a single seed. You cannot stop and restart the generator from its stopping point.

Reproducing a Random Number Stream

If you want to create reproducible streams of random numbers, then use the CALL STREAMINIT routine to specify a seed value for random number generation. Use the CALL STREAMINIT routine once per DATA step before any invocation of the RAND function. If you omit the call to the CALL STREAMINIT routine (or if you specify a non-positive seed value in the CALL STREAMINIT routine), then RAND uses a call to the system clock to seed itself. For more information, see CALL STREAMINIT "Example: Creating a Reproducible Stream of Random Numbers" on page 255.

Duplicate Values in the Mersenne-Twister RNG Algorithm

The Mersenne-Twister RNG algorithm has an extremely long period, but this does not imply that large random samples are devoid of duplicate values. The RAND function returns at most 2^{32} distinct values. In a random uniform sample of size 10^5, the chance of drawing at least one duplicate is greater than 50%. The expected number of duplicates in a random uniform sample of size M is approximately $M^2/2^{33}$ when M is much less than 2^{32}. For example, you should expect about 115 duplicates in a random uniform sample of size $M=10^6$. These results are consequences of the famous "birthday matching problem" in probability theory.

Bernoulli Distribution

x = **RAND**('BERNOULLI',*p*)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \begin{cases} 1 & p = 0,\ x = 0 \\ p^x(1-p)^{1-x} & 0 < p < 1,\ x = 0,\ 1 \\ 1 & p = 1,\ x = 1 \end{cases}$$

Range $x = 0, 1$

p

is a numeric probability of success.

Range $0 \le p \le 1$

Beta Distribution

x = **RAND**('BETA',*a*,*b*)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}$$

Range $0 < x < 1$

a

is a numeric shape parameter.

Range $a > 0$

b

is a numeric shape parameter.

Range $b > 0$

Binomial Distribution

x = **RAND**('BINOMIAL',*p*,*n*)

Arguments

x

is an integer observation from the distribution with the following probability density function:

$$f(x) = \begin{cases} 1 & p = 0,\ x = 0 \\ \binom{n}{x} p^x (1 - p)^{n-x} & 0 < p < 1,\ x = 0, \ldots, n \\ 1 & p = 1,\ x = n \end{cases}$$

Range $x = 0, 1, \ldots, n$

p

is a numeric probability of success.

Range $0 \leq p \leq 1$

n

is an integer parameter that counts the number of independent Bernoulli trials.

Range $n = 1, 2, \ldots$

Cauchy Distribution

x = **RAND**('CAUCHY')

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{1}{\pi\left(1 + x^2\right)}$$

Range $-\infty < x < \infty$

Chi-Square Distribution

x = **RAND**('CHISQUARE',*df*)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{2^{-df/2}}{\Gamma\left(\frac{df}{2}\right)} x^{df/2-1} e^{-x/2}$$

Range $x > 0$

df

is a numeric degrees of freedom parameter.

Range $df > 0$

Erlang Distribution

x = **RAND**('ERLANG',*a*)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

Range $x > 0$

a

is an integer numeric shape parameter.

Range $a = 1, 2, \ldots$

Exponential Distribution

x = **RAND**('EXPONENTIAL')

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = e^{-x}$$

Range $x > 0$

F Distribution

x = **RAND**('F',*n*, *d*)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{\Gamma\left(\frac{n+d}{2}\right)}{\Gamma\left(\frac{n}{2}\right)\Gamma\left(\frac{d}{2}\right)} \frac{n^{n/2} d^{d/2} x^{n/2-1}}{(d+nx)^{(n+d)/2}}$$

Range $x > 0$

n

is a numeric numerator degrees of freedom parameter.

Range $n > 0$

d

is a numeric denominator degrees of freedom parameter.

Range $d > 0$

Gamma Distribution

x = **RAND**('GAMMA',*a*)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x}$$

Range $x > 0$

a

is a numeric shape parameter.

Range $a > 0$

Geometric Distribution

x = **RAND**('GEOMETRIC',*p*)

Arguments

x

is an integer count that denotes the number of trials that are needed to obtain one success. *X* is an integer observation from the distribution with the following probability density function:

$$f(x) = \begin{cases} (1 - p)^{x-1} p & 0 < p < 1,\ x = 1, 2, \ldots \\ 1 & p = 1,\ x = 1 \end{cases}$$

Range $x = 1, 2, \ldots$

p

is a numeric probability of success.

Range $0 < p \leq 1$

Hypergeometric Distribution

x = **RAND**('HYPER',*N*,*R*,*n*)

Arguments

x

is an integer observation from the distribution with the following probability density function:

$$f(x) = \frac{\binom{R}{x}\binom{N - R}{n - x}}{\binom{N}{n}}$$

Range $x = \max(0, (n - (N - R))), \ldots, \min(n, R)$

N

is an integer population size parameter.

Range $N = 1, 2, ...$

R

is an integer number of items in the category of interest.

Range $R = 0, 1, ..., N$

n

is an integer sample size parameter.

Range $n = 1, 2, ..., N$

The hypergeometric distribution is a mathematical formalization of an experiment in which you draw n balls from an urn that contains N balls, R of which are red. The hypergeometric distribution is the distribution of the number of red balls in the sample of n.

Lognormal Distribution

x = **RAND**('LOGNORMAL')

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{e^{-\ln^2(x)/2}}{x\sqrt{2\pi}}$$

Range $x > 0$

Negative Binomial Distribution

x = **RAND**('NEGBINOMIAL',p,k)

Arguments

x

is an integer observation from the distribution with the following probability density function:

$$f(x) = \begin{cases} \binom{x+k-1}{k-1}(1-p)^x p^k & 0 < p < 1,\ x = 0, 1, ... \\ 1 & p = 1,\ x = 0 \end{cases}$$

Range $x = 0, 1, ...$

k

is an integer parameter that is the number of successes. However, non-integer k values are allowed as well.

Range $k = 1, 2, ...$

p

is a numeric probability of success.

Range $0 < p \leq 1$

The negative binomial distribution is the distribution of the number of failures before k successes occur in sequential independent trials, all with the same probability of success, p.

Normal Distribution

x = **RAND**('NORMAL',<,θ,λ>)

Arguments

x

is an observation from the normal distribution with a mean of θ and a standard deviation of λ that has the following probability density function:

$$f(x) = \frac{1}{\lambda\sqrt{2\pi}} \exp\left(-\frac{(x-\theta)^2}{2\lambda^2}\right)$$

Range $-\infty < x < \infty$

θ

is the mean parameter.

Default 0

λ

is the standard deviation parameter.

Default 1

Range $\lambda > 0$

Poisson Distribution

x = **RAND**('POISSON',m)

Arguments

x

is an integer observation from the distribution with the following probability density function:

$$f(x) = \frac{m^x e^{-m}}{x!}$$

Range $x = 0, 1, \ldots$

m

is a numeric mean parameter.

Range $m > 0$

T Distribution

x = **RAND**('T',df)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{\Gamma\left(\frac{df+1}{2}\right)}{\sqrt{df\,\pi}\,\Gamma\left(\frac{df}{2}\right)}\left(1+\frac{x^2}{df}\right)^{-\frac{df+1}{2}}$$

Range $-\infty < x < \infty$

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Tabled Distribution

x = **RAND**('TABLE',*p1*,*p2*, ...)

Arguments

x
is an integer observation from one of the following distributions:

If $\sum_{i=1}^{n} p_i < 1$, then *x* is an observation from this probability density function:

$$f(i) = p_i, \quad i = 1, 2, \ldots, n$$

and

$$f(n+1) = 1 - \sum_{i=1}^{n} p_i$$

If for some index $\sum_{i=1}^{n} p_i \geq 1$, then *x* is an observation from this probability density function:

$$f(i) = p_i, \quad i = 1, 2, \ldots, n-1$$

and

$$f(n) = 1 - \sum_{i=1}^{n-1} p_i$$

p1, p2, ...
are numeric probability values.

Range $0 \leq p1, p2, \ldots \leq 1$

Restriction The maximum number of probability parameters depends on your operating environment, but the maximum number of parameters is at least 32,767.

The tabled distribution takes on the values 1, 2, ..., *n* with specified probabilities.

Note: By using the FORMAT statement, you can map the set {1, 2, ..., *n*} to any set of *n* or fewer elements.

Triangular Distribution

x = **RAND**('TRIANGLE',*h*)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \begin{cases} \frac{2x}{h} & 0 \le x \le h \\ \frac{2(1 - x)}{1 - h} & h < x \le 1 \end{cases}$$

In this equation, $0 \le h \le 1$.

Range $0 \le x \le 1$

Note The distribution can be easily shifted and scaled.

h

is the horizontal location of the peak of the triangle.

Range $0 \le h \le 1$

Uniform Distribution

x = **RAND**('UNIFORM')

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = 1$$

Range $0 < x < 1$

The uniform random number generator that the RAND function uses is the Mersenne-Twister (Matsumoto and Nishimura 1998). This generator has a period of $2^{19937} - 1$ and 623-dimensional equidistribution up to 32-bit accuracy. This algorithm underlies the generators for the other available distributions in the RAND function.

Weibull Distribution

x = **RAND**('WEIBULL',a,b)

Arguments

x

is an observation from the distribution with the following probability density function:

$$f(x) = \frac{a}{b^a} x^{a-1} e^{-\left(\frac{x}{b}\right)^a}$$

Range $x \ge 0$

a

is a numeric shape parameter.

Range $a > 0$

b
is a numeric scale parameter.

Range $b > 0$

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=rand('BERN',.75);`	`0`
`x=rand('BETA',3,0.1);`	`.99920`
`x=rand('BINOM',0.75,10);`	`10`
`x=rand('CAUCHY');`	`-1.41525`
`x=rand('CHISQ',22);`	`25.8526`
`x=rand('ERLANG', 7);`	`7.67039`
`x=rand('EXPO');`	`1.48847`
`x=rand('F',12,322);`	`1.99647`
`x=rand('GAMMA',7.25);`	`6.59588`
`x=rand('GEOM',0.02);`	`43`
`x=rand('HYPER',10,3,5);`	`1`
`x=rand('LOGN');`	`0.66522`
`x=rand('NEGB',0.8,5);`	`33`
`x=rand('NORMAL');`	`1.03507`
`x=rand('POISSON',6.1);`	`6`
`x=rand('T',4);`	`2.44646`
`x=rand('TABLE',.2,.5);`	`2`
`x=rand('TRIANGLE',0.7);`	`.63811`
`x=rand('UNIFORM');`	`.96234`
`x=rand('WEIB',0.25,2.1);`	`6.55778`

See Also

CALL Routines:

- "CALL STREAMINIT Routine" on page 254

References

Fishman, G. S. 1996. *Monte Carlo: Concepts, Algorithms, and Applications*. New York, USA: Springer-Verlag.

Fushimi, M., and S. Tezuka. "The k-Distribution of Generalized Feedback Shift Register Pseudorandom Numbers." 1983. *Communications of the ACM* 26: 516–523.

Gentle, J. E. 1998. *Random Number Generation and Monte Carlo Methods*. New York, USA: Springer-Verlag.

Lewis, T. G., and W. H. Payne. "Generalized Feedback Shift Register Pseudorandom Number Algorithm." 1973. *Journal of the ACM* 20: 456-468.

Matsumoto, M., and Y. Kurita. "Twisted GFSR Generators." 1992. *ACM Transactions on Modeling and Computer Simulation* 2: 179–194.

Matsumoto, M., and Y. Kurita. "Twisted GFSR Generators II." 1994. *ACM Transactions on Modeling and Computer Simulation* 4: 254–266.

Matsumoto, M., and T. Nishimura. "Mersenne Twister: A 623–Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." 1998. *ACM Transactions on Modeling and Computer Simulation* 8: 3–30.

Ripley, B. D. 1987. *Stochastic Simulation*. New York, USA: Wiley.

Robert, C. P., and G. Casella. 1999. *Monte Carlo Statistical Methods*. New York, USA: Springer-Verlag.

Ross, S. M. 1997. *Simulation*. San Diego, USA: Academic Press.

RANEXP Function

Returns a random variate from an exponential distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANEXP routine instead of the RANEXP function.

Syntax

RANEXP(*seed*)

Required Argument

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Seed Values" on page 11 for more information about seed values

Details

The RANEXP function returns a variate that is generated from an exponential distribution with parameter 1. An inverse transform method applied to a RANUNI uniform variate is used.

An exponential variate X with parameter LAMBDA can be generated:

```
x=ranexp(seed)/lambda;
```

An extreme value variate X with location parameter ALPHA and scale parameter BETA can be generated:

```
x=alpha-beta*log(ranexp(seed));
```

A geometric variate X with parameter P can be generated as follows:

```
x=floor(-ranexp(seed)/log(1-p));
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "Generating Multiple Variables from One Seed in Random-Number Functions" on page 22 .

Comparisons

The CALL RANEXP routine, an alternative to the RANEXP function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- "CALL RANEXP Routine" on page 217

RANGAM Function

Returns a random variate from a gamma distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANGAM routine instead of the RANGAM function.

Syntax

RANGAM(*seed*,*a*)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Seed Values" on page 11 for more information about seed values

a

is a numeric constant, variable, or expression that specifies the shape parameter.

Range $a > 0$

Details

The RANGAM function returns a variate that is generated from a gamma distribution with parameter *a*. For $a > 1$, an acceptance-rejection method due to Cheng (1977) is used. (See "References" on page 1001). For $a \leq 1$, an acceptance-rejection method due to Fishman is used (1978, Algorithm G2) (See "References" on page 1001).

A gamma variate X with shape parameter ALPHA and scale BETA can be generated:

```
x=beta*rangam(seed,alpha);
```

If 2*ALPHA is an integer, a chi-square variate X with 2*ALPHA degrees of freedom can be generated:

```
x=2*rangam(seed,alpha);
```

If N is a positive integer, an Erlang variate X can be generated:

```
x=beta*rangam(seed,N);
```

It has the distribution of the sum of N independent exponential variates whose means are BETA.

And finally, a beta variate X with parameters ALPHA and BETA can be generated:

```
y1=rangam(seed,alpha);
y2=rangam(seed,beta);
x=y1/(y1+y2);
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "Generating Multiple Variables from One Seed in Random-Number Functions" on page 22.

Comparisons

The CALL RANGAM routine, an alternative to the RANGAM function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- "CALL RANGAM Routine" on page 219

RANGE Function

Returns the range of the nonmissing values.

Category: Descriptive Statistics

Syntax

RANGE(*argument-1*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one nonmissing argument is required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Details

The RANGE function returns the difference between the largest and the smallest of the nonmissing arguments.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x0=range(.,.);`	`.`
`x1=range(-2,6,3);`	`8`
`x2=range(2,6,3,.);`	`4`
`x3=range(1,6,3,1);`	`5`
`x4=range(of x1-x3);`	`4`

RANK Function

Returns the position of a character in the ASCII or EBCDIC collating sequence.

Category: Character

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

See: “RANK Function: Windows” in *SAS Companion for Windows*
“RANK Function: UNIX” in *SAS Companion for UNIX Environments*

Syntax

RANK(*x*)

Required Argument

x
specifies a character constant, variable, or expression.

Details

The RANK function returns an integer that represents the position of the first character in the character expression. The result depends on your operating environment.

Example

The following SAS statements produce these results.

SAS Statement	Result	
	ASCII	EBCDIC
`n=rank('A');` `put n;`	`65`	`193`

See Also

Functions:

- "BYTE Function" on page 149
- "COLLATE Function" on page 308

RANNOR Function

Returns a random variate from a normal distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANNOR routine instead of the RANNOR function.

Syntax

RANNOR(*seed*)

Required Argument

seed
is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Seed Values" on page 11 for more information about seed values

Details

The RANNOR function returns a variate that is generated from a normal distribution with mean 0 and variance 1. The Box-Muller transformation of RANUNI uniform variates is used.

A normal variate X with mean MU and variance S2 can be generated with this code:

```
x=MU+sqrt(S2)*rannor(seed);
```

A lognormal variate X with mean *exp*(MU + S2/2) and variance *exp*(2*MU + 2*S2) −*exp*(2*MU + S2) can be generated with this code:

```
x=exp(MU+sqrt(S2)*rannor(seed));
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "Generating Multiple Variables from One Seed in Random-Number Functions" on page 22.

Comparisons

The CALL RANNOR routine, an alternative to the RANNOR function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- "CALL RANNOR Routine" on page 222

RANPOI Function

Returns a random variate from a Poisson distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANPOI routine instead of the RANPOI function.

Syntax

RANPOI(*seed*,*m*)

Required Arguments

seed
is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Seed Values" on page 11 for more information about seed values.

m
is a numeric constant, variable, or expression that specifies the mean of the distribution.

Range $m \geq 0$

Details

The RANPOI function returns a variate that is generated from a Poisson distribution with mean *m*. For $m < 85$, an inverse transform method applied to a RANUNI uniform variate is used (Fishman 1976) (See "References" on page 1001). For $m \geq 85$, the normal approximation of a Poisson random variable is used. To expedite execution, internal variables are calculated only on initial calls (that is, with each new *m*).

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see "Generating Multiple Variables from One Seed in Random-Number Functions" on page 22.

Comparisons

The CALL RANPOI routine, an alternative to the RANPOI function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- "CALL RANPOI Routine" on page 228

RANTBL Function

Returns a random variate from a tabled probability distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANTBL routine instead of the RANTBL function.

Syntax

RANTBL(*seed*,*p1*,... p_i... ,p_n)

Required Arguments

seed
is a numeric constant, variable, or expression with an integer value. If $seed \leq 0$, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Seed Values" on page 11 for more information about seed values.

***p**i*

is a numeric constant, variable, or expression.

Range $0 \le p_i \le 1$ for $0 < i \le n$

Details

The RANTBL function returns a variate that is generated from the probability mass function defined by p_1 through p_n. An inverse transform method applied to a RANUNI uniform variate is used. RANTBL returns

1 *with probability* p_1

2 *with probability* p_2

.

.

.

n *with probability* p_n

$n+1$ *with probability* $1 - \sum_{i=1}^{n} p_i$ *if* $\sum_{i=1}^{n} p_i \le 1$

If, for some index $j<n$, $\sum_{i=1}^{j} p_i \ge 1$, RANTBL returns only the indices 1 through *j* with the probability of occurrence of the index *j* equal to $1 - \sum_{i=1}^{j-1} p_i$.

Let n=3 and P1, P2, and P3 be three probabilities with P1+P2+P3=1, and M1, M2, and M3 be three variables. The variable X in these statements

```
array m{3} m1-m3;
x=m{rantbl(seed,of p1-p3)};
```

will be assigned one of the values of M1, M2, or M3 with probabilities of occurrence P1, P2, and P3, respectively.

For a discussion and example of an effective use of the random number CALL routines, see "Starting, Stopping, and Restarting a Stream" on page 26.

Comparisons

The CALL RANTBL routine, an alternative to the RANTBL function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- “CALL RANTBL Routine” on page 230

RANTRI Function

Returns a random variate from a triangular distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANTRI routine instead of the RANTRI function.

Syntax

RANTRI(*seed*,*h*)

Required Arguments

seed

is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See for more information about seed values.“Seed Values” on page 11

h

is a numeric constant, variable, or expression that specifies the mode of the distribution.

Range $0 < h < 1$

Details

The RANTRI function returns a variate that is generated from the triangular distribution on the interval (0,1) with parameter *h*, which is the modal value of the distribution. An inverse transform method applied to a RANUNI uniform variate is used.

A triangular distribution X on the interval (A,B) with mode C, where A ≤ C ≤ B, can be generated:

```
x=(b-a)*rantri(seed,(c-a)/(b-a))+a;
```

For a discussion about seeds and streams of data, as well as examples of using the random-number functions, see “Generating Multiple Variables from One Seed in Random-Number Functions” on page 22.

Comparisons

The CALL RANTRI routine, an alternative to the RANTRI function, gives greater control of the seed and random number streams.

See Also

Functions:

- “RAND Function” on page 806

CALL Routines:

- "CALL RANTRI Routine" on page 233

RANUNI Function

Returns a random variate from a uniform distribution.

Category: Random Number

Tip: If you want to change the seed value during execution, you must use the CALL RANUNI routine instead of the RANUNI function.

Syntax

RANUNI(*seed*)

Required Argument

seed
is a numeric constant, variable, or expression with an integer value. If *seed* ≤ 0, the time of day is used to initialize the seed stream.

Range $seed < 2^{31}-1$

See "Seed Values" on page 11 for more information about seed values

Details

The RANUNI function returns a number that is generated from the uniform distribution on the interval (0,1) using a prime modulus multiplicative generator with modulus $2^{31}-$ and multiplier 397204094 (Fishman and Moore 1982) (See "References" on page 1001).

You can use a multiplier to change the length of the interval and an added constant to move the interval. For example,

```
random_variate=a*ranuni(seed)+b;
```

returns a number that is generated from the uniform distribution on the interval (b,a+b).

Comparisons

The CALL RANUNI routine, an alternative to the RANUNI function, gives greater control of the seed and random number streams.

See Also

Functions:

- "RAND Function" on page 806

CALL Routines:

- "CALL RANUNI Routine" on page 235

RENAME Function

Renames a member of a SAS library, an entry in a SAS catalog, an external file, or a directory.

Categories: External Files
SAS File I/O

Syntax

RENAME(*old-name*, *new-name* <, *type*<, *description* <, *password* <, *generation*> > > >)

Required Arguments

old-name
specifies a character constant, variable, or expression that is the current name of a member of a SAS library, an entry in a SAS catalog, an external file, or an external directory.

For a data set, *old-name* can be a one-level or two-level name. For a catalog entry, *old-name* can be a one-level, two-level, or four-level name. For an external file or directory, *old-name* must be the full pathname of the file or the directory. If the value for *old-name* is not specified, then SAS uses the current directory.

new-name
specifies a character constant, variable, or expression that is the new one-level name for the library member, catalog entry, external file, or directory.

Optional Arguments

type
is a character constant, variable, or expression that specifies the type of element to rename. *Type* can be a null argument, or one of the following values:

ACCESS	specifies a SAS/ACCESS descriptor that was created using SAS/ACCESS software.
CATALOG	specifies a SAS catalog or catalog entry.
DATA	specifies a SAS data set.
VIEW	specifies a SAS data set view.
FILE	specifies an external file or directory.

Default 'DATA'

description
specifies a character constant, variable, or expression that is the description of a catalog entry. You can specify *description* only when the value of *type* is CATALOG. *Description* can be a null argument.

password
is a character constant, variable, or expression that specifies the password for the data set that is being renamed. *Password* can be a null argument.

generation
is a numeric constant, variable, or expression that specifies the generation number of the data set that is being renamed. *Generation* can be a null argument.

Details

You can use the RENAME function to rename members of a SAS library or entries in a SAS catalog. SAS returns 0 if the operation was successful, and a value other than 0 if the operation was not successful.

To rename an entry in a catalog, specify the four-level name for *old-name* and a one-level name for *new-name*. You must specify CATALOG for *type* when you rename an entry in a catalog.

Operating Environment Information
Use RENAME in directory-based operating environments only. If you use RENAME in a mainframe operating environment, SAS generates an error.

Examples

Example 1: Renaming Data Sets and Catalog Entries

The following examples rename a SAS data set from DATA1 to DATA2, and also rename a catalog entry from A.SCL to B.SCL.

```
rc1=rename('mylib.data1', 'data2');
rc2=rename('mylib.mycat.a.scl', 'b', 'catalog');
```

Example 2: Renaming an External File

The following examples rename external files.

```
   /* Rename a file that is located in another directory. */
rc=rename('/local/u/testdir/first',
          '/local/u/second', 'file');
   /* Rename a PC file. */
rc=rename('d:\temp', 'd:\testfile', 'file');
```

Example 3: Renaming a Directory

The following example renames a directory in the UNIX operating environment.

```
rc=rename('/local/u/testdir/', '/local/u/oldtestdir', 'file');
```

Example 4: Renaming a Generation Data Set

The following example renames the generation data set WORK.ONE to WORK.TWO, where the password for WORK.ONE#003 is *my-password*.

```
rc=rename('work.one','two',,,3,'my-password');
```

See Also

Functions:

- “FDELETE Function” on page 404
- “FILEEXIST Function” on page 410
- “EXIST Function” on page 396

REPEAT Function

Returns a character value that consists of the first argument repeated n+1 times.

Category: Character

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

Syntax

REPEAT(*argument*,*n*)

Required Arguments

argument
specifies a character constant, variable, or expression.

n
specifies the number of times to repeat *argument*.

Restriction *n* must be greater than or equal to 0.

Details

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

The REPEAT function returns a character value consisting of the first argument repeated *n* times. Thus, the first argument appears *n*+1 times in the result.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=repeat('ONE',2);` `put x;`	`ONEONEONE`

RESOLVE Function

Returns the resolved value of the argument after it has been processed by the macro facility.

Category: Macro

Syntax

RESOLVE(*argument*)

Required Argument

argument
is a character constant, variable, or expression with a value that is a macro expression.

Details

If the RESOLVE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

RESOLVE is fully documented in *SAS Macro Language: Reference*.

See Also

Functions:

- "SYMGET Function" on page 901

REVERSE Function

Reverses a character string.

Category: Character

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

Tip: DBCS equivalent function is KREVERSE in *SAS National Language Support (NLS): Reference Guide*.

Syntax

REVERSE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the REVERSE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The last character in the argument becomes the first character in the result, the next-to-last character in the argument becomes the second character in the result, and so on.

Note: Trailing blanks in the argument become leading blanks in the result.

Example

The following SAS statements produce this result.

SAS Statement	Result
	----+----1
backward=reverse('xyz '); put backward $5.;	zyx

REWIND Function

Positions the data set pointer at the beginning of a SAS data set.

Category: SAS File I/O

Syntax

REWIND(*data-set-id*)

Required Argument

data-set-id
is a numeric variable that specifies the data set identifier that the OPEN function returns.

Restriction The data set cannot be opened in IS mode.

Details

REWIND returns 0 if the operation was successful, ≠0 if it was not successful. After a call to REWIND, a call to FETCH reads the first observation in the data set.

If there is an active WHERE clause, REWIND moves the data set pointer to the first observation that satisfies the WHERE condition.

Example

This example calls FETCHOBS to fetch the tenth observation in the data set MYDATA. Next, the example calls REWIND to return to the first observation and fetch the first observation.

```
%let dsid=%sysfunc(open(mydata,i));
%let rc=%sysfunc(fetchobs(&dsid,10));
%let rc=%sysfunc(rewind(&dsid));
%let rc=%sysfunc(fetch(&dsid));
```

See Also

Functions:

- “FETCH Function” on page 405
- “FETCHOBS Function” on page 406
- “FREWIND Function” on page 496

- "NOTE Function" on page 692
- "OPEN Function" on page 716
- "POINT Function" on page 746

RIGHT Function

Right aligns a character expression.

Category: Character

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

Tip: DBCS equivalent function is KRIGHT.

Syntax

RIGHT(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the RIGHT function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The RIGHT function returns an argument with trailing blanks moved to the start of the value. The length of the result is the same as the length of the argument.

Example

The following SAS statements produce these results.

SAS Statement	Result
	`----+----1----+`

`a='Due Date  ';`
`b=right(a);`
`put a $10.;`
`put b $10.;`
	`Due Date`
	`  Due Date`

See Also

Functions:

- "COMPRESS Function" on page 327
- "LEFT Function" on page 616

- "TRIM Function" on page 924

RMS Function

Returns the root mean square of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

RMS(*argument*<,*argument*,…>)

Required Argument

argument
is a numeric constant, variable, or expression.

Tip The argument list can consist of a variable list, which is preceded by OF.

Details

The root mean square is the square root of the arithmetic mean of the squares of the values. If all the arguments are missing values, then the result is a missing value. Otherwise, the result is the root mean square of the non-missing values.

Let n be the number of arguments with non-missing values, and let $x_1, x_2, \ldots, x_n$ be the values of those arguments. The root mean square is

$$\sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_n^2}{n}}$$

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=rms(1,7);`	`5`
`x2=rms(.,1,5,11);`	`7`
`x3=rms(of x1-x2);`	`6.0827625303`

ROUND Function

Rounds the first argument to the nearest multiple of the second argument, or to the nearest integer when the second argument is omitted.

Category: Truncation

Syntax

ROUND (*argument* <,*rounding-unit*>)

Required Argument

argument
is a numeric constant, variable, or expression to be rounded.

Optional Argument

rounding-unit
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details

Basic Concepts

The ROUND function rounds the first argument to a value that is very close to a multiple of the second argument. The result might not be an exact multiple of the second argument.

Differences between Binary and Decimal Arithmetic

Computers use binary arithmetic with finite precision. If you work with numbers that do not have an exact binary representation, computers often produce results that differ slightly from the results that are produced with decimal arithmetic.

For example, the decimal values 0.1 and 0.3 do not have exact binary representations. In decimal arithmetic, 3*0.1 is exactly equal to 0.3, but this equality is not true in binary arithmetic. As the following example shows, if you write these two values in SAS, they appear the same. If you compute the difference, however, you can see that the values are different.

```
data _null_;
   point_three=0.3;
   three_times_point_one=3*0.1;
   difference=point_three - three_times_point_one;
   put point_three= ;
   put three_times_point_one= ;
   put difference= ;
run;
```

The following lines are written to the SAS log:

```
   point_three= 0.3
   three_times_point_one= 0.3
   difference= -5.55112E-17
```

Operating Environment Information
The example above was executed in a z/OS 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 SAS program, the value is computed by the standard informat as 3/10. To be consistent with the standard informat, **ROUND(0.33,0.1)** computes the result as 3/10, and the following statement produces the results that you would expect.

```
if round(x,0.1) = 0.3 then
   ... more SAS statements ...
```

However, if you use the variable Y instead of the constant 0.3, as the following statement shows, the results might be unexpected depending on how the variable Y is computed.

```
if round(x,0.1) = y then
   ... more SAS statements ...
```

If SAS reads Y as the characters "0.3" using the standard informat, the result is the same as if a constant 0.3 appeared in the IF statement. If SAS reads Y with a different informat, or if a program other than SAS reads Y, then there is no guarantee that the characters "0.3" would produce a value of exactly 3/10. Imprecision can also be caused by computation involving numbers that do not have exact binary representations, or by porting data sets from one operating environment to another that has a different floating-point representation.

If you know that Y is a decimal number with one decimal place, but are not certain that Y has exactly the same value as would be produced by the standard informat, it is better to use the following statement:

```
if round(x,0.1) = round(y,0.1) then
   ... more SAS statements ...
```

Testing for Approximate Equality

You should not use the ROUND function as a general method to test for approximate equality. Two numbers that differ only in the least significant bit can round to different values if one number rounds down and the other number rounds up. Testing for approximate equality depends on how the numbers have been computed. If both numbers are computed to high relative precision, you could test for approximate equality by using the ABS and the MAX functions, as the following example shows.

```
if abs(x-y) <= 1e-12 * max( abs(x), abs(y) ) then
   ... more SAS statements ...
```

Producing Expected Results

In general, **ROUND(argument, rounding-unit)** produces the result that you expect from decimal arithmetic if the result has no more than nine significant digits and any of the following conditions are true:

- The rounding unit is an integer.
- The rounding unit is a power of 10 greater than or equal to 1e-15. (If the rounding unit is less than one, ROUND treats it as a power of 10 if the reciprocal of the rounding unit differs from a power of 10 in at most the three or four least significant bits.)

- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```
options pageno=1 nodate;

data rounding;
   d1 = round(1234.56789,100)     - 1200;
   d2 = round(1234.56789,10)      - 1230;
   d3 = round(1234.56789,1)       - 1235;
   d4 = round(1234.56789,.1)      - 1234.6;
   d5 = round(1234.56789,.01)     - 1234.57;
   d6 = round(1234.56789,.001)    - 1234.568;
   d7 = round(1234.56789,.0001)   - 1234.5679;
   d8 = round(1234.56789,.00001)  - 1234.56789;
   d9 = round(1234.56789,.1111)   - 1234.5432;
      /* d10 has too many decimal places in the value for */
      /* rounding-unit.                                   */
   d10 = round(1234.56789,.11111) - 1234.54321;
run;
proc print data=rounding noobs;
run;
```

Display 2.54 *Results of Rounding Based on the Value of the Rounding Unit*

The SAS System 1

d1	d2	d3	d4	d5	d6	d7	d8	d9	d10
0	0	0	0	0	0	0	0	0	-1.7053E-13

Operating Environment Information

The example above was executed in a z/OS environment. If you use other operating environments, the results will be slightly different.

When the Rounding Unit Is the Reciprocal of an Integer

When the rounding unit is the reciprocal of an integer, the ROUND function computes the result by dividing by the integer. (ROUND treats the rounding unit as a reciprocal of an integer if the reciprocal of the rounding unit differs from an integer in at most the three or four least significant bits.) Therefore, you can safely compare the result from ROUND with the ratio of two integers, but not with a multiple of the rounding unit. For example:

```
options pageno=1 nodate;

data rounding2;
   drop pi unit;
   pi = arcos(-1);
   unit=1/7;
   d1=round(pi,unit) - 22/7;
   d2=round(pi, unit) - 22*unit;
run;
proc print data=rounding2 noobs;
```

```
run;
```

Display 2.55 *Results of Rounding by the Reciprocal of an Integer*

```
The SAS System                                    1

d1          d2

 0     2.2204E-16
```

Operating Environment Information
The example above was executed in an z/OS environment. If you use other operating environments, the results will be slightly different.

Computing Results in Special Cases

The ROUND function computes the result by multiplying an integer by the rounding unit when all of the following conditions are true:

- The rounding unit is not an integer.
- The rounding unit is not a power of 10.
- The rounding unit is not the reciprocal of an integer.
- The result that you expect from decimal arithmetic has no more than four decimal places.

For example:

```
data _null_;
   difference=round(1234.56789,.11111) - 11111*.11111;
   put difference=;
run;
```

The following line is written to the SAS log:

```
   difference=0
```

Operating Environment Information
The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

Computing Results When the Value Is Halfway between Multiples of the Rounding Unit

When the value to be rounded is approximately halfway between two multiples of the rounding unit, the ROUND function rounds up the absolute value and restores the original sign. For example:

```
data test;
   do i=8 to 17;
      value=0.5 - 10**(-i);
      round=round(value);
      output;
   end;
   do i=8 to 17;
      value=-0.5 + 10**(-i);
      round=round(value);
      output;
```

```
   end;
run;
proc print data=test noobs;
   format value 19.16;
run;
```

Display 2.56 *Results of Rounding When Values Are Halfway between Multiples of the Rounding Unit*

The SAS System

i	value	round
8	0.4999999900000000	0
9	0.4999999990000000	0
10	0.4999999999000000	0
11	0.4999999999900000	0
12	0.4999999999990000	0
13	0.4999999999999000	1
14	0.4999999999999900	1
15	0.4999999999999900	1
16	0.5000000000000000	1
17	0.5000000000000000	1
8	-0.4999999900000000	0
9	-0.4999999990000000	0
10	-0.4999999999000000	0
11	-0.4999999999900000	0
12	-0.4999999999990000	0
13	-0.4999999999999000	-1
14	-0.4999999999999900	-1
15	-0.4999999999999900	-1
16	-0.5000000000000000	-1
17	-0.5000000000000000	-1

Operating Environment Information

The example above was executed in a z/OS environment. If you use other operating environments, the results might be slightly different.

The approximation is relative to the size of the value to be rounded, and is computed in a manner that is shown in the following DATA step. This DATA step code will not always produce results exactly equivalent to the ROUND function.

```
data testfile;
   do i = 1 to 17;
      value = 0.5 - 10**(-i);
      epsilon = min(1e-6, value * 1e-12);
      temp = value + .5 + epsilon;
      fraction = modz(temp, 1);
      round = temp - fraction;
      output;
   end;
run;
proc print data=testfile noobs;
   format value 19.16;
```

```
run;
```

Comparisons

The ROUND, ROUNDE, and ROUNDZ functions are similar with four exceptions:

- ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDE returns an even multiple when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument.
- When the rounding unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUND and ROUNDE perform extra computations, called fuzzing, to try to make the result agree with decimal arithmetic in the most common situations. ROUNDZ does not fuzz the result.

Example

The following example compares the results that are returned by the ROUND function with the results that are returned by the ROUNDE function. The output was generated from the UNIX operating environment.

```
data results;
   do x=0 to 4 by .25;
      Rounde=rounde(x);
      Round=round(x);
      output;
   end;
run;
proc print data=results noobs;
run;
```

Display 2.57 *Results That Are Returned by the ROUND and ROUNDE Functions*

The SAS System

x	Rounde	Round
0.00	0	0
0.25	0	0
0.50	0	1
0.75	1	1
1.00	1	1
1.25	1	1
1.50	2	2
1.75	2	2
2.00	2	2
2.25	2	2
2.50	2	3
2.75	3	3
3.00	3	3
3.25	3	3
3.50	4	4
3.75	4	4
4.00	4	4

See Also

Functions:

- "CEIL Function" on page 294
- "CEILZ Function" on page 296
- "FLOOR Function" on page 480
- "FLOORZ Function" on page 481
- "INT Function" on page 555
- "INTZ Function" on page 596
- "ROUNDE Function" on page 840
- "ROUNDZ Function" on page 843

ROUNDE Function

Rounds the first argument to the nearest multiple of the second argument, and returns an even multiple when the first argument is halfway between the two nearest multiples.

Category: Truncation

Syntax

ROUNDE (*argument* <,*rounding-unit*>)

Required Argument

argument
is a numeric constant, variable, or expression to be rounded.

Optional Argument

rounding-unit
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details

The ROUNDE function rounds the first argument to the nearest multiple of the second argument. If you omit the second argument, ROUNDE uses a default value of 1 for *rounding-unit*.

Comparisons

The ROUND, ROUNDE, and ROUNDZ functions are similar with four exceptions:

- ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDE returns an even multiple when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument.
- When the rounding unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUND and ROUNDE perform extra computations, called fuzzing, to try to make the result agree with decimal arithmetic in the most common situations. ROUNDZ does not fuzz the result.

Example

The following example compares the results that are returned by the ROUNDE function with the results that are returned by the ROUND function.

```
data results;
   do x=0 to 4 by .25;
      Rounde=rounde(x);
      Round=round(x);
      output;
   end;
run;
proc print data=results noobs;
run;
```

Display 2.58 *Results That are Returned by the ROUNDE and ROUND Functions*

The SAS System

x	Rounde	Round
0.00	0	0
0.25	0	0
0.50	0	1
0.75	1	1
1.00	1	1
1.25	1	1
1.50	2	2
1.75	2	2
2.00	2	2
2.25	2	2
2.50	2	3
2.75	3	3
3.00	3	3
3.25	3	3
3.50	4	4
3.75	4	4
4.00	4	4

See Also

Functions:

- “CEIL Function” on page 294
- “CEILZ Function” on page 296
- “FLOOR Function” on page 480
- “FLOORZ Function” on page 481
- “INT Function” on page 555

- "INTZ Function" on page 596
- "ROUND Function" on page 833
- "ROUNDZ Function" on page 843

ROUNDZ Function

Rounds the first argument to the nearest multiple of the second argument, using zero fuzzing.

Category: Truncation

Syntax

ROUNDZ (*argument* <,*rounding-unit*>)

Required Argument

argument
is a numeric constant, variable, or expression to be rounded.

Optional Argument

rounding-unit
is a positive, numeric constant, variable, or expression that specifies the rounding unit.

Details

The ROUNDZ function rounds the first argument to the nearest multiple of the second argument. If you omit the second argument, ROUNDZ uses a default value of 1 for *rounding-unit*.

Comparisons

The ROUND, ROUNDE, and ROUNDZ functions are similar with four exceptions:

- ROUND returns the multiple with the larger absolute value when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDE returns an even multiple when the first argument is approximately halfway between the two nearest multiples of the second argument.
- ROUNDZ returns an even multiple when the first argument is exactly halfway between the two nearest multiples of the second argument.
- When the rounding unit is less than one and not the reciprocal of an integer, the result that is returned by ROUNDZ might not agree exactly with the result from decimal arithmetic. ROUND and ROUNDE perform extra computations, called fuzzing, to try to make the result agree with decimal arithmetic in the most common situations. ROUNDZ does not fuzz the result.

Examples

Example 1: Comparing Results from the ROUNDZ and ROUND Functions

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

```
data test;
   do i=10 to 17;
      Value=3.5 - 10**(-i);
      Roundz=roundz(value);
      Round=round(value);
      output;
   end;
   do i=16 to 12 by -1;
      value=3.5 + 10**(-i);
      roundz=roundz(value);
      round=round(value);
      output;
   end;
run;
proc print data=test noobs;
   format value 19.16;
run;
```

Display 2.59 *Results That Are Returned by the ROUNDZ and ROUND Functions*

The SAS System

i	Value	Roundz	Round
10	3.4999999999000000	3	3
11	3.4999999999900000	3	3
12	3.4999999999990000	3	4
13	3.4999999999999000	3	4
14	3.4999999999999900	3	4
15	3.5000000000000000	3	4
16	3.5000000000000000	4	4
17	3.5000000000000000	4	4
16	3.5000000000000000	4	4
15	3.5000000000000000	4	4
14	3.5000000000000100	4	4
13	3.5000000000001000	4	4
12	3.5000000000010000	4	4

Example 2: Sample Output from the ROUNDZ Function

These examples show the results that are returned by ROUNDZ.

Table 2.2 *Results Using ROUNDZ*

SAS Statement	Result

`var1=223.456;`
`x=roundz(var1,1);`
`put x 9.5;`
	`223.00000`

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

`x=roundz(223.456,100);` `put x 9.5;`	`200.00000`
`x=roundz(223.456);` `put x 9.5;`	`223.00000`
`x=roundz(223.456,.3);` `put x 9.5;`	`223.50000`

See Also

Functions:

- "ROUND Function" on page 833
- "ROUNDE Function" on page 840

SAVING Function

Returns the future value of a periodic saving.

Category: Financial

Syntax

SAVING(*f*,*p*,*r*,*n*)

Required Arguments

f

is numeric, the future amount (at the end of *n* periods).

Range $f \geq 0$

p

is numeric, the fixed periodic payment.

Range $p \geq 0$

r
is numeric, the periodic interest rate expressed as a decimal.

Range $r \geq 0$

n
is an integer, the number of compounding periods.

Range $n \geq 0$

Details

The SAVING function returns the missing argument in the list of four arguments from a periodic saving. The arguments are related by

$$f = \frac{p(1 + r)\left((1 + r)^n - 1\right)}{r}$$

One missing argument must be provided. It is then calculated from the remaining three. No adjustment is made to convert the results to round numbers.

Example

A savings account pays a 5 percent nominal annual interest rate, compounded monthly. For a monthly deposit of $100, the number of payments that are needed to accumulate at least $12,000, can be expressed as

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

The value returned is 97.18 months. The fourth argument is set to missing, which indicates that the number of payments is to be calculated. The 5 percent 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.

Category: Financial

Syntax

SAVINGS(*base-date*, *initial-deposit-date*, *deposit-amount*, *deposit-number*, *deposit-interval*, *compounding-interval*, *date-1*, *rate-2* <,*date-2*, *rate-2*,...>)

Required Arguments

base-date
is a SAS date. The value that is returned is the balance of the savings at *base-date*.

initial-deposit-date
is a SAS date. *initial-deposit-date* is the date of the first deposit. Subsequent deposits are at the beginning of subsequent deposit intervals.

deposit-amount
is numeric. All deposits are assumed constant. *deposit-amount* is the value of each deposit.

deposit-number
is a positive integer. *deposit-number* is the number of deposits.

deposit-interval
is a SAS interval. *deposit-interval* is the frequency at which deposits are made.

compounding-interval
is a SAS interval. *compounding-interval* is the compounding interval.

date
is a SAS date. Each date is paired with a rate. *date* is the time that *rate* takes effect.

rate
is a numeric percentage. Each rate is paired with a date. *rate* is the interest rate that starts on *date*.

Details

The following details apply to the SAVINGS function:

- The values for rates must be between –99 and 120.
- *deposit-interval* cannot be 'CONTINUOUS'.
- The list of date-rate pairs does not need to be in chronological order.
- When multiple rate changes occur on a single date, the SAVINGS function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.
- There must be a valid date-rate pair whose date is at or prior to both the *initial-deposit-date* and the *base-date*.

Example

- If you deposit $300 monthly for two years into an account that compounds quarterly at an annual rate of 4%, the balance of the account after five years can be expressed as follows:

```
amount_base1 = SAVINGS("01jan2005"d, "01jan2000"d, 300, 24,
                                     "MONTH", "QUARTER", "01jan2000"d, 4.00);
```

- If the interest rate increases by a quarter-point each year, then the balance of the account could be expressed as follows:

```
amount_base2 = SAVINGS("01jan2005"d, "01jan2000"d, 300, 24,
                                     "MONTH", "QUARTER", "01jan2000"d, 4.00,
                                     "01jan2001"d, 4.25, "01jan2002"d, 4.50,
                                     "01jan2003"d, 4.75, "01jan2004"d, 5.00);
```

- To determine the balance after one year of deposits, the following statement sets amount_base3 to the desired balance:

```
amount_base3 = SAVINGS("01jan2001"d, "01jan2000"d, 300, 24,
                                     "MONTH", "QUARTER", "01jan2000"d, 4);
```

The SAVINGS function ignores deposits after the base date, so the deposits after the reference date do not affect the value that is returned.

SCAN Function

Returns the *n*th word from a character string.

Category: Character

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

Tip: The DBCS equivalent function is The DBCS equivalent function is KSCAN.

Syntax

SCAN(*string*, *count*<,*charlist* <,*modifiers*> >)

Required Arguments

string
specifies a character constant, variable, or expression.

count
is a nonzero numeric constant, variable, or expression that has an integer value that specifies the number of the word in the character string that you want SCAN to select. For example, a value of 1 indicates the first word, a value of 2 indicates the second word, and so on. The following rules apply:

- If *count* is positive, SCAN counts words from left to right in the character string.
- If *count* is negative, SCAN counts words from right to left in the character string.

Optional Arguments

charlist
specifies an optional character expression that initializes a list of characters. This list determines which characters are used as the delimiters that separate words. The following rules apply:

- By default, all characters in *charlist* are used as delimiters.
- If you specify the K modifier in the *modifier* argument, then all characters that are *not* in *charlist* are used as delimiters.

Tip You can add more characters to *charlist* by using other modifiers.

modifier
specifies a character constant, a variable, or an expression in which each non-blank character modifies the action of the SCAN function. Blanks are ignored. You can use the following characters as modifiers:

a or A	adds alphabetic characters to the list of characters.
b or B	scans backward from right to left instead of from left to right, regardless of the sign of the *count* argument.
c or C	adds control characters to the list of characters.
d or D	adds digits to the list of characters.

f or F adds an underscore and English letters (that is, valid first characters in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

g or G adds graphic characters to the list of characters. Graphic characters are characters that, when printed, produce an image on paper.

h or H adds a horizontal tab to the list of characters.

i or I ignores the case of the characters.

k or K causes all characters that are not in the list of characters to be treated as delimiters. That is, if K is specified, then characters that are in the list of characters are kept in the returned value rather than being omitted because they are delimiters. If K is not specified, then all characters that are in the list of characters are treated as delimiters.

l or L adds lowercase letters to the list of characters.

m or M specifies that multiple consecutive delimiters, and delimiters at the beginning or end of the *string* argument, refer to words that have a length of zero. If the M modifier is not specified, then multiple consecutive delimiters are treated as one delimiter, and delimiters at the beginning or end of the *string* argument are ignored.

n or N adds digits, an underscore, and English letters (that is, the characters that can appear in a SAS variable name using VALIDVARNAME=V7) to the list of characters.

o or O processes the *charlist* and *modifier* arguments only once, rather than every time the SCAN function is called. Using the O modifier in the DATA step (excluding WHERE clauses), or in the SQL procedure can make SCAN run faster when you call it in a loop where the *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 of 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 blanks from the word that SCAN returns.If you specify both the Q and R modifiers, then 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, then use the TRIM function instead of the SCAN function with the T modifier.

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

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

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

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

Details

Definition of "Delimiter" and "Word"

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

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

In the SCAN function, "word" refers to a substring that has all of the following characteristics:

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

A word can have a length of 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 both 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, 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 ! $ % & ( ) * + , - . / ; < ¬ | ¢

If you use the *modifier* argument without specifying any characters as delimiters, then the only delimiters that will be 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 *charlist* argument. Modifiers do not add to the list of default modifiers.

The Length of the Result

In a DATA step, most variables have a fixed length. If the word returned by the SCAN function is assigned to a variable that has a fixed length greater than the length of the returned word, then the value of that variable will be padded with blanks. Macro variables have varying lengths and are not padded with blanks.

The maximum length of the word that is returned by the SCAN function depends on the environment from which it is called:

- In a DATA step, if the SCAN function returns a value to a variable that has not yet been given a length, then that variable is given a length of 200 characters. If you need the SCAN function to assign to a variable a word that is longer than 200 characters, then you should explicitly specify the length of that variable.

  If you use the SCAN function in an expression that contains operators or other functions, a word that is returned by the SCAN function can have a length of up to 32,767 characters, except in a WHERE clause. In that case, the maximum length is 200 characters.
- In the SQL procedure, or in a WHERE clause in any procedure, the maximum length of a word that is returned by the SCAN function is 200 characters.
- In the macro processor, the maximum length of a word that is returned by the SCAN function is 65,534 characters.

The minimum length of the word that is returned by the SCAN function depends on whether the M modifier is specified. See "Using the SCAN Function with the M Modifier" on page 851. See also "Using the SCAN Function without the M Modifier" on page 851.

Using the SCAN Function with the M Modifier

If you specify the M modifier, then 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, then the SCAN function returns a word with a length of zero if one of the following 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, then 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, then the SCAN function does the following:

- ignores delimiters at the beginning or end of the string
- treats two or more consecutive delimiters as if they were a single delimiter

If the string contains no characters other than delimiters, or if you specify a count that is greater in absolute value than the number of words in the string, then the SCAN function returns one of the following:

- 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.

Examples

Example 1: Finding the First and Last Words in a String

The following example scans a string for the first and last words. Note the following:

- A negative count instructs the SCAN function to scan from right to left.
- Leading and trailing delimiters are ignored because the M modifier is not used.
- In the last observation, all characters in the string are delimiters.

```
data firstlast;
   input String $60.;
   First_Word = scan(string, 1);
   Last_Word = scan(string, -1);
   datalines4;
Jack and Jill
& Bob & Carol & Ted & Alice &
Leonardo
! $ % & ( ) * + , - . / ;
;;;;
proc print data=firstlast;
run;
```

Display 2.60 *Results of Finding the First and Last Words in a String*

The SAS System

Obs	String	First_Word	Last_Word
1	Jack and Jill	Jack	Jill
2	& Bob & Carol & Ted & Alice &	Bob	Alice
3	Leonardo	Leonardo	Leonardo
4	! $ % & ( ) * + , - . / ;		

Example 2: Finding All Words in a String without Using the M Modifier

The following example scans a string from left to right until the word that is returned is blank. Because the M modifier is not used, the SCAN function does not return any words that have a length of zero. Because blanks are included among the default delimiters, the SCAN function returns a blank word only when the count exceeds the number of words in the string. Therefore, the loop can be stopped when SCAN returns a blank word.

```
data all;
   length word $20;
   drop string;
   string = ' The quick brown fox jumps over the lazy dog.   ';
   do until(word=' ');
      count+1;
      word = scan(string, count);
      output;
```

```
   end;
run;
proc print data=all noobs;
run;
```

Display 2.61 *Results of Finding All Words without Using the M Modifier*

The SAS System

word	count
The	1
quick	2
brown	3
fox	4
jumps	5
over	6
the	7
lazy	8
dog	9
	10

Example 3: Finding All Words in a String by Using the M and O Modifiers

The following example shows the results of using the M modifier with a comma as a delimiter. With the M modifier, leading, trailing, and multiple consecutive delimiters cause the SCAN function to return words that have a length of zero. Therefore, you should not end the loop by testing for a blank word. Instead, you can use the COUNTW function with the same modifiers and delimiters to count the words in the string.

The O modifier is used for efficiency because the delimiters and modifiers are the same in every call to the SCAN and COUNTW functions.

```
data comma;
   keep count word;
   length word $30;
   string = ',leading, trailing,and multiple,,delimiters,,';
   delim = ',';
   modif = 'mo';
   nwords = countw(string, delim, modif);
   do count = 1 to nwords;
      word = scan(string, count, delim, modif);
      output;
   end;
```

```
run;
proc print data=comma noobs;
run;
```

Display 2.62 *Results of Finding All Words by Using the M and O Modifiers*

The SAS System

word	count
	1
leading	2
trailing	3
and multiple	4
	5
delimiters	6
	7
	8

Example 4: Using Comma-Separated Values, Substrings in Quotation Marks, and the O and R Modifiers

The following example uses the SCAN function with the O modifier and a comma as a delimiter, both with and without the R modifier.

The O modifier is used for efficiency because in each call of the SCAN or COUNTW function, the delimiters and modifiers do not change. The O modifier applies separately to each of the two instances of the SCAN function:

- The first instance of the SCAN function uses the same delimiters and modifiers every time SCAN is called. Consequently, you can use the O modifier for this instance.
- The second instance of the SCAN function uses the same delimiters and modifiers every time SCAN is called. Consequently, you can use the O modifier for this instance.
- The first instance of the SCAN function does not use the same modifiers as the second instance, but this fact has no bearing on the use of the O modifier.

```
data test;
   keep count word word_r;
   length word word_r $30;
   string = 'He said, "She said, ""No!""", not "Yes!"';
   delim = ',';
   modif = 'oq';
   nwords = countw(string, delim, modif);
   do count = 1 to nwords;
      word   = scan(string, count, delim, modif);
```

```
      word_r = scan(string, count, delim, modif||'r');
      output;
   end;
run;
proc print data=test noobs;
run;
```

Display 2.63 *Results of Comma-Separated Values and Substrings in Quotation Marks*

The SAS System

word	word_r	count
He said	He said	1
"She said, ""No!"""	She said, "No!"	2
not "Yes!"	not "Yes!"	3

Example 5: Finding Substrings of Digits by Using the D and K Modifiers

The following example finds substrings of digits. The *charlist* argument is null. Consequently, the list of characters is initially empty. The D modifier adds digits to the list of characters. The K modifier treats all characters that are not in the list as delimiters. Therefore, all characters except digits are delimiters.

```
data digits;
   keep count digits;
   length digits $20;
   string = 'Call (800) 555-1234 now!';
   do until(digits = ' ');
      count+1;
      digits = scan(string, count, , 'dko');
      output;
   end;
run;
proc print data=digits noobs;
run;
```

Display 2.64 *Results of Finding Substrings of Digits by Using the D and K Modifiers*

The SAS System

digits	count
800	1
555	2
1234	3
	4

See Also

Functions:

- "COUNTW Function" on page 343
- "FINDW Function" on page 467

CALL Routines:

- "CALL SCAN Routine" on page 237

SDF Function

Returns a survival function.

Category: Probability

See: "CDF Function" on page 277

Syntax

SDF(*dist*,*quantile*,*parm-1*,...,*parm-k*)

Required Arguments

dist
is a character string that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	**BERNOULLI**
Beta	**BETA**
Binomial	**BINOMIAL**
Cauchy	**CAUCHY**
Chi-Square	**CHISQUARE**
Exponential	**EXPONENTIAL**
F	**F**
Gamma	**GAMMA**
Generalized Poisson	**GENPOISSON**
Geometric	**GEOMETRIC**
Hypergeometric	**HYPERGEOMETRIC**

Distribution	Argument
Laplace	**LAPLACE**
Logistic	**LOGISTIC**
Lognormal	**LOGNORMAL**
Negative binomial	**NEGBINOMIAL**
Normal	**NORMAL** | **GAUSS**
Normal mixture	**NORMALMIX**
Pareto	**PARETO**
Poisson	**POISSON**
T	**T**
Tweedie	**TWEEDIE**
Uniform	**UNIFORM**
Wald (inverse Gaussian)	**WALD** | **IGAUSS**
Weibull	**WEIBULL**

Note Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

quantile
is a numeric constant, variable or expression that specifies the value of a random variable.

parm-1,...,parm-k
are optional *shape*, *location*, or *scale* parameters appropriate for the specific distribution.

Details

The SDF function computes the survival function (upper tail) from various continuous and discrete distributions. For more information, see the "CDF Function" on page 277.

Example

The following SAS statements produce these results.

SAS Statement	Result
`y=sdf('BERN',0,.25);`	`0.25`

SAS Statement	Result
`y=sdf('BETA',0.2,3,4);`	0.09011
`y=sdf('BINOM',4,.5,10);`	0.62305
`y=sdf('CAUCHY',2);`	0.14758
`y=sdf('CHISQ',11.264,11);`	0.42142
`y=sdf('EXPO',1);`	0.36788
`y=sdf('F',3.32,2,3);`	0.17361
`y=sdf('GAMMA',1,3);`	0.91970
`y=sdf('GENPOISSON',.9,1,.7);`	0.6321205588
`y=sdf('HYPER',2,200,50,10);`	0.47633
`y=sdf('LAPLACE',1);`	0.18394
`y=sdf('LOGISTIC',1);`	0.26894
`y=sdf('LOGNORMAL',1);`	0.5
`y=sdf('NEGB',1,.5,2);`	0.5
`y=sdf('NORMAL',1.96);`	0.025
`y=pdf('NORMALMIX',2.3,3,.33,.33,.34, .5,1.5,2.5,.79,1.6,4.3);`	0.2819
`y=sdf('PARETO',1,1);`	1
`y=sdf('POISSON',2,1);`	0.08030
`y=sdf('T',.9,5);`	0.20469
`y=sdf('TWEEDIE',.8,5);`	0.4082370836
`y=sdf('UNIFORM',0.25);`	0.75
`y=sdf('WALD',1,2);`	0.37230
`y=sdf('WEIBULL',1,2);`	0.36788

See Also

Functions:

- "CDF Function" on page 277
- "LOGCDF Function" on page 640
- "LOGPDF Function" on page 642
- "LOGSDF Function" on page 644
- "PDF Function" on page 722
- "QUANTILE Function" on page 799
- "SQUANTILE Function" on page 881

SECOND Function

Returns the second from a SAS time or datetime value.

Category: Date and Time

Syntax

SECOND(*time* | *datetime*)

Required Arguments

time
is a numeric constant, variable, or expression with a value that represents a SAS time value.

datetime
is a numeric constant, variable, or expression with a value that represents a SAS datetime value.

Details

The SECOND function produces a numeric value that represents a specific second of the minute. The result can be any number that is >= 0 and < 60.

Example

The following SAS statements produce these results.

SAS Statement	Result

`time='3:19:24't;`
`s=second(time);`
`put s;`
	`24`

`time='6:25:65't;`
`s=second(time);`
`put s;`
	`5`

`time='3:19:60't;`
`s=second(time);`
`put s;`
	`0`

See Also

Functions:

- "HOUR Function" on page 534
- "MINUTE Function" on page 661

SIGN Function

Returns the sign of a value.

Category: Mathematical

Syntax

SIGN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The SIGN function returns the following values:

-1
if *argument* < 0

0
if *argument* = 0

1
if *argument* > 0.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=sign(-5);`	`-1`
`x=sign(5);`	`1`
`x=sign(0);`	`0`

SIN Function

Returns the sine.

Category: Trigonometric

Syntax

SIN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression and is expressed in radians. If the magnitude of *argument* is so great that `mod(argument,pi)` is accurate to less than about three decimal places, SIN returns a missing value.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=sin(0.5);`	`0.4794255386`
`x=sin(0);`	`0`
`x=sin(3.14159/4);`	`.7071063121`

SINH Function

Returns the hyperbolic sine.

Category: Hyperbolic

Syntax

SINH(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The SINH function returns the hyperbolic sine of the argument, which is given by

$$\left(\varepsilon^{argument} - \varepsilon^{-argument}\right) / 2$$

Example

The following SAS statements produce these results.

SAS Statement	Result
x=sinh(0);	0
x=sinh(1);	1.1752011936
x=sinh(-1.0);	-1.175201194

SKEWNESS Function

Returns the skewness of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

SKEWNESS(*argument-1*,*argument-2*,*argument-3*<,…*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

At least three non-missing arguments are required. Otherwise, the function returns a missing value. If all non-missing arguments have equal values, the skewness is mathematically undefined. The SKEWNESS function returns a missing value and sets _ERROR_ equal to 1.

The argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
x1=skewness(0,1,1);	-1.732050808
x2=skewness(2,4,6,3,1);	0.5901286564
x3=skewness(2,0,0);	1.7320508076
x4=skewness(of x1-x3);	-0.953097714

SLEEP Function

For a specified period of time, suspends the execution of a program that invokes this function.

Category: Special

See: "SLEEP Function: Windows" in *SAS Companion for Windows*

Syntax

SLEEP(*n*<, *unit*>)

Required Argument

n

is a numeric constant, variable, or expression that specifies the number of units of time for which you want to suspend execution of a program.

Range $n \geq 0$

Optional Argument

unit

specifies the unit of time in seconds, which is applied to *n*. For example, 1 corresponds to 1 second, .001 corresponds to 1 millisecond, and 5 corresponds to 5 seconds.

Default 1 in a Windows PC environment, .001 in other environments

Details

The SLEEP function suspends the execution of a program that invokes this function for a period of time that you specify. The program can be a DATA step, macro, IML, SCL, or anything that can invoke a function. 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 tells SAS to delay the execution of the DATA step PAYROLL for 20 seconds:

```
data payroll;
   time_slept=sleep(20,1);
   ...more SAS statements...
run;
```

Example 2: Suspending Execution Based on a Calculation of Sleep Time

The following example tells SAS to suspend the execution of the DATA step BUDGET until March 1, 2013, at 3:00 AM. SAS calculates the length of the suspension based on the target date and the date and time that the DATA step begins to execute.

```
data budget;
   sleeptime='01mar2013:03:00'dt-'01mar2013:2:59:30'dt;
   time_calc=sleep(sleeptime,1);
   put 'Calculation of sleep time:';
   put sleeptime='seconds';
run;
```

SAS writes the following output to the log:

```
Calculation of sleep time:
sleeptime=30 seconds
```

See Also

CALL Routines:

- "CALL SLEEP Routine" on page 247

SMALLEST Function

Returns the *k*th smallest nonmissing value.

Category: Descriptive Statistics

Syntax

SMALLEST (*k*, *value-1*<, *value-2*… >)

Required Arguments

k
is a numeric constant, variable, or expression that specifies which value to return.

value
specifies a numeric constant, variable, or expression.

Details

If *k* is missing, less than zero, or greater than the number of values, the result is a missing value and _ERROR_ is set to 1. Otherwise, if *k* is greater than the number of non-missing values, the result is a missing value but _ERROR_ is not set to 1.

Comparisons

The SMALLEST function differs from the ORDINAL function in that the SMALLEST function ignores missing values, but the ORDINAL function counts missing values.

Example

This example compares the values that are returned by the SMALLEST function with values that are returned by the ORDINAL function.

```
data comparison;
   label smallest_num='SMALLEST Function' ordinal_num='ORDINAL Function';
   do k = 1 to 4;
```

```
      smallest_num = smallest(k, 456, 789, .Q, 123);
      ordinal_num  = ordinal (k, 456, 789, .Q, 123);
      output;
   end;
run;
proc print data=comparison label noobs;
   var k smallest_num ordinal_num;
   title 'Results From the SMALLEST and the ORDINAL Functions';
run;
```

Display 2.65 *Comparison of Values: The SMALLEST and the ORDINAL Functions*

Results From the SMALLEST and the ORDINAL Functions

k	SMALLEST Function	ORDINAL Function
1	123	Q
2	456	123
3	789	456
4	.	789

See Also

Functions:

- “LARGEST Function” on page 611
- “ORDINAL Function” on page 718
- “PCTL Function” on page 720

SOAPWEB Function

Calls a Web service by using basic Web authentication; credentials are provided in the arguments.

Category: Web Service

Syntax

SOAPWEB (IN, URL <,*options*>)

Required Arguments

IN

specifies a character value that is the fileref. IN is used to input XML data that contains the SOAP request.

URL
specifies a character value that is the URL of the Web service endpoint.

Optional Arguments

OUT
specifies a character value that is the fileref where the SOAP response output XML will be written.

SOAPACTION
specifies a character value that is a SOAPAction element to invoke on the Web service.

WEBUSERNAME
specifies a character value that is a user name for either basic or NTLM Web authentication.

WEBPASSWORD
specifies a character value that is a password for either basic or NTLM Web authentication. Encodings that are produced by PROC PWENCODE are supported.

WEBDOMAIN
specifies a character value that is the domain or realm for the user name and password for NTLM authentication.

MUSTUNDERSTAND
specifies a numeric value that is the setting for the mustUnderstand attribute in the SOAP header.

PROXYPORT
specifies a numeric value that is an HTTP proxy server port.

PROXYHOST
specifies a character value that is an HTTP proxy server host.

PROXYUSERNAME
specifies a character value that is an HTTP proxy server user name.

PROXYPASSWORD
specifies a character value that is an HTTP proxy server password. Encodings that are produced by PROC PWENCODE are supported.

CONFIGFILE
specifies a character value that is a Spring configuration file that is used primarily to set time-out values.

DEBUG
specifies a character value that is the full path to a file that is used for debugging logging output.

Example

The following example shows how to use the SOAPWEB function in a DATA step:

```
FILENAME request 'c:\temp\Request.xml';
FILENAME response 'c:\temp\Response.xml';

data _null_;
   url="http://www.weather.gov/forecasts/xml/SOAP_server/ndfdXMLserver.php";
   soapaction=
   "http://www.weather.gov/forecasts/xml/DWMLgen/wsdl/ndfdXML.wsdl#CornerPoints";
```

```
   proxyhost="someproxy.abc.xyz.com";
   proxyport=80;

   rc = soapweb("request", url, "response", soapaction, , , , , proxyport,
        proxyhost);
run;
```

This section provides information about the SOAP request:

```
Request.xml:
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ndf="http://www.weather.gov/forecasts/xml/DWMLgen/wsdl/ndfdXML.wsdl">
   <soapenv:Header/>
   <soapenv:Body>
      <ndf:CornerPoints soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
           encoding/">
         <sector xsi:type="dwml:sectorType"
xmlns:dwml="http://www.weather.gov/forecasts/xml/DWMLgen/schema/DWML.xsd">
      alaska</sector>
      </ndf:CornerPoints>
   </soapenv:Body>
</soapenv:Envelope>
```

See Also

Functions:

- “SOAPWS Function” on page 873
- “SOAPWSMETA Function” on page 875
- “SOAPWEBMETA Function” on page 867
- “SOAPWIPSERVICE Function” on page 869
- “SOAPWIPSRS Function” on page 871

SOAPWEBMETA Function

Calls a Web service by using basic Web authentication; credentials for the authentication domain are retrieved from metadata.

Category: Web Service

Syntax

SOAPWEBMETA (IN, URL <,*options*>)

Required Arguments

IN

specifies a character value that is the fileref. IN is used to input XML data that contains the SOAP request.

URL
specifies a character value that is the URL of the Web service endpoint.

Optional Arguments

OUT
specifies a character value that is the fileref where the SOAP response output XML will be written.

SOAPACTION
specifies a character value that is a SOAPAction element to invoke on the Web service.

WEBAUTHDOMAIN
specifies a character value that is the authentication domain from which to retrieve a user name and password from metadata for basic Web authentication.

MUSTUNDERSTAND
specifies a numeric value that is the setting for the mustUnderstand attribute in the SOAP header.

PROXYPORT
specifies a numeric value that is an HTTP proxy server port.

PROXYHOST
specifies a character value that is an HTTP proxy server host.

PROXYUSERNAME
specifies a character value that is an HTTP proxy server user name.

PROXYPASSWORD
specifies a character value that is an HTTP proxy server password. Encodings that are produced by PROC PWENCODE are supported.

CONFIGFILE
specifies a character value that is a Spring configuration file that is used primarily to set time-out values.

DEBUG
specifies a character value that is the full path to a file that is used for debugging logging output.

Example

The following example shows how to use the SOAPWEBMETA function with the DATA step:

```
FILENAME request 'C:\temp\Request.xml';
FILENAME response 'C:\temp\Response.xml';

OPTIONS metauser="metadata-user"
        metapass="password"
        metaprotocol=bridge
        metaport=8561
        metaserver="somemachine.abc.xyz.com";

data _null_;
   url="http://somemachine/basicauth/AddService.asmx";
   soapaction="http://tempuri.org/Add";
   webauthdomain="DefaultAuth";
```

```
   rc = soapwebmeta("request", url, "response", soapaction, webauthdomain);
run;
```

See Also

Functions:

- “SOAPWS Function” on page 873
- “SOAPWSMETA Function” on page 875
- “SOAPWEB Function” on page 865
- “SOAPWIPSERVICE Function” on page 869
- “SOAPWIPSRS Function” on page 871

SOAPWIPSERVICE Function

Calls a SAS registered Web service by using WS-Security authentication; credentials are provided in the arguments.

Category: Web Service

Note: This function uses the SAS environments file.

Syntax

SOAPWIPSERVICE (IN, SERVICE <,*options*>)

Required Arguments

IN
specifies a character value that is the fileref. IN is used to input XML data that contains the SOAP request.

SERVICE
specifies the service name of the endpoint service as the service is stored in the Service Registry.

Optional Arguments

OUT
specifies a character value that is the fileref where the SOAP response output XML will be written.

SOAPACTION
specifies a character value that is a SOAPAction element to invoke on the Web service.

WSSUSERNAME
specifies a character value that is a WS-Security user name.

WSSPASSWORD
specifies a character value that is a WS-Security password, which is the password for WSSUSERNAME. Encodings that are produced by PROC PWENCODE are supported.

ENVFILE
specifies a character value that is the location of the SAS environments file.

ENVIRONMENT
specifies a character value that is the environment defined in the SAS environments file to use.

MUSTUNDERSTAND
specifies a numeric value that is the setting for the mustUnderstand attribute in the SOAP header.

CONFIGFILE
specifies a character value that is a Spring configuration file that is used primarily to set time-out values.

DEBUG
specifies a character value that is the full path to a file that is used for debugging logging output.

Details

The SAS Environments File

The name of the service is provided in the Service Registry. The SAS environments file is used to locate the Service Registry and the destination service, as well as the Security Token Service, which generates a security token with the provided credentials.

Example

The following example shows how to use the SOAPWIPSERVICE function in a DATA step:

```
FILENAME request 'c:\temp\Request.xml';
FILENAME response 'c:\temp\Response.xml';

data _null;
    service="ReportRepositoryService";
    soapaction="http://www.test.com/xml/schema/test-svcs/reportrepository-9.3/
               DirectoryServiceInterface/isDirectory";
    envfile="http://somemachine.abc.xyz.com/schemas/test-environment.xml";
    environment="test";
    wssusername="user-name";
    wsspassword="password";

    rc=soapwipservice("REQUEST", service, "RESPONSE", soapaction, wssusername,
                      wsspassword, envfile, environment);
run;
```

This section gives you information about the SOAP request:

```
Request.xml:
<soapenv:Envelope xmlns:rep="http://www.test.com/xml/schema/test-svcs/
                        reportrepository-9.3"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header>
      <Action xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://
                    www.test.com/xml/schema/test-svcs/reportrepository-9.3/
```

```
                    DirectoryServiceInterface/isDirectory</Action>
   </soapenv:Header>
   <soapenv:Body>
      <rep:isDirectoryDirectoryServiceInterfaceRequest>
         <rep:dirPathUrl>SBIP://Foundation/Users/someuser/My Folder
                              </rep:dirPathUrl>
      </rep:isDirectoryDirectoryServiceInterfaceRequest>
   </soapenv:Body>
</soapenv:Envelope>

test-environments.xml:

<environments xmlns="http://www.test.com/xml/schema/test-environments-9.3
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.test.com/xml/schema/test-environments-9.3
      http://www.test.com/xml/schema/test-environments-9.3/
             test-environments-9.3.xsd">

<environment name="default" default="true">
   <desc>Default Test Environment</desc>
   <service-registry>http://machine1.abc.xyz.com:8080/TESTWIPServices/remote/
                           serviceRegistry
   </service-registry>
</environment>

<environment name="test" default="false">
   <desc>Environment for PROC SOAP testing</desc>
   <service-registry>http://machine2.abc.xyz.com:8080/TESTWIPSoapServices/
                           Service Registry/serviceRegistry
   </service-registry>
</environment>

</environments>
```

See Also

Functions:

- “SOAPWS Function” on page 873
- “SOAPWSMETA Function” on page 875
- “SOAPWEB Function” on page 865
- “SOAPWEBMETA Function” on page 867
- “SOAPWIPSRS Function” on page 871

SOAPWIPSRS Function

Calls a SAS registered Web service by using WS-Security authentication; credentials are provided in the arguments.

Category: Web Service

Notes: The credentials that are provided are used to generate a security token to call the destination service. The URL of the destination service is provided.

The Registry Service is called directly to determine how to locate the Security Token Service.

Syntax

SOAPWIPSRS (IN, URL, SRSURL<,*options*>)

Required Arguments

IN
specifies a character value that is the fileref. IN is used to input XML data that contains the SOAP request.

URL
specifies a character value that is the URL of the Web service endpoint.

SRSURL
specifies a character value that is the URL of the System Registry Service.

Optional Arguments

OUT
specifies a character value that is the fileref where the SOAP response output XML will be written.

SOAPACTION
specifies a character value that is a SOAPAction element to invoke on the Web service.

WSSUSERNAME
specifies a character value that is a WS-Security user name.

WSSPASSWORD
specifies a character value that is a WS-Security password, which is the password for WSSUSERNAME. Encodings that are produced by PROC PWENCODE are supported.

MUSTUNDERSTAND
specifies a numeric value that is the setting for the mustUnderstand attribute in the SOAP header.

CONFIGFILE
specifies a character value that is a Spring configuration file that is used primarily to set time-out values.

DEBUG
specifies a character value that is the full path to a file that is used for debugging logging output.

Example

The following example shows how to use the SOAPWIPSRS function in a DATA step:

```
FILENAME request 'c:\temp\Request.xml';
FILENAME response 'c:\temp\Response.xml';

data _null_;
   url="http://somemachine.abc.xyz.com:8080/TESTWIPSoapServices/services/
               ReportRepositoryService";
```

```
   soapaction="http://www.test.com/xml/schema/test-svcs/reportrepository-9.3/
               DirectoryServiceInterface/isDirectory";
   srsurl="http://somemachine.abc.xyz.com:8080/TESTWIPSoapServices/services/
           ServiceRegistry";
   WSSUSERNAME="user-name";
   WSSPASSWORD="password";

   rc = soapwipsrs("request", url, srsurl, "response", soapaction, wssusername,
        wsspassword);
run;
```

This section provides information about the SOAP request:

```
Request.xml:
<soapenv:Envelope xmlns:rep="http://www.test.com/xml/schema/test-svcs/
                             reportrepository-9.3"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Header>
      <Action
xmlns="http://schemas.xmlsoap.org/ws/2004/08/addressing">http://www.test.com/
       xml/schema/test-svcs/reportrepository-9.3/DirectoryServiceInterface/
       isDirectory</Action>
   </soapenv:Header>
   <soapenv:Body>
      <rep:isDirectoryDirectoryServiceInterfaceRequest>
         <rep:dirPathUrl>SBIP://Foundation/Users/someuser/My Folder
         </rep:dirPathURL>
      </rep:isDirectoryDirectoryServiceInterfaceRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

See Also

Functions:

- “SOAPWS Function” on page 873
- “SOAPWSMETA Function” on page 875
- “SOAPWEB Function” on page 865
- “SOAPWEBMETA Function” on page 867
- “SOAPWIPSERVICE Function” on page 869

SOAPWS Function

Calls a Web service by using WS-Security authentication; credentials are provided in the arguments.

Category: Web Service

Syntax

SOAPWS (IN, URL<,*options*>)

Required Arguments

IN
specifies a character value that is the fileref. IN is used to input XML data that contains the SOAP request.

URL
specifies a character value that is the URL of the Web service endpoint.

Optional Arguments

OUT
specifies a character value that is the fileref where the SOAP response output XML will be written.

SOAPACTION
specifies a character value that is a SOAPAction element to invoke on the Web service.

WSSUSERNAME
specifies a character value that is a WS-Security user name.

WSSPASSWORD
specifies a character value that is a WS-Security password, which is the password for WSSUSERNAME. Encodings that are produced by PROC PWENCODE are supported.

MUSTUNDERSTAND
specifies a numeric value that is the setting for the mustUnderstand attribute in the SOAP header.

PROXYPORT
specifies a numeric value that is an HTTP proxy server port.

PROXYHOST
specifies a character value that is an HTTP proxy server host.

PROXYUSERNAME
specifies a character value that is an HTTP proxy server user name.

PROXYPASSWORD
specifies a character value that is an HTTP proxy server password. Encodings that are produced by PROC PWENCODE are supported.

CONFIGFILE
specifies a character value that is a Spring configuration file that is used primarily to set time-out values.

DEBUG
specifies a character value that is the full path to a file that is used for debugging logging output.

Example

The following example shows how to use the SOAPWS function in a DATA step:

```
FILENAME request 'C:\temp\Request.xml';
FILENAME response 'C:\temp\Response.xml';

data _null_;
   url="http://somemachine.na.abc.com/SASBIWS/ProcSoapServices.asmx";
   soapaction="http://tempuri.org/ProcSoapServices/copyintoout_xml_att";
```

```
    WSSUSERNAME="sasuser";
    WSSPASSWORD="password";

    rc = soapws("request", url, "response", soapaction, wssusername,
                wsspassword);
run;
```

See Also

Functions:

- "SOAPWSMETA Function" on page 875
- "SOAPWEB Function" on page 865
- "SOAPWEBMETA Function" on page 867
- "SOAPWIPSERVICE Function" on page 869
- "SOAPWIPSRS Function" on page 871

SOAPWSMETA Function

Calls a Web service by using WS-Security authentication; credentials for the provided authentication domain are retrieved from metadata.

Category: Web Service

Syntax

SOAPWSMETA (IN, URL <,*options*>)

Required Arguments

IN
specifies a character value that is the fileref. IN is used to input XML data that contains the SOAP request.

URL
specifies a character value that is the URL of the Web service endpoint.

Optional Arguments

OUT
specifies a character value that is the fileref where the SOAP response output XML will be written.

SOAPACTION
specifies a character value that is a SOAPAction element to invoke on the Web service.

WSSAUTHDOMAIN
specifies a character value that is the authentication domain for which to retrieve credentials to be used for WS-Security authentication.

MUSTUNDERSTAND
specifies a numeric value that is the setting for the mustUnderstand attribute in the SOAP header.

PROXYPORT
specifies a numeric value that is an HTTP proxy server port.

PROXYHOST
specifies a character value that is an HTTP proxy server host.

PROXYUSERNAME
specifies a character value that is an HTTP proxy server user name.

PROXYPASSWORD
specifies a character value that is an HTTP proxy server password. Encodings that are produced by PROC PWENCODE are supported.

CONFIGFILE
specifies a character value that is a Spring configuration file that is used primarily to set time-out values.

DEBUG
specifies a character value that is the full path to a file that is used for debugging logging output.

See Also

Functions:

- "SOAPWS Function" on page 873
- "SOAPWEB Function" on page 865
- "SOAPWEBMETA Function" on page 867
- "SOAPWIPSERVICE Function" on page 869
- "SOAPWIPSRS Function" on page 871

SOUNDEX Function

Encodes a string to facilitate searching.

Category: Character

Restrictions: SOUNDEX algorithm is English-biased.

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

Syntax

SOUNDEX(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

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

The Basics

The SOUNDEX function encodes a character string according to an algorithm that was originally developed by Margaret K. Odell and Robert C. Russel (US Patents 1261167 (1918) and 1435663 (1922)). The algorithm is described in Knuth, *The Art of Computer Programming, Volume 3*. (See "References" on page 1001.) Note that the SOUNDEX algorithm is English-biased and is less useful for languages other than English.

The SOUNDEX function returns a copy of the *argument* that is encoded by using the following steps:

1. Retain the first letter in the *argument* and discard the following letters:

   A E H I O U W Y

2. Assign the following numbers to these classes of letters:

   - 1: B F P V
   - 2: C G J K Q S X Z
   - 3: D T
   - 4: L
   - 5: M N
   - 6: R

3. If two or more adjacent letters have the same classification from Step 2, then discard all but the first. (Adjacent refers to the position in the word before discarding letters.)

The algorithm that is described in Knuth adds trailing zeros and truncates the result to the length of 4. You can perform these operations with other SAS functions.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=soundex('Paul');` `put x;`	`P4`

`word='amnesty';`
`x=soundex(word);`
`put x;`
	`A523`

SPEDIS Function

Determines the likelihood of two words matching, expressed as the asymmetric spelling distance between the two words.

Category: Character

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

Syntax

SPEDIS(*query*,*keyword*)

Required Arguments

query
identifies the word to query for the likelihood of a match. SPEDIS removes trailing blanks before comparing the value.

keyword
specifies a target word for the query. SPEDIS removes trailing blanks before comparing the value.

Details

Length of Returned Variable

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

The Basics

SPEDIS returns the distance between the query and a keyword, a nonnegative value that is usually less than 100 but never greater than 200 with the default costs.

SPEDIS computes an asymmetric spelling distance between two words as the normalized cost for converting the keyword to the query word by using a sequence of operations. SPEDIS(*QUERY*, *KEYWORD*) is *not* the same as SPEDIS(*KEYWORD*, *QUERY*).

Costs for each operation that is required to convert the keyword to the query are listed in the following table:

Operation	Cost	Explanation
match	0	no change
singlet	25	delete one of a double letter
doublet	50	double a letter
swap	50	reverse the order of two consecutive letters

Operation	Cost	Explanation
truncate	50	delete a letter from the end
append	35	add a letter to the end
delete	50	delete a letter from the middle
insert	100	insert a letter in the middle
replace	100	replace a letter in the middle
firstdel	100	delete the first letter
firstins	200	insert a letter at the beginning
firstrep	200	replace the first letter

The distance is the sum of the costs divided by the length of the query. If this ratio is greater than one, the result is rounded down to the nearest whole number.

Comparisons

The SPEDIS function is similar to the COMPLEV and COMPGED functions, but COMPLEV and COMPGED are much faster, especially for long strings.

Example

```
data words;
   input Operation $ Query $ Keyword $;
   Distance = spedis(query,keyword);
   Cost = distance * length(query);
   datalines;
match       fuzzy        fuzzy
singlet     fuzy         fuzzy
doublet     fuuzzy       fuzzy
swap        fzuzy        fuzzy
truncate    fuzz         fuzzy
append      fuzzys       fuzzy
delete      fzzy         fuzzy
insert      fluzzy       fuzzy
replace     fizzy        fuzzy
firstdel    uzzy         fuzzy
firstins    pfuzzy       fuzzy
firstrep    wuzzy        fuzzy
several     floozy       fuzzy
;
proc print data = words;
run;
```

Display 2.66 *Costs for SPEDIS Operations*

The SAS System

Obs	Operation	Query	Keyword	Distance	Cost
1	match	fuzzy	fuzzy	0	0
2	singlet	fuzy	fuzzy	6	24
3	doublet	fuuzzy	fuzzy	8	48
4	swap	fzuzy	fuzzy	10	50
5	truncate	fuzz	fuzzy	12	48
6	append	fuzzys	fuzzy	5	30
7	delete	fzzy	fuzzy	12	48
8	insert	fluzzy	fuzzy	16	96
9	replace	fizzy	fuzzy	20	100
10	firstdel	uzzy	fuzzy	25	100
11	firstins	pfuzzy	fuzzy	33	198
12	firstrep	wuzzy	fuzzy	40	200
13	several	floozy	fuzzy	50	300

See Also

Functions:

- “COMPLEV Function” on page 323
- “COMPGED Function” on page 317

SQRT Function

Returns the square root of a value.

Category: Mathematical

Syntax

SQRT(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression. *Argument* must be nonnegative.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=sqrt(36);`	`6`
`x=sqrt(25);`	`5`
`x=sqrt(4.4);`	`2.0976176963`

SQUANTILE Function

Returns the quantile from a distribution when you specify the right probability (SDF).

Category: Quantile

See: "SDF Function" on page 856

Syntax

SQUANTILE(*dist*, *probability*, *parm-1, ... , parm-k*)

Required Arguments

dist

is a character constant, variable, or expression that identifies the distribution. Valid distributions are as follows:

Distribution	Argument
Bernoulli	**BERNOULLI**
Beta	**BETA**
Binomial	**BINOMIAL**
Cauchy	**CAUCHY**
Chi-Square	**CHISQUARE**
Exponential	**EXPONENTIAL**
F	**F**
Gamma	**GAMMA**
Generalized Poisson	**GENPOISSON**

Distribution	Argument
Geometric	**GEOMETRIC**
Hypergeometric	**HYPERGEOMETRIC**
Laplace	**LAPLACE**
Logistic	**LOGISTIC**
Lognormal	**LOGNORMAL**
Negative binomial	**NEGBINOMIAL**
Normal	**NORMAL** | **GAUSS**
Normal mixture	**NORMALMIX**
Pareto	**PARETO**
Poisson	**POISSON**
T	**T**
Tweedie	**TWEEDIE**
Uniform	**UNIFORM**
Wald (inverse Gaussian)	**WALD** | **IGAUSS**
Weibull	**WEIBULL**

Note: Except for T, F, and NORMALMIX, you can minimally identify any distribution by its first four characters.

probability
is a numeric constant, variable, or expression that specifies the value of a random variable.

parm-1,…,parm-k
are optional *shape*, *location*, or *scale* parameters that are appropriate for the specific distribution.

Details

The SQUANTILE function computes the probability from various continuous and discrete distributions. For more information, see "Details" on page 279.

Example

This is an example of the SQUANTILE function.

```
data;
   dist = 'logistic';
   sdf = squantile(dist,1.e-20);
   put sdf=;
   p = sdf(dist,sdf);
   put p= /* p will be 1.e-20 */;
run;
```

SAS writes the following output to the log:

```
sdf=46.05170186
p=1E-20
```

See Also

Functions:

- “CDF Function” on page 277
- “LOGCDF Function” on page 640
- “LOGPDF Function” on page 642
- “LOGSDF Function” on page 644
- “PDF Function” on page 722
- “QUANTILE Function” on page 799
- “SDF Function” on page 856

STD Function

Returns the standard deviation of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

STD(*argument-1*,*argument-2*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=std(2,6);`	`2.8284271247`

SAS Statement	Result
x2=std(2,6,.);	2.8284271427
x3=std(2,4,6,3,1);	1.9235384062
x4=std(of x1-x3);	0.5224377453

STDERR Function

Returns the standard error of the mean of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

STDERR(*argument-1*,*argument-2*<,…*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
x1=stderr(2,6);	2
x2=stderr(2,6,.);	2
x3=stderr(2,4,6,3,1);	0.8602325267
x4=stderr(of x1-x3);	0.3799224911

STFIPS Function

Converts state postal codes to FIPS state codes.

Category: State and ZIP Code

Syntax

STFIPS(*postal-code*)

Required Argument

postal-code
specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

The STFIPS function converts a two-character state postal code (or world-wide GSA geographic code for U.S. territories) to the corresponding numeric U.S. Federal Information Processing Standards (FIPS) code.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed case state name.

Example

The following examples show the differences when using STFIPS, STNAME, and STNAMEL.

SAS Statement	Result
`fips=stfips ('NC');` `put fips;`	`37`
`state=stname('NC');` `put state;`	`NORTH CAROLINA`
`state=stnamel('NC');` `put state;`	`North Carolina`

See Also

Functions:

- “FIPNAME Function” on page 475
- “FIPNAMEL Function” on page 476
- “FIPSTATE Function” on page 477
- “STNAME Function” on page 886
- “STNAMEL Function” on page 887

STNAME Function

Converts state postal codes to uppercase state names.

Category: State and ZIP Code

Syntax

STNAME(*postal-code*)

Required Argument

postal-code
specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

The STNAME function converts a two-character state postal code (or world-wide GSA geographic code for U.S. territories) to the corresponding state name in uppercase.

Note: For Version 6, the maximum length of the value that is returned is 200 characters. For Version 7 and beyond, the maximum length is 20 characters.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed case state name.

Example

The following SAS statements produce these results.

SAS Statement	Result
`fips=stfips ('NC');` `put fips;`	`37`
`state=stname('NC');` `put state;`	`NORTH CAROLINA`
`state=stnamel('NC');` `put state;`	`North Carolina`

See Also

Functions:

- “FIPNAME Function” on page 475

- "FIPNAMEL Function" on page 476
- "FIPSTATE Function" on page 477
- "STFIPS Function" on page 884
- "STNAMEL Function" on page 887

STNAMEL Function

Converts state postal codes to mixed case state names.

Category: State and ZIP Code

Syntax

STNAMEL(*postal-code*)

Required Argument

postal-code
specifies a character expression that contains the two-character standard state postal code. Characters can be mixed case. The function ignores trailing blanks, but generates an error if the expression contains leading blanks.

Details

If the STNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

The STNAMEL function converts a two-character state postal code (or world-wide GSA geographic code for U.S. territories) to the corresponding state name in mixed case.

Note: For Version 6, the maximum length of the value that is returned is 200 characters. For Version 7 and beyond, the maximum length is 20 characters.

Comparisons

The STFIPS, STNAME, and STNAMEL functions take the same argument but return different values. STFIPS returns a numeric U.S. Federal Information Processing Standards (FIPS) code. STNAME returns an uppercase state name. STNAMEL returns a mixed case state name.

Example

The following examples show the differences when using STFIPS, STNAME, and STNAMEL.

SAS Statement	Result
`fips=stfips ('NC');` `put fips;`	`37`
`state=stname('NC');` `put state;`	`NORTH CAROLINA`

SAS Statement	Result
`state=stnamel('NC');` `put state;`	`North Carolina`

See Also

Functions:

- "FIPNAME Function" on page 475
- "FIPNAMEL Function" on page 476
- "FIPSTATE Function" on page 477
- "STFIPS Function" on page 884
- "STNAME Function" on page 886

STRIP Function

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

Category: Character

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

Syntax

STRIP(*string*)

Required Argument

string
is a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the STRIP function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

The Basics

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

Assigning the results of STRIP to a variable does not affect the length of the receiving variable. If the value that is trimmed is shorter than the length of the receiving variable, SAS pads the value with new trailing blanks.

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

Comparisons

The following list compares the STRIP function with the TRIM and TRIMN functions:

- For strings that are blank, the STRIP and TRIMN functions return a string with a length of zero, whereas the TRIM function returns a single blank.
- For strings that lack leading blanks, the STRIP and TRIMN functions return the same value.
- For strings that lack leading blanks but have at least one non-blank character, the STRIP and TRIM functions return the same value.

Note: **STRIP(*string*)** returns the same result as **TRIMN(LEFT(*string*))**, but the STRIP function runs faster.

Example

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

```
data lengthn;
   input string $char8.;
   original = '*' || string || '*';
   stripped = '*' || strip(string) || '*';
   datalines;
abcd
  abcd
    abcd
abcdefgh
 x y z
;
proc print data=lengthn;
run;
```

Display 2.67 *Results from the STRIP Function*

The SAS System

Obs	string	original	stripped
1	abcd	*abcd *	*abcd*
2	abcd	* abcd *	*abcd*
3	abcd	* abcd*	*abcd*
4	abcdefgh	*abcdefgh*	*abcdefgh*
5	x y z	* x y z *	*x y z*

See Also

Functions:

- "CAT Function" on page 263
- "CATS Function" on page 270
- "CATT Function" on page 272
- "CATX Function" on page 274
- "LEFT Function" on page 616
- "TRIM Function" on page 924
- "TRIMN Function" on page 926

SUBPAD Function

Returns a substring that has a length you specify, using blank padding if necessary.

Category: Character

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

Syntax

SUBPAD(*string*, *position* <, *length*>)

Required Arguments

string
specifies a character constant, variable, or expression.

position
is a positive integer that specifies the position of the first character in the substring.

Optional Argument

length
is a non-negative integer that specifies the length of the substring. If you do not specify *length*, the SUBPAD function returns the substring that extends from the position that you specify to the end of the string.

Details

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

If the substring that you specify extends beyond the length of the string, the result is padded with blanks.

Comparisons

The SUBPAD function is similar to the SUBSTR function except for the following differences:

- If the value of *length* in SUBPAD is zero, SUBPAD returns a zero-length string. If the value of *length* in SUBSTR is zero, SUBSTR

- writes a note to the log stating that the third argument is invalid
- sets _ERROR_=1
- returns the substring that extends from the position that you specified to the end of the string.

- If the substring that you specify extends past the end of the string, SUBPAD pads the result with blanks to yield the length that you requested. If the substring that you specify extends past the end of the string, SUBSTR
  - writes a note to the log stating that the third argument is invalid
  - sets _ERROR_=1
  - returns the substring that extends from the position that you specified to the end of the string.

See Also

Functions:

- "SUBSTRN Function" on page 894

SUBSTR (left of =) Function

Replaces character value contents.

Category: Character

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

Tip: DBCS equivalent functions are KSUBSTR and KSUBSTRB .

Syntax

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

Required Arguments

variable
specifies a character variable.

position
specifies a numeric constant, variable, or expression that is the beginning character position.

characters-to-replace
specifies a character constant, variable, or expression that will replace the contents of *variable*.

Tip Enclose a literal string of characters in quotation marks.

Optional Argument

length
specifies a numeric constant, variable, or expression that is the length of the substring that will be replaced.

Restriction *length* cannot be larger than the length of the expression that remains in *variable* after *position*.

Tip If you omit *length*, SAS uses all of the characters on the right side of the assignment statement to replace the values of *variable*.

Details

If you use an undeclared variable, it will be assigned a default length of 8 when the SUBSTR function is compiled.

When you use the SUBSTR function on the left side of an assignment statement, SAS replaces the value of *variable* with the expression on the right side. SUBSTR replaces *length* characters starting at the character that you specify in *position*.

Example

The following SAS statements produce these results.

SAS Statement	Result

`a='KIDNAP';`
`substr(a,1,3)='CAT';`
`put a;`
	`CATNAP`

`b=a;`
`substr(b,4)='TY';`
`put b;`
	`CATTY`

See Also

Functions:

- "SUBSTR (right of =) Function" on page 892

SUBSTR (right of =) Function

Extracts a substring from an argument.

Category: Character

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

Tip: DBCS equivalent functions are KSUBSTR and KSUBSTRB.

Syntax

<*variable*=> **SUBSTR(***string*,*position*<,*length*>)

Required Arguments

string
specifies a character constant, variable, or expression.

position
specifies a numeric constant, variable, or expression that is the beginning character position.

Optional Arguments

variable
specifies a valid SAS variable name.

length
specifies a numeric constant, variable, or expression that is the length of the substring to extract.

Interaction If *length* is zero, a negative value, or larger than the length of the expression that remains in *string* after *position*, SAS extracts the remainder of the expression. SAS also sets _ERROR_ to 1 and prints a note to the log indicating that the *length* argument is invalid.

Tip If you omit *length*, SAS extracts the remainder of the expression.

Details

In a DATA step, if the SUBSTR (right of =) function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The SUBSTR function returns a portion of an expression that you specify in *string*. The portion begins with the character that you specify by *position*, and is the number of characters that you specify in *length*.

Example

The following SAS statements produce these results.

SAS Statement	Result
	`----+----1----+----2`

`date='06MAY98';`
`month=substr(date,3,3);`
`year=substr(date,6,2);`
`put @1 month @5 year;`
	`MAY 98`

See Also

Functions:

- "SUBPAD Function" on page 890
- "SUBSTR (left of =) Function" on page 891
- "SUBSTRN Function" on page 894

SUBSTRN Function

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

Category: Character

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

Tip: KSUBSTR has the same functionality.

Syntax

SUBSTRN(*string*, *position* <, *length*>)

Required Arguments

string
specifies a character or numeric constant, variable, or expression.

If *string* is numeric, then it is converted to a character value that uses the BEST32. format. Leading and trailing blanks are removed, and no message is sent to the SAS log.

position
is an integer that specifies the position of the first character in the substring.

Optional Argument

length
is an integer that specifies the length of the substring. If you do not specify *length*, the SUBSTRN function returns the substring that extends from the position that you specify to the end of the string.

Details

Length of Returned Variable

In a DATA step, if the SUBSTRN function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The Basics

The following information applies to the SUBSTRN function:

- The SUBSTRN function returns a string with a length of zero if either *position* or *length* has a missing value.

- If the position that you specify is non-positive, the result is truncated at the beginning, so that the first character of the result is the first character of the string. The length of the result is reduced accordingly.
- If the length that you specify extends beyond the end of the string, the result is truncated at the end, so that the last character of the result is the last character of the string.

Using the SUBSTRN Function in a Macro

If you call SUBSTRN by using the %SYSFUNC macro, then the macro processor resolves the first argument (*string*) to determine whether the argument is character or numeric. If you do not want the first argument to be evaluated as a macro expression, use one of the macro-quoting functions in the first argument.

Comparisons

The following table lists comparisons between the SUBSTRN and the SUBSTR functions:

Condition	Function	Result
the value of *position* is nonpositive	SUBSTRN	returns a result beginning at the first character of the string.
the value of *position* is nonpositive	SUBSTR	• writes a note to the log stating that the second argument is invalid. • sets _ERROR_ =1. • returns the substring that extends from the position that you specified to the end of the string.
the value of *length* is nonpositive	SUBSTRN	returns a result with a length of zero.
the value of *length* is nonpositive	SUBSTR	• writes a note to the log stating that the third argument is invalid. • sets _ERROR_ =1. • returns the substring that extends from the position that you specified to the end of the string.
the substring that you specify extends past the end of the string	SUBSTRN	truncates the result.
the substring that you specify extends past the end of the string	SUBSTR	• writes a note to the log stating that the third argument is invalid. • sets _ERROR_=1. • returns the substring that extends from the position that you specified to the end of the string.

Examples

Example 1: Manipulating Strings with the SUBSTRN Function

The following example shows how to manipulate strings with the SUBSTRN function.

```
data test;
   retain string "abcd";
   drop string;
   do Position = -1 to 6;
      do Length = max(-1,-position) to 7-position;
         Result = substrn(string, position, length);
         output;
      end;
   end;
   datalines;
abcd
;
proc print noobs data=test;
run;
```

Display 2.68 *Output from the SUBSTRN Function*

The SAS System

Position	Length	Result
-1	1	
-1	2	
-1	3	a
-1	4	ab
-1	5	abc
-1	6	abcd
-1	7	abcd
-1	8	abcd
0	0	
0	1	
0	2	a
0	3	ab
0	4	abc
0	5	abcd
0	6	abcd
0	7	abcd
1	-1	
1	0	
1	1	a
1	2	ab
1	3	abc
1	4	abcd
1	5	abcd
1	6	abcd
2	-1	
2	0	
2	1	b
2	2	bc
2	3	bcd
2	4	bcd
2	5	bcd

Example 2: Comparison between the SUBSTR and SUBSTRN Functions

The following example compares the results of using the SUBSTR function and the SUBSTRN function when the first argument is numeric.

```
data _null_;
     substr_result = "*" || substr(1234.5678,2,6) || "*";
     put substr_result=;
     substrn_result = "*" || substrn(1234.5678,2,6)  || "*";
     put substrn_result=;
run;
```

Log 2.22 *Results from the SUBSTR and SUBSTRN Functions*

```
substr_result=*  1234*
substrn_result=*234.56*
```

See Also

Functions:

- "SUBPAD Function" on page 890
- "SUBSTR (left of =) Function" on page 891
- "SUBSTR (right of =) Function" on page 892

SUM Function

Returns the sum of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

SUM(*argument*,*argument*,…)

Required Argument

argument
specifies a numeric constant, variable, or expression. If all the arguments have missing values, then one of the following occurs:

- If you use only one argument, then the value of that argument is returned.
- If you use two or more arguments, then a standard missing value (.) is returned.

Otherwise, the result is the sum of the nonmissing values. The argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=sum(4,9,3,8);`	24
`x2=sum(4,9,3,8,.);`	24

`x1=9;`
`x2=39;`
`x3=sum(of x1-x2);`
	48

`x1=5; x2=6; x3=4; x4=9;`
`y1=34; y2=12; y3=74; y4=39;`
`result=sum(of x1-x4, of y1-y5);`
	183

`x1=55;`
`x2=35;`
`x3=6;`
`x4=sum(of x1-x3, 5);`
	101

`x1=7;`
`x2=7;`
`x5=sum(x1-x2);`
	0

`y1=20;`
`y2=30;`
`x6=sum(of y:);`
	50

SUMABS Function

Returns the sum of the absolute values of the non-missing arguments.

Category: Descriptive Statistics

Syntax

SUMABS(*value-1* <,*value-2*…>)

Required Argument

value
specifies a numeric expression.

Details

If all arguments have missing values, then the result is a missing value. Otherwise, the result is the sum of the absolute values of the non-missing values.

Examples

Example 1: Calculating the Sum of Absolute Values

The following example returns the sum of the absolute values of the non-missing arguments.

```
data _null_;
   x=sumabs(1,.,-2,0,3,.q,-4);
   put x=;
run;
```

SAS writes the following output 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 non-missing arguments.

```
data _null_;
   x1 = 1;
   x2 = 3;
   x3 = 4;
   x4 = 3;
   x5 = 1;
   x = sumabs(of x1-x5);
   put x=;
run;
```

SAS writes the following output to the log:

```
x=12
```

SYMEXIST Function

Returns an indication of the existence of a macro variable.

Category: Macro

See: "SYMEXIST Function" in *SAS Macro Language: Reference*

Syntax

SYMEXIST (*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name

- a character expression that constructs a macro variable name

Details

The SYMEXIST function searches any enclosing local symbol tables and then the global symbol table for the indicated macro variable and returns `1` if the macro variable is found or `0` if the macro variable is not found.

For more information, see the"SYMEXIST Function" in *SAS Macro Language: Reference*.

SYMGET Function

Returns the value of a macro variable during DATA step execution.

Category: Macro

Syntax

SYMGET(*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name
- a character expression that constructs a macro variable name

Details

If the SYMGET function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

The SYMGET function returns the value of a macro variable during DATA step execution. For more information, see the "SYMGET Function" in *SAS Macro Language: Reference*.

See Also

- *SAS Macro Language: Reference*

CALL Routines:

- "CALL SYMPUT Routine" on page 256

SYMGLOBL Function

Returns an indication of whether a macro variable is in global scope to the DATA step during DATA step execution.

Category: Macro

See: "SYMGLOBL Function" in *SAS Macro Language: Reference*

Syntax

SYMGLOBL (*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand.
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name.
- a character expression that constructs a macro variable name.

Details

The SYMGLOBL function searches only the global symbol table for the indicated macro variable and returns `1` if the macro variable is found or `0` if the macro variable is not found.

For more information, see "SYMGLOBL Function" in *SAS Macro Language: Reference*.

SYMLOCAL Function

Returns an indication of whether a macro variable is in local scope to the DATA step during DATA step execution.

Category: Macro

See: "SYMLOCAL Function" in *SAS Macro Language: Reference*

Syntax

SYMLOCAL (*argument*)

Required Argument

argument
can be one of the following items:

- the name of a macro variable within double quotation marks but without an ampersand.
- the name of a DATA step character variable, specified with no quotation marks, which contains a macro variable name.
- a character expression that constructs a macro variable name.

Details

The SYMLOCAL function searches the enclosing local symbol tables for the indicated macro variable and returns `1` if the macro variable is found or `0` if the macro variable is not found.

For more information, see “SYMLOCAL Function” in *SAS Macro Language: Reference* .

SYSEXIST Function

Returns a value that indicates whether an operating-environment variable exists in your environment.

Categories: SAS File I/O
Special

Syntax

SYSEXIST (*argument*)

Required Argument

argument
specifies a character variable that is the name of an operating-environment variable that you want to test.

Details

The SYSEXIST function searches for the existence of an operating-environment variable and returns 1 if the variable is found or 0 if the variable is not found.

Comparisons

The SYSEXIST function tests for the existence of an operating-environment variable. The SYSGET function retrieves the value of an operating-environment variable.

Example

The following example assumes that HOME is a valid operating-environment variable in your environment, and that TEST is not valid. SYSEXIST tests both values:

```
data _null_;
   rc=sysexist("HOME");
   put rc=;
   rc=sysexist("TEST");
   put rc=;
run;
```

SAS writes the following output to the log:

```
rc=1
rc=0
```

If SYSEXIST returns a value of 1, then the variable that is being tested is an operating-environment variable. If SYSEXIST returns a value of 0, then the variable that is being tested is not an operating-environment variable in your environment.

See Also

Functions:

- "SYSGET Function" on page 904

SYSGET Function

Returns the value of the specified operating environment variable.

Category: Special

See: "SYSGET Function: UNIX" in *SAS Companion for UNIX Environments*
"SYSGET Function: z/OS" in *SAS Companion for z/OS*

Syntax

SYSGET(*operating-environment-variable*)

Required Argument

operating-environment-variable
is a character constant, variable, or expression with a value that is the name of an operating environment variable. The case of *operating-environment-variable* must agree with the case that is stored in the operating environment. Trailing blanks in the argument of SYSGET are significant. Use the TRIM function to remove them.

Operating Environment Information
The term *operating-environment-variable* used in the description of this function refers to a name that represents a numeric, character, or logical value in the operating environment. For more information, see the SAS documentation for your operating environment.

Details

If the SYSGET function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

If the value of the operating environment variable is truncated or the variable is not defined in the operating environment, SYSGET displays a warning message in the SAS log.

Example

This example obtains the value of two environment variables in the UNIX environment:

```
data _null_;
   length result $200;
   input env_var $;
   result=sysget(trim(env_var));
   put env_var= result=;
```

```
   datalines;
USER
PATH
;
```

Executing this DATA step for user ABCDEF displays these lines:

```
ENV_VAR=USER RESULT=abcdef
ENV_VAR=PATH RESULT=path-for-abcdef
```

See Also

Functions:

- "ENVLEN Function" on page 392

SYSMSG Function

Returns error or warning message text from processing the last data set or external file function.

Categories: SAS File I/O
External Files

Syntax

SYSMSG()

Details

SYSMSG returns the text of error messages or warning messages that are produced when a data set or external file access function encounters an error condition. If no error message is available, the returned value is blank. The internally stored error message is reset to blank after a call to SYSMSG, so subsequent calls to SYSMSG before another error condition occurs return blank values.

Example

This example uses SYSMSG to write to the SAS log the error message generated if FETCH cannot copy the next observation into the Data Set Data Vector. The return code is 0 only when a record is fetched successfully:

```
%let rc=%sysfunc(fetch(&dsid));
%if &rc ne 0 %then
   %put %sysfunc(sysmsg());
```

See Also

Functions:

- "FETCH Function" on page 405
- "SYSRC Function" on page 909

SYSPARM Function

Returns the system parameter string.

Category: Special

Syntax

SYSPARM()

Details

If the SYSPARM function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

SYSPARM allows you to access a character string specified with the SYSPARM= system option at SAS invocation or in an OPTIONS statement.

Note: If the SYSPARM= system option is not specified, the SYSPARM function returns a string with a length of zero.

Example

This example shows the SYSPARM= system option and the SYSPARM function.

```
options sysparm='yes';
data a;
   If sysparm()='yes' then
      do;
      ...SAS Statements...
      end;
run;
```

See Also

System Options:

- "SYSPARM= System Option" in *SAS Macro Language: Reference*

SYSPROCESSID Function

Returns the process ID of the current process.

Category: Special

Syntax

SYSPROCESSID()

Details

The SYSPROCESSID function returns the 32–character hexadecimal ID of the current process. This ID can be passed to the SYSPROCESSNAME function to obtain the name of the current process.

Examples

Example 1: Using a DATA Step

The following DATA step writes the current process id to the SAS log:

```
data _null_;
   id=sysprocessid();
   put id;
run;
```

Example 2: Using SAS Macro Language

The following SAS Macro Language code writes the current process id to the SAS log:

```
   %let id=%sysfunc(sysprocessid());
   %put &id;
```

See Also

Functions:

- "SYSPROCESSNAME Function" on page 907

SYSPROCESSNAME Function

Returns the process name that is associated with a given process ID, or returns the name of the current process.

Category: Special

Syntax

SYSPROCESSNAME(<*process_id*>)

Required Argument

process_id
specifies a 32–character hexadecimal process id.

Details

The SYSPROCESSNAME function returns the process name associated with the process id you supply as an argument. You can use the value returned from the SYSPROCESSID function as the argument to SYSPROCESSNAME. If you omit the argument, then SYSPROCESSNAME returns the name of the current process.

You can also use the values stored in the automatic macro variables SYSPROCESSID and SYSSTARTID as arguments to SYSPROCESSNAME.

Examples

Example 1: Using SYSPROCESSNAME Without an Argument in a DATA Step

The following DATA step writes the current process name to the SAS log:

```
data _null_;
   name=sysprocessname();
   put name;
run;
```

Example 2: Using SYSPROCESSNAME With an Argument in SAS Macro Language

The following SAS Macro Language code writes the process name associated with the given process id to the SAS log:

```
%let id=&sysprocessid;
%let name=%sysfunc(sysprocessname(&id));
%put &name;
```

See Also

Functions:

- "SYSPROCESSID Function" on page 906

SYSPROD Function

Determines whether a product is licensed.

Category: Special

Syntax

SYSPROD(*product-name*)

Required Argument

product-name
specifies a character constant, variable, or expression with a value that is the name of a SAS product.

Requirement *Product-name* must be the correct official name of the product or solution.

Details

The SYSPROD function returns 1 if a specific SAS software product is licensed, 0 if it is a SAS software product but not licensed for your system, and -1 if the product name is not recognized. If SYSPROD indicates that a product is licensed, it means that the final license expiration date has not passed.

It is possible for a SAS software product to exist on your system even though the product is no longer licensed. In this case, SAS cannot access this product. Similarly, it is possible for a product to be licensed, but not installed.

Use SYSPROD in the DATA step, in an IML step, or in an SCL program.

Example

These examples determine whether a specified product is licensed.

- **x=sysprod('graph');**

  If SAS/GRAPH software is currently licensed, then SYSPROD returns a value of 1. If SAS/GRAPH software is not currently licensed, then SYSPROD returns a value of 0.

- **x=sysprod('abc');**

  SYSPROD returns a value of –1 because ABC is not a valid product name.

- **x=sysprod('base');**

  or

  x=sysprod('base sas');

  SYSPROD always returns a value of 1 because the Base product must be licensed for the SYSPROD function to run successfully.

SYSRC Function

Returns a system error number.

Categories: SAS File I/O
External Files

Syntax

SYSRC()

Details

SYSRC returns the error number for the last system error encountered by a call to one of the data set functions or external file functions.

Example

This example determines the error message if FILEREF does not exist:

```
%if %sysfunc(fileref(myfile)) ne 0 %then
   %put %sysfunc(sysrc()) - %sysfunc(sysmsg());
```

See Also

Functions:

- “FILEREF Function” on page 414

- "SYSMSG Function" on page 905

SYSTEM Function

Issues an operating environment command during a SAS session, and returns the system return code.

Category: Special

See: "SYSTEM Function: z/OS" in *SAS Companion for z/OS*

Syntax

SYSTEM(*command*)

Required Argument

command

specifies any of the following: a system command that is enclosed in quotation marks (explicit character string), an expression whose value is a system command, or the name of a character variable whose value is a system command that is executed.

Operating Environment Information

See the SAS documentation for your operating environment for information about what you can specify. The system return code is dependent on your operating environment.

Restriction The length of the command cannot be greater than 1024 characters, including trailing blanks.

Comparisons

The SYSTEM function is similar to the X statement, the X command, and the CALL SYSTEM routine. In most cases, the X statement, X command, or %SYSEXEC macro statement are preferable because they require less overhead. However, the SYSTEM function can be executed conditionally, and accepts expressions as arguments. The X statement is a global statement and executes as a DATA step is being compiled, regardless of whether SAS encounters a conditional statement.

Example

Execute the host command TIMEDATA if the macro variable SYSDAY is **`Friday`**.

```
data _null_;
   if "&sysday"="Friday" then do;
      rc=system("timedata");
   end;
   else rc=system("errorck");
run;
```

See Also

CALL Routines:

- "CALL SYSTEM Routine" on page 259

Statements:

- "X Statement" in *SAS Statements: Reference*

TAN Function

Returns the tangent.

Category: Trigonometric

Syntax

TAN(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression and is expressed in radians. If the magnitude of *argument* is so great that **`mod(argument,pi)`** is accurate to less than about three decimal places, TAN returns a missing value.

Restriction cannot be an odd multiple of $\pi/2$

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=tan(0.5);`	`0.5463024898`
`x=tan(0);`	`0`
`x=tan(3.14159/3);`	`1.7320472695`

TANH Function

Returns the hyperbolic tangent.

Category: Hyperbolic

Syntax

TANH(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Details

The TANH function returns the hyperbolic tangent of the argument, which is given by

$$\frac{\left(\varepsilon^{argument} - \varepsilon^{-argument}\right)}{\left(\varepsilon^{argument} + \varepsilon^{-argument}\right)}$$

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=tanh(0);`	`0`
`x=tanh(0.5);`	`0.4621171573`
`x=tanh(-0.5);`	`-0.462117157`

TIME Function

Returns the current time of day as a numeric SAS time value.

Category: Date and Time

Syntax

TIME()

Example

SAS assigns CURRENT a SAS time value corresponding to 14:32:00 if the following statements are executed exactly at 2:32 PM:

```
current=time();
put current=time.;
```

TIMEPART Function

Extracts a time value from a SAS datetime value.

Category: Date and Time

Syntax

TIMEPART(*datetime*)

Required Argument

datetime
is a numeric constant, variable, or expression that represents a SAS datetime value.

Example

SAS assigns TIME a SAS value that corresponds to 10:40:17 if the following statements are executed exactly at 10:40:17 a.m. on any date:

```
datim=datetime();
time=timepart(datim);
```

TIMEVALUE Function

Returns the equivalent of a reference amount at a base date by using variable interest rates.

Category: Financial

Syntax

TIMEVALUE(*base-date*, *reference-date*, *reference-amount*, *compounding-interval*, *date-1*, *rate-1* <,*date-2*, *rate-2*,...>)

Required Arguments

base-date
is a SAS date. The value that is returned is the time value of *reference-amount* at *base-date*.

reference-date
is a SAS date. *reference-date* is the date of *reference-amount*.

reference-amount
is numeric. *reference-amount* is the amount at *reference-date*.

compounding-interval
is a SAS interval. *compounding-interval* is the compounding interval.

date
is a SAS date. Each date is paired with a rate. *date* is the time that *rate* takes effect.

rate
is a numeric percentage. Each rate is paired with a date. *rate* is the interest rate that starts on *date*.

Details

The following details apply to the TIMEVALUE function:

- The values for rates must be between –99 and 120.
- The list of date-rate pairs does not need to be sorted by date.
- When multiple rate changes occur on a single date, the TIMEVALUE function applies only the final rate that is listed for that date.
- Simple interest is applied for partial periods.

- There must be a valid date-rate pair whose date is at or prior to both the *reference-date* and the *base-date*.

Example

- You can express the accumulated value of an investment of $1,000 at a nominal interest rate of 10% compounded monthly for one year as the following:

```
amount_base1 = TIMEVALUE("01jan2001"d, "01jan2000"d, 1000,
                                       "MONTH", "01jan2000"d, 10);
```

- If the interest rate jumps to 20% halfway through the year, the resulting calculation would be as follows:

```
amount_base2 = TIMEVALUE("01jan2001"d, "01jan2000"d, 1000,
                                       "MONTH", 01jan2000"d, 10, "01jan2000"d, 20);
```

- The date-rate pairs do not need to be sorted by date. This flexibility allows amount_base2 and amount_base3 to assume the same value:

```
amount_base3 = TIMEVALUE("01jan2001"d, "01jan2000"d, 1000,
                                       "MONTH", "01jul2000"d, 20, "01jan2000"d, 10);
```

TINV Function

Returns a quantile from the *t* distribution.

Category: Quantile

Syntax

TINV(*p*,*df*<,*nc*>)

Required Arguments

p
is a numeric probability.

Range $0 < p < 1$

df
is a numeric degrees of freedom parameter.

Range $df > 0$

Optional Argument

nc
is an optional numeric noncentrality parameter.

Details

The TINV function returns the p^{th} quantile from the Student's *t* distribution with degrees of freedom *df* and a noncentrality parameter *nc*. The probability that an observation from a *t* distribution is less than or equal to the returned quantile is *p*.

TINV accepts a noninteger degree of freedom parameter *df*. If the optional parameter *nc* is not specified or is 0, the quantile from the central *t* distribution is returned.

CAUTION:

For large values of nc, the algorithm can fail. In that case, a missing value is returned.

Note: TINV is the inverse of the PROBT function.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x=tinv(.95,2);`	`2.9199855804`
`x=tinv(.95,2.5,3);`	`11.033833625`

See Also

Functions:

- "QUANTILE Function" on page 799

TNONCT Function

Returns the value of the noncentrality parameter from the Student's *t* distribution.

Category: Mathematical

Syntax

TNONCT(*x*,*df*,*prob*)

Required Arguments

x

is a numeric random variable.

df

is a numeric degrees of freedom parameter.

Range $df > 0$

prob

is a probability.

Range $0 < prob < 1$

Details

The TNONCT function returns the nonnegative noncentrality parameter from a noncentral *t* distribution whose parameters are *x*, *df*, and *nc*. A Newton-type algorithm is used to find a root *nc* of the equation

$$P_t(x \mid df, nc) - prob = 0$$

The following relationship applies to the preceding equation:

$$P_t(x \mid df, nc) = \frac{1}{\Gamma\left(\frac{df}{2}\right)} \int_0^{\infty} v^{\frac{df}{2}-1} \varepsilon^{-v} \int_{-\infty}^{x\sqrt{\frac{2v}{df}}} \varepsilon^{-\frac{(u-nc)^2}{2}} du dv$$

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

Example

The following example computes the noncentrality parameter from the *t* distribution.

```
data work;
   x=2;
   df=4;
   do nc=1 to 3 by .5;
      prob=probt(x,df,nc);
      ncc=tnonct(x,df,prob);
      output;
   end;
run;
proc print;
run;
```

Display 2.69 *Computations of the Noncentrality Parameter from the t Distribution*

The SAS System

Obs	x	df	nc	prob	ncc
1	2	4	1.0	0.76457	1.00000
2	2	4	1.5	0.61893	1.50000
3	2	4	2.0	0.45567	2.00000
4	2	4	2.5	0.30115	2.50000
5	2	4	3.0	0.17702	3.00000

TODAY Function

Returns the current date as a numeric SAS date value.

Category: Date and Time

Alias: DATE

Syntax

TODAY()

Details

The TODAY function produces the current date in the form of a SAS date value, which is the number of days since January 1, 1960.

Example

These statements illustrate a practical use of the TODAY function:

```
data _null_;
   tday=today();
   if (tday-datedue)> 15 then
      do;
         put 'As of ' tday date9. ' Account #'
            account 'is more than 15 days overdue.';
      end;
run;
```

TRANSLATE Function

Replaces specific characters in a character string.

Category: Character

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

Tip: DBCS equivalent function is KTRANSLATE .

See: “TRANSLATE Function: Windows” in *SAS Companion for Windows*

“TRANSLATE Function: UNIX” in *SAS Companion for UNIX Environments*

“TRANSLATE Function: z/OS” in *SAS Companion for z/OS*

Syntax

TRANSLATE(*source*,*to-1*,*from-1*<,…*to-n*,*from-n*>)

Required Arguments

source
specifies a character constant, variable, or expression that contains the original character string.

to
specifies the characters that you want TRANSLATE to use as substitutes.

from
specifies the characters that you want TRANSLATE to replace.

Operating Environment Information
You must have pairs of *to* and *from* arguments on some operating environments. On other operating environments, a segment of the collating sequence replaces null *from* arguments. See the SAS documentation for your operating environment for more information.

Interaction Values of *to* and *from* correspond on a character-by-character basis; TRANSLATE changes the first character of *from* to the first character of *to*, and so on. If *to* has fewer characters than *from*, TRANSLATE changes the extra *from* characters to blanks. If *to* has more characters than *from*, TRANSLATE ignores the extra *to* characters.

Details

In a DATA step, if the TRANSLATE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the first argument.

The maximum number of pairs of *to* and *from* arguments that TRANSLATE accepts depends on the operating environment you use to run SAS. There is no functional difference between using several pairs of short arguments, or fewer pairs of longer arguments.

Comparisons

The TRANWRD function differs from TRANSLATE in that it scans for words (or patterns of characters) and replaces those words with a second word (or pattern of characters).

Example

The following SAS statements produce this result.

SAS Statement	Result
`x=translate('XYZW','AB','VW');` `put x;`	`XYZB`

See Also

Functions:

- “TRANWRD Function” on page 921

TRANSTRN Function

Replaces or removes all occurrences of a substring in a character string.

Category: Character

Syntax

TRANSTRN(*source*,*target*,*replacement*)

Required Arguments

source
specifies a character constant, variable, or expression that you want to translate.

target
specifies a character constant, variable, or expression that is searched for in *source*.

Requirement The length for *target* must be greater than zero.

replacement
specifies a character constant, variable, or expression that replaces *target*.

Details

Length of Returned Variable

In a DATA step, if the TRANSTRN function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. You can use the LENGTH statement, before calling TRANSTRN, to change the length of the value.

The Basics

The TRANSTRN function replaces or removes all occurrences of a given substring within a character string. The TRANSTRN function does not remove trailing blanks in the *target* string and the *replacement* string. To remove all occurrences of *target*, specify *replacement* as TRIMN("").

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. 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

These statements and these values produce these results:

```
name=transtrn(name, "Mrs.", "Ms.");
name=transtrn(name, "Miss", "Ms.");
put name;
```

Value	Result
Mrs. Joan Smith	Ms. Joan Smith
Miss Alice Cooper	Ms. Alice Cooper

Example 2: Removing Blanks from the Search String

In this example, the TRANSTRN function does not replace the source string because the target string contains blanks.

```
data list;
   input salelist $;
   length target $10 replacement $3;
   target='FISH';
   replacement='NIP';
   salelist=transtrn(salelist,target,replacement);
   put salelist;
   datalines;
CATFISH
;
```

The LENGTH statement pads *target* with blanks to the length of 10, which causes the TRANSTRN function to search for the character string **'FISH '** in SALELIST. Because the search fails, this line is written to the SAS log:

```
CATFISH
```

You can use the TRIM function to exclude trailing blanks from a target or replacement variable. Use the TRIM function with *target*:

```
salelist=transtrn(salelist,trim(target),replacement);
put salelist;
```

Now, this line is written to the SAS log:

```
CATNIP
```

Example 3: Zero Length in the Third Argument of the TRANSTRN Function

The following example shows the results of the TRANSTRN function when the third argument, *replacement*, has a length of zero. In the DATA step, a character constant that consists of two quotation marks represents a single blank, and not a zero-length string. In the following example, the results for *string1* are different from the results for *string2*.

```
data _null_;
   string1='*' || transtrn('abcxabc', 'abc', trimn('')) || '*';
   put string1=;
   string2='*' || transtrn('abcxabc', 'abc',  '') || '*';
```

```
  put string2=;
run;
```

SAS writes the following output to the log:

Log 2.23 *Output When the Third Argument of TRANSTRN Has a Length of Zero*

```
string1=*x*
string2=* x *
```

See Also

Functions:

- "TRANSLATE Function" on page 917

TRANWRD Function

Replaces all occurrences of a substring in a character string.

Category: Character

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

Syntax

TRANWRD(*source*,*target*,*replacement*)

Required Arguments

source
specifies a character constant, variable, or expression that you want to translate.

target
specifies a character constant, variable, or expression that is searched for in *source*.

Requirement The length for *target* must be greater than zero.

replacement
specifies a character constant, variable, or expression that replaces *target*. When the replacement string has a length of zero, TRANWRD uses a single blank instead.

Details

Length of Returned Variable

In a DATA step, if the TRANWRD function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes. You can use the LENGTH statement, before calling TRANWRD, to change the length of the value.

The Basics

The TRANWRD function copies the value in *source* to the result string while searching for all non-overlapping substrings in *source* that are equal to the value in *target*. Each of these substrings is omitted from the result and the value of *replacement* is copied in its

place. The TRANWRD function does not remove trailing blanks in the *target* string or the *replacement* string.

Comparisons

The TRANWRD function differs from the TRANSTRN function because TRANSTRN allows the replacement string to have a length of zero. TRANWRD uses a single blank instead when the replacement string has a length of zero.

The TRANSLATE function converts every occurrence of a user-supplied character to another character. TRANSLATE can scan for more than one character in a single call. 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 TRANWRD scans for substrings and replaces those substrings with a second substring.

Examples

Example 1: Replacing All Occurrences of a Word

These statements and these values produce these results:

```
name=tranwrd(name, "Mrs.", "Ms.");
name=tranwrd(name, "Miss", "Ms.");
put name;
```

Value	Result
Mrs. Joan Smith	Ms. Joan Smith
Miss Alice Cooper	Ms. Alice Cooper

Example 2: Removing Blanks From the Search String

In this example, the TRANWRD function does not replace the source string because the target string contains blanks.

```
data list;
   input salelist $;
   length target $10 replacement $3;
   target='FISH';
   replacement='NIP';
   salelist=tranwrd(salelist,target,replacement);
   put salelist;
   datalines;
CATFISH
;
```

The LENGTH statement pads *target* with blanks to the length of 10, which causes the TRANWRD function to search for the character string **'FISH '** in SALELIST. Because the search fails, this line is written to the SAS log:

```
CATFISH
```

You can use the TRIM function to exclude trailing blanks from a target or replacement variable. Use the TRIM function with *target*:

```
salelist=tranwrd(salelist,trim(target),replacement);
put salelist;
```

Now, this line is written to the SAS log:

```
CATNIP
```

Example 3: Zero Length in the Third Argument of the TRANWRD Function

The following example shows the results of the TRANWRD function when the third argument, *replacement*, has a length of zero. In this case, TRANWRD uses a single blank. In the DATA step, a character constant that consists of two consecutive quotation marks represents a single blank, and not a zero-length string. In this example, the results for *string1* and *string2* are the same:

```
data _null_;
   string1='*' || tranwrd('abcxabc', 'abc', trimn('')) || '*';
   put string1=;
   string2='*' || tranwrd('abcxabc', 'abc', '') || '*';
   put string2=;
run;
```

Log 2.24 *Output When the Third Argument of TRANWRD Has a Length of Zero*

```
string1=* x *
string2=* x *
```

Example 4: Removing Repeated Commas

You can use the TRANWRD function to remove repeated commas in text, and replace the repeated commas with a single comma. In the following example, the TRANWRD function is used twice: to replace three commas with one comma, and to replace the ending two commas with a period:

```
data _null_;
   mytxt='If you exercise your power to vote,,,then your opinion will be heard,,';
   newtext=tranwrd(mytxt, ',,,', ',');
   newtext2=tranwrd(newtext, ',,' , '.');
   put // mytxt= / newtext= / newtext2=;
run;
```

Log 2.25 *Output from Removing Repeated Commas*

```
mytxt=If you exercise your power to vote,,,then your opinion will be heard,,
newtext=If you exercise your power to vote,then your opinion will be heard,,
newtext2=If you exercise your power to vote,then your opinion will be heard.
```

See Also

Functions:

- "TRANSLATE Function" on page 917

TRIGAMMA Function

Returns the value of the trigamma function.

Category: Mathematical

Syntax

TRIGAMMA(*argument*)

Required Argument

argument
specifies a numeric constant, variable, or expression.

Restriction Nonpositive integers are invalid.

Details

The TRIGAMMA function returns the derivative of the DIGAMMA function. For *argument* > 0, the TRIGAMMA function is the second derivative of the LGAMMA function.

Example

The following SAS statement produces this result.

SAS Statement	Result
`x=trigamma(3);`	`0.3949340668`

TRIM Function

Removes trailing blanks from a character string, and returns one blank if the string is missing.

Category: Character

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

Tip: DBCS equivalent function is “KTRIM Function” in *SAS National Language Support (NLS): Reference Guide*.

Syntax

TRIM(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable

In a DATA step, if the TRIM function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

The Basics

TRIM copies a character argument, removes trailing blanks, and returns the trimmed argument as a result. If the argument is blank, TRIM returns one blank. TRIM is useful for concatenating because concatenation does not remove trailing blanks.

Assigning the results of TRIM to a variable does not affect the length of the receiving variable. If the trimmed value is shorter than the length of the receiving variable, SAS pads the value with new blanks as it assigns it to the variable.

Comparisons

The TRIM and TRIMN functions are similar. TRIM returns one blank for a blank string. TRIMN returns a string with a length of zero for a blank string.

Examples

Example 1: Removing Trailing Blanks

These statements and this data line produce these results:

```
data test;
   input part1 $ 1-10 part2 $ 11-20;
   hasblank=part1||part2;
   noblank=trim(part1)||part2;
   put hasblank;
   put noblank;
   datalines;
```

Data Line	Result
	`----+----1----+----2`
`apple sauce`	`apple sauce`
	`applesauce`

Example 2: Concatenating a Blank Character Expression

SAS Statement	Result
`x="A"||trim(" ")||"B"; put x;`	`A B`

SAS Statement	Result
`x="    "; y=">"||trim(x)||"<"; put y;`	`> <`

See Also

Functions:

- "COMPRESS Function" on page 327
- "LEFT Function" on page 616
- "RIGHT Function" on page 832
- "TRIMN Function" on page 926

TRIMN Function

Removes trailing blanks from character expressions, and returns a string with a length of zero if the expression is missing.

Category: Character

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

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

The following SAS statements produce these results.

SAS Statement	Result
`x="A"||trimn("")||"B";` `put x;`	`AB`

`x="   ";`
`z=">"||trimn(x)||"<";`
`put z;`
	`><`

See Also

Functions:

- “COMPRESS Function” on page 327
- “LEFT Function” on page 616
- “RIGHT Function” on page 832
- “TRIM Function” on page 924

TRUNC Function

Truncates a numeric value to a specified number of bytes.

Category: Truncation

Syntax

TRUNC(*number*,*length*)

Required Arguments

number
specifies a numeric constant, variable, or expression.

length
specifies an integer.

Details

The TRUNC function truncates a full-length *number* (stored as a double) to a smaller number of bytes, as specified in *length* and pads the truncated bytes with 0s. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them.

Example

```
data test;
  length x 3;
  x=1/5;
run;
data test2;
  set test;
  if x ne 1/5 then
     put 'x ne 1/5';
  if x eq trunc(1/5,3) then
     put 'x eq trunc(1/5,3)';
run;
```

The variable X is stored with a length of 3 and, therefore, each of the above comparisons is true.

UNIFORM Function

Returns a random variate from a uniform distribution.

Category: Random Number

Alias: RANUNI

See: "RANUNI Function" on page 826

UPCASE Function

Converts all letters in an argument to uppercase.

Category: Character

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

Syntax

UPCASE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

In a DATA step, if the UPCASE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

The UPCASE function copies a character argument, converts all lowercase letters to uppercase letters, and returns the altered value as a result.

The results of the UPCASE function depend directly on the translation table that is in effect (see TRANTAB system option), and indirectly on the ENCODING system option and the LOCALE system option.

Example

The following SAS statements produce this result.

SAS Statement	Result
`name=upcase('John B. Smith');` `put name;`	`JOHN B. SMITH`

See Also

Functions:

- “LOWCASE Function” on page 646
- “PROPCASE Function” on page 773

URLDECODE Function

Returns a string that was decoded using the URL escape syntax.

Category: Web Tools

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

Syntax

URLDECODE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable in a DATA Step

If the URLDECODE function returns a value to a variable that has not previously been assigned a length, then that variable is given the length of the argument.

The Basics

The URL escape syntax is used to hide characters that might otherwise be significant when used in a URL.

A URL escape sequence can be one of the following:

- a plus sign, which is replaced by a blank
- a sequence of three characters beginning with a percent sign and followed by two hexadecimal characters, which is replaced by a single character that has the specified hexadecimal value.

argument can be decoded using either SAS session encoding or UTF-8 encoding. To decode *argument* by using the SAS session encoding, set the system option

URLENCODING=SESSION. To decode *argument* by using UTF-8 encoding, set the system option URLENCODING=UTF8.

Operating Environment Information
In operating environments that use EBCDIC, SAS performs an extra translation step after it recognizes an escape sequence. The specified character is assumed to be an ASCII encoding. SAS uses the transport-to-local translation table to convert this character to an EBCDIC character in operating environments that use EBCDIC. For more information, see the TRANTAB option .

Example

The following SAS statements produce these results using SAS session decoding.

SAS Statement	Result
`x1=urldecode ('abc+def');` `put x1;`	`abc def`
`x2=urldecode ('why%3F');` `put x2;`	`why?`
`x3=urldecode ('%41%42%43%23%31');` `put x3;`	`ABC#1`

See Also

Functions:

- “URLENCODE Function” on page 930

System Options:

- “URLENCODING= System Option” in *SAS System Options: Reference*

URLENCODE Function

Returns a string that was encoded using the URL escape syntax.

Category: Web Tools

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

Syntax

URLENCODE(*argument*)

Required Argument

argument
specifies a character constant, variable, or expression.

Details

Length of Returned Variable in a DATA Step

If the URLENCODE function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes.

The Basics

argument can be encoded using either SAS session encoding or UTF-8 encoding. To encode *argument* by using the SAS session encoding, set the system option URLENCODING=SESSION. To encode *argument* by using UTF-8 encoding, set the system option URLENCODING=UTF8.

The URLENCODE function encodes characters that might otherwise be significant when used in a URL. This function encodes all characters except for the following:

- all alphanumeric characters
- dollar sign ($)
- hyphen (-)
- underscore (_)
- at sign (@)
- period (.)
- exclamation point (!)
- asterisk (*)
- open parenthesis (( )and close parenthesis ( ))
- comma (,).

Note: The encoded string might be longer than the original string. Ensure that you consider the additional length when you use this function.

Example

The following SAS statements produce these results using SAS session encoding.

SAS Statement	Result
`x1=urlencode ('abc def');` `put x1;`	`abc%20def`
`x2=urlencode ('why?');` `put x2;`	`why%3F`
`x3=urlencode ('ABC#1');` `put x3;`	`ABC%231`

See Also

Functions:

- “URLDECODE Function” on page 929

System Options:

- "URLENCODING= System Option" in *SAS System Options: Reference*

USS Function

Returns the uncorrected sum of squares of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

USS(*argument-1*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least one nonmissing argument is required. Otherwise, the function returns a missing value. If you have more than one argument, the argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
`x1=uss(4,2,3.5,6);`	`68.25`
`x2=uss(4,2,3.5,6,.);`	`68.25`
`x3=uss(of x1-x2);`	`9316.125`

UUIDGEN Function

Returns the short or binary form of a Universal Unique Identifier (UUID).

Category: Special

Syntax

UUIDGEN(<*max-warnings*<,*binary-result*> >)

Required Argument

max-warnings
specifies an integer value that represents the maximum number of warnings that this function writes to the log.

Default 1

Optional Argument

binary-result
specifies an integer value that indicates whether this function should return a binary result. Nonzero indicates a binary result should be returned. Zero indicates that a character result should be returned.

Default 0

Details

Length of Returned Variable in a DATA Step

If the UUIDGEN function returns a value to a variable that has not previously been assigned a length, then that variable is given a length of 200 bytes.

The Basics

The UUIDGEN function returns a UUID (a unique value) for each call. The default result is 36 characters long and it looks like this:

```
5ab6fa40-426b-4375-bb22-2d0291f43319
```

A binary result is 16 bytes long.

See Also

Universal Unique Identifiers:

- "Universal Unique Identifiers" in Chapter 39 of *SAS Language Reference: Concepts*

VAR Function

Returns the variance of the nonmissing arguments.

Category: Descriptive Statistics

Syntax

VAR(*argument-1*,*argument-2*<,...*argument-n*>)

Required Argument

argument
specifies a numeric constant, variable, or expression. At least two nonmissing arguments are required. Otherwise, the function returns a missing value. The argument list can consist of a variable list, which is preceded by OF.

Example

The following SAS statements produce these results.

SAS Statement	Result
x1=var(4,2,3.5,6);	2.7291666667
x2=var(4,6,.);	2
x3=var(of x1-x2);	0.2658420139

VARFMT Function

Returns the format that is assigned to a SAS data set variable.

Category: SAS File I/O

Syntax

VARFMT(*data-set-id*,*var-num*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

If no format has been assigned to the variable, a blank string is returned.

Examples

Example 1: Using VARFMT to Obtain the Format of the Variable NAME

This example obtains the format of the variable NAME in the SAS data set MYDATA.

```
%let dsid=%sysfunc(open(mydata,i));
%if &dsid %then
   %do;
      %let fmt=%sysfunc(varfmt(&dsid,
                       %sysfunc(varnum
                       (&dsid,NAME))));
      %let rc=%sysfunc(close(&dsid));
   %end;
```

Example 2: Using VARFMT to Obtain the Format of all the Numeric Variables in a Data Set

This example creates a data set that contains the name and formatted content of each numeric variable in the SAS data set MYDATA.

```
data vars;
   length name $ 8 content $ 12;
   drop dsid i num rc fmt;
   dsid=open("mydata","i");
   num=attrn(dsid,"nvars");
   do while (fetch(dsid)=0);
      do i=1 to num;
         name=varname(dsid,i);
         if (vartype(dsid,i)='N') then do;
            fmt=varfmt(dsid,i);
            if fmt='' then fmt="BEST12.";
            content=putc(putn(getvarn
                        (dsid,i),fmt),"$char12.");
            output;
            end;
      end;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- “VARINFMT Function” on page 935
- “VARNUM Function” on page 940

VARINFMT Function

Returns the informat that is assigned to a SAS data set variable.

Category: SAS File I/O

Syntax

VARINFMT(*data-set-id*,*var-num*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

If no informat has been assigned to the variable, a blank string is returned.

Examples

Example 1: Using VARINFMT to Obtain the Informat of the Variable NAME

This example obtains the informat of the variable NAME in the SAS data set MYDATA.

```
%let dsid=%sysfunc(open(mydata,i));
%if &dsid %then
   %do;
      %let fmt=%sysfunc(varinfmt(&dsid,
                        %sysfunc(varnum
                           (&dsid,NAME))));
      %let rc=%sysfunc(close(&dsid));
   %end;
```

Example 2: Using VARINFMT to Obtain the Informat of all the Variables in a Data Set

This example creates a data set that contains the name and informat of the variables in MYDATA.

```
data vars;
   length name $ 8 informat $ 10 ;
   drop dsid i num rc;
   dsid=open("mydata","i");
   num=attrn(dsid,"nvars");
   do i=1 to num;
      name=varname(dsid,i);
      informat=varinfmt(dsid,i);
      output;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- “OPEN Function” on page 716
- “VARFMT Function” on page 934
- “VARNUM Function” on page 940

VARLABEL Function

Returns the label that is assigned to a SAS data set variable.

Category: SAS File I/O

Syntax

VARLABEL(*data-set-id*,*var-num*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

If no label has been assigned to the variable, a blank string is returned.

Comparisons

VLABEL returns the label that is associated with the given variable.

Example

This example obtains the label of the variable NAME in the SAS data set MYDATA.

Example Code 2.1 *Obtaining the Label of the Variable NAME*

```
%let dsid=%sysfunc(open(mydata,i));
%if &dsid %then
   %do;
      %let fmt=%sysfunc(varlabel(&dsid,
                      %sysfunc(varnum
                              (&dsid,NAME))));
      %let rc=%sysfunc(close(&dsid));
   %end;
```

See Also

Functions:

- "OPEN Function" on page 716
- "VARNUM Function" on page 940

VARLEN Function

Returns the length of a SAS data set variable.

Category: SAS File I/O

Syntax

VARLEN(*data-set-id*,*var-num*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

VLENGTH returns the compile-time (allocated) size of the given variable.

Example

- This example obtains the length of the variable ADDRESS in the SAS data set MYDATA.

```
%let dsid=%sysfunc(open(mydata,i));
%if &dsid %then
   %do;
      %let len=%sysfunc(varlen(&dsid,
                      %sysfunc(varnum
                      (&dsid,ADDRESS))));
      %let rc=%sysfunc(close(&dsid));
   %end;
```

- This example creates a data set that contains the name, type, and length of the variables in MYDATA.

```
data vars;
   length name $ 8 type $ 1;
   drop dsid i num rc;
   dsid=open("mydata","i");
   num=attrn(dsid,"nvars");
   do i=1 to num;
      name=varname(dsid,i);
      type=vartype(dsid,i);
      length=varlen(dsid,i);
      output;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- "OPEN Function" on page 716
- "VARNUM Function" on page 940

VARNAME Function

Returns the name of a SAS data set variable.

Category: SAS File I/O

Syntax

VARNAME(*data-set-id*,*var-num*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Example

This example copies the names of the first five variables in the SAS data set CITY (or all of the variables if there are fewer than five) into a macro variable.

```
%macro names;
   %let dsid=%sysfunc(open(city,i));
   %let varlist=;
   %do i=1 %to
      %sysfunc(min(5,%sysfunc(attrn
                           (&dsid,nvars))));
      %let varlist=&varlist %sysfunc(varname
                                (&dsid,&i));
   %end;
   %put varlist=&varlist;
%mend names;
%names
```

See Also

Functions:

- "OPEN Function" on page 716
- "VARNUM Function" on page 940

VARNUM Function

Returns the number of a variable's position in a SAS data set.

Category: SAS File I/O

Syntax

VARNUM(*data-set-id*,*var-name*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-name
specifies the variable's name.

Details

VARNUM returns the number of a variable's position in a SAS data set, or 0 if the variable is not in the SAS data set. This is the same variable number that is next to the variable in the output from PROC CONTENTS.

Example

- This example obtains the number of a variable's position in the SAS data set CITY, given the name of the variable.

```
%let dsid=%sysfunc(open(city,i));
%let citynum=%sysfunc(varnum(&dsid,CITYNAME));
%let rc=%sysfunc(fetch(&dsid));
%let cityname=%sysfunc(getvarc
                      (&dsid,&citynum));
```

- This example creates a data set that contains the name, type, format, informat, label, length, and position of the variables in SASUSER.HOUSES.

```
/* Tested 2/27/98 – OK */
%INCLUDE '/local/u/lirezn/sasuser/assist.src';

data vars;
   length name $ 8 type $ 1
          format informat $ 10 label $ 40;
   drop dsid i num rc;
   dsid=open("sasuser.houses","i");
   num=attrn(dsid,"nvars");
   do i=1 to num;
      name=varname(dsid,i);
      type=vartype(dsid,i);
      format=varfmt(dsid,i);
      informat=varinfmt(dsid,i);
      label=varlabel(dsid,i);
      length=varlen(dsid,i);
      position=varnum(dsid,name);
```

```
      output;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- "OPEN Function" on page 716
- "VARNAME Function" on page 939

VARRAY Function

Returns a value that indicates whether the specified name is an array.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VARRAY (*name*)

Required Argument

name
specifies a name that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

VARRAY returns 1 if the given name is an array; it returns 0 if the given name is not an array.

Comparisons

- VARRAY returns a value that indicates whether the specified name is an array. VARRAYX returns a value that indicates whether the value of the specified expression is an array.
- VARRAY does not accept an expression as an argument. VARRAYX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`array x(3) x1-x3;`
`a=varray(x);`
`B=varray(x1);`
`put a=;`
`put B=;`
	`a=1`
	`B=0`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VARRAYX Function

Returns a value that indicates whether the value of the specified argument is an array.

Category: Variable Information

Syntax

VARRAYX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression.

Restriction The value of the specified expression cannot denote an array reference.

Details

VARRAYX returns 1 if the value of the given argument is the name of an array; it returns 0 if the value of the given argument is not the name of an array.

Comparisons

- VARRAY returns a value that indicates whether the specified name is the name of an array. VARRAYX returns a value that indicates whether the value of the specified expression is the name of an array.
- VARRAY does not accept an expression as an argument. VARRAYX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

array x(3) x1-x3;
array vx(4) $6 vx1 vx2 vx3 vx4
('x' 'x1' 'x2' 'x3');
y=varrayx(vx(1));
z=varrayx(vx(2));
put y=;
put z=;
	y=1
	z=0

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VARTYPE Function

Returns the data type of a SAS data set variable.

Category: SAS File I/O

Syntax

VARTYPE(*data-set-id*,*var-num*)

Required Arguments

data-set-id
specifies the data set identifier that the OPEN function returns.

var-num
specifies the number of the variable's position in the SAS data set.

Tips This number is next to the variable in the list that is produced by the CONTENTS procedure.

The VARNUM function returns this number.

Details

VARTYPE returns C for a character variable or N for a numeric variable.

Examples

Example 1: Using VARTYPE to Determine Which Variables Are Numeric

This example places the names of all the numeric variables of the SAS data set MYDATA into a macro variable.

```
%let dsid=%sysfunc(open(mydata,i));
```

```
%let varlist=;
%do i=1 %to %sysfunc(attrn(&dsid,nvars));
  %if (%sysfunc(vartype(&dsid,&i)) = N) %then
     %let varlist=&varlist %sysfunc(varname
                                  (&dsid,&i));
%end;
%let rc=%sysfunc(close(&dsid));
```

Example 2: Using VARTYPE to Determine Which Variables Are Character

This example creates a data set that contains the name and formatted contents of each character variable in the SAS data set MYDATA.

```
data vars;
   length name $ 8 content $ 20;
   drop dsid i num fmt rc;
   dsid=open("mydata","i");
   num=attrn(dsid,"nvars");
   do while (fetch(dsid)=0);
      do i=1 to num;
         name=varname(dsid,i);
         fmt=varfmt(dsid,i);
         if (vartype(dsid,i)='C') then do;
            content=getvarc(dsid,i);
            if (fmt ne '' ) then
             content=left(putc(content,fmt));
            output;
            end;
      end;
   end;
   rc=close(dsid);
run;
```

See Also

Functions:

- "VARNUM Function" on page 940

VERIFY Function

Returns the position of the first character in a string that is not in any of several other strings.

Category: Character

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

Tip: DBCS equivalent function is KVERIFY .

Syntax

VERIFY(*source*,*excerpt-1*<, …, *excerpt-n*>)

Required Arguments

source
specifies a character constant, variable, or expression.

excerpt
specifies a character constant, variable, or expression. If you specify more than one excerpt, separate them with a comma.

Details

The VERIFY function returns the position of the first character in *source* that is not present in any *excerpt*. If VERIFY finds every character in *source* in at least one *excerpt*, it returns a 0.

Example

The following SAS statements produce this result.

SAS Statement	Result

`data scores;`
`  input Grade : $1. @@;`
`  check='abcdf';`
`  if verify(grade,check)>0 then`
`  put @1 'INVALID ' grade=;`
`  datalines;`
`a b c b c d f a a q a b d d b`
`;`
	`INVALID Grade=q`

See Also

Functions:

- "FINDC Function" on page 459

VFORMAT Function

Returns the format that is associated with the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VFORMAT (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VFORMAT function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VFORMAT returns the complete format name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VFORMAT returns the format that is associated with the specified variable. VFORMATX, however, evaluates the argument to determine the variable name. The function then returns the format that is associated with that variable name.
- VFORMAT does not accept an expression as an argument. VFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 best6.;`
`y=vformat(x(1));`
`put y=;`
	`y=BEST6.`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VFORMATD Function

Returns the decimal value of the format that is associated with the specified variable.

Category: Variable Information

Syntax

VFORMATD (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VFORMATD returns the format decimal value that is associated with the specified variable. VFORMATDX, however, evaluates the argument to determine the variable name. The function then returns the format decimal value that is associated with that variable name.
- VFORMATD does not accept an expression as an argument. VFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 comma8.2;`
`y=vformatd(x(1));`
`put y=;`
	`y=2`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VFORMATDX Function

Returns the decimal value of the format that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VFORMATDX (*expression*)

Required Argument

expression
specifies a SAS character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

- VFORMATD returns the format decimal value that is associated with the specified variable. VFORMATDX, however, evaluates the argument to determine the variable name. The function then returns the format decimal value that is associated with that variable name.
- VFORMATD does not accept an expression as an argument. VFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 comma8.2;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vformatdx(vx(1));`
`z=vformatdx('x'||'1');`
`put y=;`
`put z=;`
	`y=2`
	`z=2`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VFORMATN Function

Returns the format name that is associated with the specified variable.

Category: Variable Information

Syntax

VFORMATN (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VFORMATN function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VFORMATN returns only the format name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VFORMATN returns the format name that is associated with the specified variable. VFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the format name that is associated with that variable name.
- VFORMATN does not accept an expression as an argument. VFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 best6.;`
`y=vformatn(x(1));`
`put y=;`
	`y=BEST`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VFORMATNX Function

Returns the format name that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VFORMATNX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VFORMATNX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VFORMATNX returns only the format name, which does not include the length or the period (for example, $CHAR).

Comparisons

- VFORMATN returns the format name that is associated with the specified variable. VFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the format name that is associated with that variable name.
- VFORMATN does not accept an expression as an argument. VFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 best6.;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vformatnx(vx(1));`
`put y=;`
	`y=BEST`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VFORMATW Function

Returns the format width that is associated with the specified variable.

Category: Variable Information

Syntax

VFORMATW (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VFORMATW returns the format width that is associated with the specified variable. VFORMATWX, however, evaluates the argument to determine the variable name. The function then returns the format width that is associated with that variable name.
- VFORMATW does not accept an expression as an argument. VFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 best6.;`
`y=vformatw(x(1));`
`put y=;`
	`y=6`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VFORMATWX Function

Returns the format width that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VFORMATWX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Comparisons

- VFORMATW returns the format width that is associated with the specified variable. VFORMATWX, however, evaluates the argument to determine the variable name. The function then returns the format width that is associated with that variable name.
- VFORMATW does not accept an expression as an argument. VFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 best6.;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vformatwx(vx(1));`
`put y=;`
	`y=6`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VFORMATX Function

Returns the format that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VFORMATX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VFORMATX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VFORMATX returns the complete format name which includes the width and the period (for example, $CHAR20.).

Comparisons

- VFORMAT returns the format that is associated with the specified variable. VFORMATX, however, evaluates the argument to determine the variable name. The function then returns the format that is associated with that variable name.
- VFORMAT does not accept an expression as an argument. VFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`array x(3) x1-x3;`
`format x1 best6.;`
`format x2 20.10;`
`array vx(3) $6 vx1 vx2 vx3`
`('x1' 'x2' 'x3');`
`y=vformatx(vx(1));`
`z=vformatx(vx(2));`
`put y=;`
`put z=;`
	`y=BEST6.`
	`z=F20.10`

See Also

Functions:

- Variable Information functions in"SAS Functions and CALL Routines by Category" on page 65

VINARRAY Function

Returns a value that indicates whether the specified variable is a member of an array.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VINARRAY (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

VINARRAY returns 1 if the given variable is a member of an array; it returns 0 if the given variable is not a member of an array.

Comparisons

- VINARRAY returns a value that indicates whether the specified variable is a member of an array. VINARRAYX, however, evaluates the argument to determine the variable name. The function then returns a value that indicates whether the variable name is a member of an array.
- VINARRAY does not accept an expression as an argument. VINARRAYX accepts expressions, but the value of the specified expression cannot denote an array reference.

- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`array x(3) x1-x3;`
`y=vinarray(x);`
`Z=vinarray(x1);`
`put y=;`
`put Z=;`
	`y=0`
	`z=1`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINARRAYX Function

Returns a value that indicates whether the value of the specified argument is a member of an array.

Category: Variable Information

Syntax

VINARRAYX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

VINARRAYX returns 1 if the value of the given argument is a member of an array; it returns 0 if the value of the given argument is not a member of an array.

Comparisons

- VINARRAY returns a value that indicates whether the specified variable is a member of an array. VINARRAYX, however, evaluates the argument to determine the variable name. The function then returns a value that indicates whether the variable name is a member of an array.

- VINARRAY does not accept an expression as an argument. VINARRAYX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`array x(3) x1-x3;`
`array vx(4) $6 vx1 vx2 vx3 vx4`
`  ('x' 'x1' 'x2' 'x3');`
`y=vinarrayx(vx(1));`
`z=vinarrayx(vx(2));`
`put y=;`
`put z=;`
	`y=0`
	`z=1`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMAT Function

Returns the informat that is associated with the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VINFORMAT (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VINFORMAT function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VINFORMAT returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VINFORMAT returns the informat that is associated with the specified variable. VINFORMATX, however, evaluates the argument to determine the variable name. The function then returns the informat that is associated with that variable name.
- VINFORMAT does not accept an expression as an argument. VINFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x $char6.;`
`input x;`
`y=vinformat(x);`
`put y=;`
	`y=$CHAR6.`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMATD Function

Returns the decimal value of the informat that is associated with the specified variable.

Category: Variable Information

Syntax

VINFORMATD (*var*)

Required Argument

var

specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VINFORMATD returns the informat decimal value that is associated with the specified variable. VINFORMATDX, however, evaluates the argument to determine the variable name. The function then returns the informat decimal value that is associated with that variable name.
- VINFORMATD does not accept an expression as an argument. VINFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x comma8.2;`
`input x;`
`y=vinformatd(x);`
`put y=;`
	`y=2`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMATDX Function

Returns the decimal value of the informat that is associated with the value of the specified variable.

Category: Variable Information

Syntax

VINFORMATDX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified variable cannot denote an array reference.

Comparisons

- VINFORMATD returns the informat decimal value that is associated with the specified variable. VINFORMATDX, however, evaluates the argument to determine the variable name. The function then returns the informat decimal value that is associated with that variable name.
- VINFORMATD does not accept an expression as an argument. VINFORMATDX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x1 x2 x3 comma9.3;`
`input x1 x2 x3;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vinformatdx(vx(1));`
`put y=;`
	`y=3`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMATN Function

Returns the informat name that is associated with the specified variable.

Category: Variable Information

Syntax

VINFORMATN (*var*)

Required Argument

var

specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VINFORMATN function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VINFORMATN returns only the informat name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VINFORMATN returns the informat name that is associated with the specified variable. VINFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the informat name that is associated with that variable name.
- VINFORMATN does not accept an expression as an argument. VINFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x $char6.;`
`input x;`
`y=vinformatn(x);`
`put y=;`
	`y=$CHAR`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMATNX Function

Returns the informat name that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VINFORMATNX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VINFORMATNX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VINFORMATNX returns only the informat name, which does not include the width or the period (for example, $CHAR).

Comparisons

- VINFORMATN returns the informat name that is associated with the specified variable. VINFORMATNX, however, evaluates the argument to determine the variable name. The function then returns the informat name that is associated with that variable name.
- VINFORMATN does not accept an expression as an argument. VINFORMATNX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x1 x2 x3 $char6.;`
`input x1 x2 x3;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vinformatnx(vx(1));`
`put y=;`
	`y=$CHAR`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMATW Function

Returns the informat width that is associated with the specified variable.

Category: Variable Information

Syntax

VINFORMATW (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- VINFORMATW returns the informat width that is associated with the specified variable. VINFORMATWX, however, evaluates the argument to determine the variable name. The function then returns the informat width that is associated with that variable name.
- VINFORMATW does not accept an expression as an argument. VINFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, type, and length, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x $char6.;`
`input x;`
`y=vinformatw(x);`
`put y=;`
	`y=6`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMATWX Function

Returns the informat width that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VINFORMATWX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Comparisons

- VINFORMATW returns the informat width that is associated with the specified variable. VINFORMATWX, however, evaluates the argument to determine the variable name. The function then returns the informat width that is associated with that variable name.
- VINFORMATW does not accept an expression as an argument. VINFORMATWX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x1 x2 x3 $char6.;`
`input x1 x2 x3;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vinformatwx(vx(1));`
`put y=;`
	`y=6`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VINFORMATX Function

Returns the informat that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VINFORMATX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VINFORMATX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VINFORMATX returns the complete informat name, which includes the width and the period (for example, $CHAR20.).

Comparisons

- VINFORMAT returns the informat that is associated with the specified variable. VINFORMATX, however, evaluates the argument to determine the variable name. The function then returns the informat that is associated with that variable name.
- VINFORMAT does not accept an expression as an argument. VINFORMATX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, length, and type, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`informat x1 x2 x3 $char6.;`
`input x1 x2 x3;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vinformatx(vx(1));`
`put y=;`
	`y=$CHAR6.`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VLABEL Function

Returns the label that is associated with the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VLABEL (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VLABEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

If there is no label, VLABEL returns the variable name.

Comparisons

- VLABEL returns the label of the specified variable or the name of the specified variable, if no label exists. VLABELX, however, evaluates the argument to determine the variable name. The function then returns the label that is associated with that variable name, or the variable name if no label exists.
- VLABEL does not accept an expression as an argument. VLABELX accepts expressions, but the value of the specified expression cannot denote an array reference.
- VLABEL has the same functionality as CALL LABEL.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statements	Results

`array x(3) x1-x3;`
`label x1='Test1';`
`y=vlabel(x(1));`
`put y=;`
	`y=Test1`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VLABELX Function

Returns the label that is associated with the value of the specified argument.

Category: Variable Information

Syntax

VLABELX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VLABELX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

If there is no label, VLABELX returns the variable name.

Comparisons

- VLABEL returns the label of the specified variable, or the name of the specified variable if no label exists. VLABELX, however, evaluates the argument to determine the variable name. The function then returns the label that is associated with that variable name, or the variable name if no label exists.
- VLABEL does not accept an expression as an argument. VLABELX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`label x1='Test1';`
`y=vlabelx(vx(1));`
`put y=;`
	`y=Test1`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VLENGTH Function

Returns the compile-time (allocated) size of the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VLENGTH (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Comparisons

- LENGTH examines the variable at run-time, trimming trailing blanks to determine the length. VLENGTH returns a compile-time constant value, which reflects the maximum length.
- LENGTHC returns the same value as VLENGTH, but LENGTHC can be used in any calling environment and its argument can be any expression.
- VLENGTH returns the length of the specified variable. VLENGTHX, however, evaluates the argument to determine the variable name. The function then returns the compile-time size that is associated with that variable name.
- VLENGTH does not accept an expression as an argument. VLENGTHX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`length x $8;`
`x='abc';`
`y=vlength(x);`
`z=length(x);`
`put y=;`
`put z=;`
	`y=8`
	`z=3`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VLENGTHX Function

Returns the compile-time (allocated) size for the variable that has a name that is the same as the value of the argument.

Category: Variable Information

Syntax

VLENGTHX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Comparisons

- LENGTH examines the variable at run-time, trimming trailing blanks to determine the length. VLENGTHX, however, evaluates the argument to determine the variable name. The function then returns the compile-time size that is associated with that variable name.
- LENGTHC accepts an expression as the argument, but it returns the length of the value of the expression, not the length of the variable that has a name equal to the value of the expression.
- VLENGTH returns the length of the specified variable. VLENGTHX returns the length for the value of the specified expression.

- VLENGTH does not accept an expression as an argument. VLENGTHX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`length x1 $8;`
`x1='abc';`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vlengthx(vx(1));`
`z=length(x1);`
`put y=;`
`put z=;`
	`y=8`
	`z=3`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VNAME Function

Returns the name of the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VNAME (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

Comparisons

- VNAME returns the name of the specified variable. VNAMEX, however, evaluates the argument to determine a variable name. If the name is a known variable name, the function returns that name. Otherwise, the function returns a blank.
- VNAME does not accept an expression as an argument. VNAMEX accepts expressions, but the value of the specified expression cannot denote an array reference.
- VNAME has the same functionality as CALL VNAME.
- Related functions return the value of other variable attributes, such as the variable label, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`y=vname(x(1));`
`put y=;`
	`y=x1`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VNAMEX Function

Validates the value of the specified argument as a variable name.

Category: Variable Information

Syntax

VNAMEX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VNAMEX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

Comparisons

- VNAME returns the name of the specified variable. VNAMEX, however, evaluates the argument to determine a variable name. If the name is a known variable name, the function returns that name. Otherwise, the function returns a blank.
- VNAME does not accept an expression as an argument. VNAMEX accepts expressions, but the value of the specified variable cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable label, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce these results.

SAS Statement	Result

`array x(3) x1-x3;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vnamex(vx(1));`
`z=vnamex('x'||'1');`
`put y=;`
`put z=;`
	`y=x1`
	`z=x1`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VTYPE Function

Returns the type (character or numeric) of the specified variable.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VTYPE (*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VTYPE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 1.

VTYPE returns N for numeric variables and C for character variables.

Comparisons

- VTYPE returns the type of the specified variable. VTYPEX, however, evaluates the argument to determine the variable name. The function then returns the type (character or numeric) that is associated with that variable name.
- VTYPE does not accept an expression as an argument. VTYPEX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`y=vtype(x(1));`
`put y=;`
	`y=N`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VTYPEX Function

Returns the type (character or numeric) for the value of the specified argument.

Category: Variable Information

Syntax

VTYPEX (*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VTYPEX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 1.

VTYPEX returns N for numeric variables and C for character variables.

Comparisons

- VTYPE returns the type of the specified variable. VTYPEX, however, evaluates the argument to determine the variable name. The function then returns the type (character or numeric) that is associated with that variable name.
- VTYPE does not accept an expression as an argument. VTYPEX accepts expressions, but the value of the specified expression cannot denote an array reference.
- Related functions return the value of other variable attributes, such as the variable name, informat, and format, among others. For a list, see the Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65.

Example

The following SAS statements produce this result.

SAS Statement	Result

`array x(3) x1-x3;`
`array vx(3) $6 vx1 vx2 vx3`
`  ('x1' 'x2' 'x3');`
`y=vtypex(vx(1));`
`put y=;`
	`y=N`

See Also

Functions:

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VVALUE Function

Returns the formatted value that is associated with the variable that you specify.

Category: Variable Information

Restriction: Use only with the DATA step

Syntax

VVALUE(*var*)

Required Argument

var
specifies a variable that is expressed as a scalar or as an array reference.

Restriction You cannot use an expression as an argument.

Details

If the VVALUE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VVALUE returns a character string that contains the current value of the variable that you specify. The value is formatted using the current format that is associated with the variable.

Comparisons

- VVALUE returns the value that is associated with the variable that you specify. VVALUEX, however, evaluates the argument to determine the variable name. The function then returns the value that is associated with that variable name.
- VVALUE does not accept an expression as an argument. VVALUEX accepts expressions, but the value of the expression cannot denote an array reference.
- VVALUE and an assignment statement both return a character string that contains the current value of the variable that you specify. With VVALUE, the value is formatted using the current format that is associated with the variable. With an assignment statement, however, the value is unformatted.
- The PUT function allows you to reformat a specified variable or constant. VVALUE uses the current format that is associated with the variable.

Example

The following SAS statements produce this result.

SAS Statement	Result

`y=9999;`
`format y comma10.2;`
`v=vvalue(y);`
`put v;`
	`9,999.00`

See Also

Functions:

- “VVALUEX Function” on page 975

- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

VVALUEX Function

Returns the formatted value that is associated with the argument that you specify.

Category: Variable Information

Syntax

VVALUEX(*expression*)

Required Argument

expression
specifies a character constant, variable, or expression that evaluates to a variable name.

Restriction The value of the specified expression cannot denote an array reference.

Details

If the VVALUEX function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 200.

VVALUEX returns a character string that contains the current value of the argument that you specify. The value is formatted by using the format that is currently associated with the argument.

Comparisons

- VVALUE accepts a variable as an argument and returns the value of that variable. VVALUEX, however, accepts a character expression as an argument. The function then evaluates the expression to determine the variable name and returns the value that is associated with that variable name.
- VVALUE does not accept an expression as an argument, but it does accept array references. VVALUEX accepts expressions, but the value of the expression cannot denote an array reference.
- VVALUEX and an assignment statement both return a character string that contains the current value of the variable that you specify. With VVALUEX, the value is formatted by using the current format that is associated with the variable. With an assignment statement, however, the value is unformatted.
- The PUT function allows you to reformat a specified variable or constant. VVALUEX uses the current format that is associated with the variable.

Example

The following SAS statements produce this result.

SAS Statement	Result

`date1='31mar02'd;`
`date2='date1';`
`format date1 date7.;`
`datevalue=vvaluex(date2);`
`put datevalue;`
	`31MAR02`

See Also

Functions:

- "VVALUE Function" on page 973
- Variable Information functions in "SAS Functions and CALL Routines by Category" on page 65

WEEK Function

Returns the week-number value.

Category: Date and Time

Syntax

WEEK(<*sas-date*> , <*'descriptor'*>)

Optional Arguments

sas-date
specifies the SAS data value. If the *sas-date* argument is not specified, the WEEK function returns the week-number value of the current date.

descriptor
specifies the value of the descriptor. The following descriptors can be specified in uppercase or lowercase characters.

U (default)
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('31dec2006'd, 'u')`** is 53.

Tip The U and W descriptors are similar, except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

See "The U Descriptor" on page 977

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 4th and the first

Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year.

See "The V Descriptor" on page 977

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('31dec2006'd, 'w')`** is 52.

Tip The U and W descriptors are similar except that the U descriptor considers Sunday as the first day of the week, and the W descriptor considers Monday as the first day of the week.

See "The W Descriptor" on page 977

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('01jan2007'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 4th and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. In the following example, 01jan2006 and 30dec2005 occur in the same week. The first day (Monday) of that week is 26dec2005. Therefore, **`week('01jan2006'd, 'v')`** and **`week('30dec2005'd, 'v')`** both return a value of 52. This means that both dates occur in week 52 of the year 2005.

The W Descriptor

The WEEK function with the W descriptor reads a SAS date value and returns the number of the week within the year. The number-of-the-week value is represented as a decimal number in the range 0–53, with a leading zero and maximum value of 53. Week 0 means that the first day of the week occurs in the preceding year. The fifth week of the year would be represented as 05.

Monday is considered the first day of the week. Therefore, the value of **`week('01jan2007'd, 'w')`** is 1.

Comparisons of Descriptors

U is the default descriptor. Its range is 0-53, and the first day of the week is Sunday. The V descriptor has a range of 1-53 and the first day of the week is Monday. The W descriptor has a range of 0-53 and the first day of the week is Monday.

The following list describes the descriptors and an associated week:

- Week 0:

  U indicates the days in the current Gregorian year before week 1.

  V does not apply.

  W indicates the days in the current Gregorian year before week 1.

- Week 1:

  U begins on the first Sunday in a Gregorian year.

  V begins on the Monday between December 29 of the previous Gregorian year and January 4 of the current Gregorian year. The first ISO week can span the previous and current Gregorian years.

  W begins on the first Monday in a Gregorian year.

- End of Year Weeks:

  U specifies that the last week (52 or 53) in the year can contain less than 7 days. A Sunday to Saturday period that spans 2 consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

  V specifies that the last week (52 or 53) of the ISO year contains 7 days. However, the last week of the ISO year can span the current Gregorian and next Gregorian year.

  W specifies that the last week (52 or 53) in the year can contain less than 7 days. A Monday to Sunday period that spans two consecutive Gregorian years is designated as 52 and 0 or 53 and 0.

Example

The following example shows the values of the U, V, and W descriptors for dates near the end of certain years and the beginning of the next year. Examining the full data set illustrates how the behavior can differ between the various descriptors depending on the day of the week for January 1. The output displays the first 20 observations:

```
title 'Values of the U, V, and W Descriptors';
data a(drop=i date0 date1 y);
     date0 = '20dec2005'd;
     do y = 0 to 5;
        date1 = intnx("YEAR",date0,y,'s');
        do i = 0 to 20;
          date = intnx("DAY",date1,i);
          year = YEAR(date);
          week   = week(date);
          week_u = week(date, 'u');
          week_v = week(date, 'v');
          week_w = week(date, 'w');
          output;
        end;
```

```
     end;
     format date WEEKDATX17.;
run;
proc print;
run;
```

Display 2.70 *Results of Identifying the Values of the U, V, and W Descriptors*

Values of the U, V, and W Descriptors

Obs	date	year	week	week_u	week_v	week_w
1	Tue, 20 Dec 2005	2005	51	51	51	51
2	Wed, 21 Dec 2005	2005	51	51	51	51
3	Thu, 22 Dec 2005	2005	51	51	51	51
4	Fri, 23 Dec 2005	2005	51	51	51	51
5	Sat, 24 Dec 2005	2005	51	51	51	51
6	Sun, 25 Dec 2005	2005	52	52	51	51
7	Mon, 26 Dec 2005	2005	52	52	52	52
8	Tue, 27 Dec 2005	2005	52	52	52	52
9	Wed, 28 Dec 2005	2005	52	52	52	52
10	Thu, 29 Dec 2005	2005	52	52	52	52
11	Fri, 30 Dec 2005	2005	52	52	52	52
12	Sat, 31 Dec 2005	2005	52	52	52	52
13	Sun, 1 Jan 2006	2006	1	1	52	0
14	Mon, 2 Jan 2006	2006	1	1	1	1
15	Tue, 3 Jan 2006	2006	1	1	1	1
16	Wed, 4 Jan 2006	2006	1	1	1	1
17	Thu, 5 Jan 2006	2006	1	1	1	1
18	Fri, 6 Jan 2006	2006	1	1	1	1
19	Sat, 7 Jan 2006	2006	1	1	1	1
20	Sun, 8 Jan 2006	2006	2	2	1	1
21	Mon, 9 Jan 2006	2006	2	2	2	2
22	Wed, 20 Dec 2006	2006	51	51	51	51
23	Thu, 21 Dec 2006	2006	51	51	51	51
24	Fri, 22 Dec 2006	2006	51	51	51	51
25	Sat, 23 Dec 2006	2006	51	51	51	51
26	Sun, 24 Dec 2006	2006	52	52	51	51
27	Mon, 25 Dec 2006	2006	52	52	52	52

See Also

Functions:

- "INTNX Function" on page 580

Formats:

- "WEEKUw. Format" in *SAS Formats and Informats: Reference*
- "WEEKVw. Format" in *SAS Formats and Informats: Reference*
- "WEEKWw. Format" in *SAS Formats and Informats: Reference*

Informats:

- "WEEKUw. Informat" in *SAS Formats and Informats: Reference*
- "WEEKVw. Informat" in *SAS Formats and Informats: Reference*
- "WEEKWw. Informat" in *SAS Formats and Informats: Reference*

WEEKDAY Function

From a SAS date value, returns an integer that corresponds to the day of the week.

Category: Date and Time

Syntax

WEEKDAY(*date*)

Required Argument

date
specifies a SAS expression that represents a SAS date value.

Details

The WEEKDAY function produces an integer that represents the day of the week, where 1=Sunday, 2=Monday, ..., 7=Saturday.

Example

The following SAS statements produce this result.

SAS Statement	Result
`x=weekday('16mar97'd);` `put x;`	`1`

WHICHC Function

Searches for a character value that is equal to the first argument, and returns the index of the first matching value.

Category: Search

Syntax

WHICHC(*string*, *value-1* <, *value-2*, …>)

Required Arguments

string
is a character constant, variable, or expression that specifies the value to search for.

value
is a character constant, variable, or expression that specifies the value to be searched.

Details

The WHICHC function searches the second and subsequent arguments for a value that is equal to the first argument, and returns the index of the first matching value.

If *string* is missing, then WHICHC returns a missing value. Otherwise, WHICHC compares the value of *string* with *value-1*, *value-2*, and so on, in sequence. If argument *value-i* equals *string*, then WHICHC returns the positive integer *i*. If *string* does not equal any subsequent argument, then WHICHC returns 0.

Using WHICHC is useful when the values that are being searched are subject to frequent change. If you need to perform many searches without changing the values that are being searched, using the HASH object is much more efficient.

Example

The following example searches the array for the first argument and returns the index of the first matching value.

```
data _null_;
   array fruit (*) $12 fruit1-fruit3 ('watermelon' 'apple' 'banana');
   x1=whichc('watermelon', of fruit[*]);
   x2=whichc('banana', of fruit[*]);
   x3=whichc('orange', of fruit[*]);
   put x1= / x2= / x3=;
run;
```

SAS writes the following output to the log:

```
x1=1
x2=3
x3=0
```

See Also

Functions:

- "WHICHN Function" on page 983

Other References:

- "Using the Hash Object " in Chapter 22 of *SAS Language Reference: Concepts*
- "The IN Operator in Character Comparisons" in Chapter 6 of *SAS Language Reference: Concepts*

WHICHN Function

Searches for a numeric value that is equal to the first argument, and returns the index of the first matching value.

Category: Search

Syntax

WHICHN(*argument*, *value-1* <, *value-2*, …>)

Required Arguments

argument
is a numeric constant, variable, or expression that specifies the value to search for.

value
is a numeric constant, variable, or expression that specifies the value to be searched.

Details

The WHICHN function searches the second and subsequent arguments for a value that is equal to the first argument, and returns the index of the first matching value.

If *string* is missing, then WHICHN returns a missing value. Otherwise, WHICHN compares the value of *string* with *value-1*, *value-2*, and so on, in sequence. If argument *value-i* equals *string*, then WHICHN returns the positive integer *i*. If *string* does not equal any subsequent argument, then WHICHN returns 0.

Using WHICHN is useful when the values that are being searched are subject to frequent change. If you need to perform many searches without changing the values that are being searched, using the HASH object is much more efficient.

Example

The following example searches the array for the first argument and returns the index of the first matching value.

```
data _null_;
   array dates[*] Columbus Hastings Nicea US_Independence missing
                  Magna_Carta Gutenberg
                  (1492 1066 325 1776 . 1215 1450);
   x0=whichn(., of dates[*]);
   x1=whichn(1492, of dates[*]);
   x2=whichn(1066, of dates[*]);
   x3=whichn(1450, of dates[*]);
   x4=whichn(1000, of dates[*]);
```

```
   put x0= / x1= / x2= / x3= / x4=;
run;
```

SAS writes the following output to the log:

```
x0=.
x1=1
x2=2
x3=7
x4=0
```

See Also

Functions:

- "WHICHC Function" on page 982

Other References:

- "The IN Operator in Numeric Comparisons" in Chapter 6 of *SAS Language Reference: Concepts*
- "Using the Hash Object " in Chapter 22 of *SAS Language Reference: Concepts*

YEAR Function

Returns the year from a SAS date value.

Category: Date and Time

Syntax

YEAR(*date*)

Required Argument

date
specifies a SAS expression that represents a SAS date value.

Details

The YEAR function produces a four-digit numeric value that represents the year.

Example

The following SAS statements produce this result.

SAS Statement	Result

`date='25dec97'd;`
`y=year(date);`
`put y;`
	`1997`

See Also

Functions:

- "DAY Function" on page 360
- "MONTH Function" on page 669

YIELDP Function

Returns the yield-to-maturity for a periodic cash flow stream, such as a bond.

Category: Financial

Syntax

YIELDP(*A*,*c*,*n*,*K*,k_0,*p*)

Required Arguments

A

specifies the face value.

Range $A > 0$

c

specifies the nominal annual coupon rate, expressed as a fraction.

Range $0 \le c < 1$

n

specifies the number of coupons per year.

Range $n > 0$ and is an integer

K

specifies the number of remaining coupons from settlement date to maturity.

Range $K > 0$ and is an integer

k_0

specifies the time from settlement date to the next coupon as a fraction of the annual basis.

Range $0 < k_0 \le \frac{1}{n}$

p

specifies the price with accrued interest.

Range $p > 0$

Details

The YIELDP function is based on the following relationship:

$$P = \sum_{k=1}^{K} c(k) \frac{1}{\left(1 + \frac{y}{n}\right)^{t_k}}$$

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.

```
data _null_;
   y=yieldp(1000,.01,4,14,.165,800);
   put y;
run;
```

The value returned is 0.0775031248.

YRDIF Function

Returns the difference in years between two dates according to specified day count conventions; returns a person's age.

Category: Date and Time

Syntax

YRDIF(*start-date*,*end-date*,<*basis*>)

Required Arguments

start-date
specifies a SAS date value that identifies the starting date.

end-date
specifies a SAS date value that identifies the ending date.

Optional Argument

basis
identifies a character constant or variable that describes how SAS calculates a date difference or a person's age. The following character strings are valid:

'30/360'
specifies a 30-day month and a 360-day year in calculating the number of years. Each month is considered to have 30 days, and each year 360 days, regardless of the actual number of days in each month or year.

Alias '360'

Tip If either date falls at the end of a month, it is treated as if it were the last day of a 30-day month.

'ACT/ACT'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days that fall in 365-day years divided by 365 plus the number of days that fall in 366-day years divided by 366.

Alias 'Actual'

'ACT/360'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 360, regardless of the actual number of days in each year.

'ACT/365'
uses the actual number of days between dates in calculating the number of years. SAS calculates this value as the number of days divided by 365, regardless of the actual number of days in each year.

'AGE'
specifies that a person's age will be computed.

If you do not specify a third argument, AGE becomes the default value for *basis*.

Details

Using YRDIF in Financial Applications

The Basics

The YRDIF function can be used in calculating interest for fixed income securities when the third argument, *basis*, is present. YRDIF returns the difference between two dates according to specified day count conventions.

Calculations That Use ACT/ACT Basis

In YRDIF calculations that use the ACT/ACT basis, both a 365–day year and 366–day year are taken into account. For example, if n365 equals the number of days between the start and end dates in a 365–day year, and n366 equals the number of days between the start and end dates in a 366–day year, the YRDIF calculation is computed as YRDIF=n365/365.0 + n366/366.0. This calculation corresponds to the commonly understood ACT/ACT day count basis that is documented in the financial literature. The values for *basis* also includes 30/360, ACT/360, and ACT/365. Each has well-defined meanings that must be adhered to in calculating interest payments for specific financial instruments.

Computing a Person's Age

The YRDIF function can compute a person's age. The first two arguments, *start-date* and *end-date*, are required. If the value of *basis* is AGE, then YRDIF computes the age. The age computation takes into account leap years. No other values for *basis* are valid when computing a person's age.

Examples

Example 1: Calculating a Difference in Years Based on Basis

In the following example, YRDIF returns the difference in years between two dates based on each of the options for *basis*.

```
data _null_;
   sdate='16oct1998'd;
   edate='16feb2010'd;
   y30360=yrdif(sdate, edate, '30/360');
   yactact=yrdif(sdate, edate, 'ACT/ACT');
   yact360=yrdif(sdate, edate, 'ACT/360');
   yact365=yrdif(sdate, edate, 'ACT/365');
   put y30360= / yactact= / yact360= / yact365= ;
run;
```

SAS writes the following output to the log:

```
y30360=11.333333333
yactact=11.336986301
yact360=11.502777778
yact365=11.345205479
```

Example 2: Calculating a Person's Age

You can calculate a person's age by using three arguments in the YRDIF function. The third argument, *basis*, must have a value of AGE:

```
data _null_;
   sdate='16oct1998'd;
   edate='16feb2010'd;
   age=yrdif(sdate, edate, 'AGE');
   put age= 'years';
run;
```

SAS writes the following output to the log:

```
age=11.336986301 years
```

See Also

Functions:

- "DATDIF Function" on page 355

References

"Day count convention." *Wikipedia*, 2010. Available Day count convention.

ISDA International Swaps and Derivatives Association, Inc "EMU and Market Conventions: Recent Developments." 1998. Wikipedia. Available EMU and Market Conventions: Recent Developments.

Mayle, Jan. 1994. *Standard Securities Calculation Methods – Fixed Income Securities Formulas for Analytic Measures*. Vol. 2. NY, NY: Securities Industry Association.

YYQ Function

Returns a SAS date value from year and quarter year values.

Category: Date and Time

Syntax

YYQ(*year*,*quarter*)

Required Arguments

year
specifies a two-digit or four-digit integer that represents the year. The YEARCUTOFF= system option defines the year value for two-digit dates.

quarter
specifies the quarter of the year (1, 2, 3, or 4).

Details

The YYQ function returns a SAS date value that corresponds to the first day of the specified quarter. If either *year* or *quarter* is missing, or if the quarter value is not valid, the result is missing.

Example

The following SAS statements produce these results.

SAS Statement	Result

`DateValue=yyq(2001,3);`
`put DateValue;`
`put DateValue date7.;`
`put DateValue date9.;`
	
	`15157`
	`01JUL01`
	`01JUL2001`

`StartOfQtr=yyq(99,4);`
`put StartOfQtr;`
`put StartOfQtr=worddate.;`
	`14518`
	`StartOfQtr=October 1, 1999`

See Also

Functions:

- “QTR Function” on page 798
- “YEAR Function” on page 984

System Options:

- “YEARCUTOFF= System Option” in *SAS System Options: Reference*

ZIPCITY Function

Returns a city name and the two-character postal code that corresponds to a ZIP code.

Category: State and ZIP Code

Syntax

ZIPCITY(*ZIP-code*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip If the value of *ZIP-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPCITY assumes that the value is 01040.

Details

The Basics

If the ZIPCITY function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPCITY returns a city name and the two-character postal code that corresponds to its five-digit ZIP code argument. ZIPCITY returns the character values in mixed-case. If the ZIP code is unknown, ZIPCITY returns a blank value.

Note: The SASHELP.ZIPCODE data set must be present when you use this function. If you remove the data set, ZIPCITY will return unexpected results.

How the ZIP Code Is Translated to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. It then finds the corresponding state for that range of ZIP codes. The zone table consists of start and end ZIP code values for each state to allow for exceptions, and does not validate ZIP code values.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Determining When the State Postal Code Table Was Last Updated

The SASHELP.ZIPCODE data set contains postal code information that is used with ZIPCITY and other ZIP code functions. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=SASHELP.ZIPCODE;
run;
```

Output from the CONTENTS procedure provides the date of the last update, along with the contents of the SASHELP.ZIPCODE data set.

Note: You can download the latest version of the SASHELP.ZIPCODE file from the **Technical Support Web site**. Select **Zipcode Dataset** from the Name column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the name of the city in mixed-case and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Example: Examples

The following SAS statements produce these results.

SAS Statement	Result
`city1=zipcity(27511);` `put city1;`	`Cary, NC`

`length zip $10.;`
`zip='90049-1392';`
`zip=substr(zip,1,5);`
`city2=zipcity(zip);`
`put city2;`
	`Los Angeles, CA`

`city3=zipcity(4338);` `put city3;`	`Augusta, ME`
`city4=zipcity(01040);` `put city4;`	`Holyoke, MA`

See Also

Functions:

- "ZIPFIPS Function" on page 993
- "ZIPNAME Function" on page 994
- "ZIPNAMEL Function" on page 996
- "ZIPSTATE Function" on page 998

ZIPCITYDISTANCE Function

Returns the geodetic distance between two ZIP code locations.

Categories: Distance
State and ZIP Code

Syntax

ZIPCITYDISTANCE(*zip-code-1*, *zip-code-2*)

Required Argument

zip-code
specifies a numeric or character expression that contains the ZIP code of a location in the United States of America.

Details

The ZIPCITYDISTANCE function returns the geodetic distance in miles between two ZIP code locations. The centroid of each ZIP code is used in the calculation.

The SASHELP.ZIPCODE data set must be present when you use this function. If you remove the data set, then ZIPCITYDISTANCE will return unexpected results.

The SASHELP.ZIPCODE data set contains postal code information that is used with ZIPCITYDISTANCE and other ZIP code functions. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=SASHELP.ZIPCODE;
run;
```

Output from the CONTENTS procedure provides the date of the last update, along with the contents of the SASHELP.ZIPCODE data set.

Note: You can download the latest version of the SASHELP.ZIPCODE file from the SAS external Web site. The file is located at the **Technical Support Web site**. Select **Zipcode Dataset** from the **Name** column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set.

Example

In the following example, the first ZIP code identifies a location in San Francisco, CA, and the second ZIP code identifies a location in Bangor, ME. ZIPCITYDISTANCE returns the distance in miles between these two locations.

```
data _null_;
   distance=zipcitydistance('94103', '04401');
   put 'Distance from San Francisco, CA, to Bangor, ME: ' distance 4. ' miles';
run;
```

SAS writes the following output to the log:

```
Distance from San Francisco, CA, to Bangor, ME: 2782 miles
```

See Also

Functions:

- "ZIPCITY Function" on page 990

ZIPFIPS Function

Converts ZIP codes to two-digit FIPS codes.

Category: State and ZIP Code

Syntax

ZIPFIPS(*zip-code*)

Required Argument

zip-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip If the value of *zip-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPFIPS assumes that the value is 01040.

Details

The Basics

The ZIPFIPS function returns the two-digit numeric U.S. Federal Information Processing Standards (FIPS) code that corresponds to its five-digit ZIP code argument.

How the Zip Code Is Translated to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. It then finds the corresponding state for that range of ZIP codes. The zone table consists of start and end ZIP code values for each state to allow for exceptions, and does not validate ZIP code values.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Example

The following SAS statements produce these results.

SAS Statement	Result
`fips1=zipfips('27511');` `put fips1;`	`37`

SAS Statement	Result
`fips2=zipfips('01040');` `put fips2;`	25
`fips3=zipfips(1040);` `put fips3;`	25
`fips4=zipfips(59017);` `put fips4;`	30
`fips5=zipfips(24862);` `put fips5;`	54

See Also

Functions:

- “ZIPCITY Function” on page 990
- “ZIPNAME Function” on page 994
- “ZIPNAMEL Function” on page 996
- “ZIPSTATE Function” on page 998

ZIPNAME Function

Converts ZIP codes to uppercase state names.

Category: State and ZIP Code

Syntax

ZIPNAME(*zip-code*)

Required Argument

zip-code
specifies a numeric or character expression that contains a five-digit ZIP code.

Tip If the value of *zip-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPNAME assumes that the value is 01040.

Details

The Basics

If the ZIPNAME function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPNAME returns the name of the state or U.S. territory that corresponds to its five-digit ZIP code argument. ZIPNAME returns character values up to 20 characters long, all in uppercase.

How the Zip Code Is Translated to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. It then finds the corresponding state for that range of ZIP codes. The zone table consists of start and end ZIP code values for each state to allow for exceptions, and does not validate ZIP code values.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the mixed-case name of the city and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the upper-case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed-case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the uppercase two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Example

The following SAS statements produce these results.

SAS Statement	Result
`state1=zipname('27511');` `put state1;`	`NORTH CAROLINA`
`state2=zipname('01040');` `put state2;`	`MASSACHUSETTS`
`state3=zipname(1040);` `put state3;`	`MASSACHUSETTS`
`state4=zipname('59017');` `put state4;`	`MONTANA`

`length zip $10.;`
`zip='90049-1392';`
`zip=substr(zip,1,5);`
`state5=zipname(zip);`
`put state5;`
	`CALIFORNIA`

See Also

Functions:

- "ZIPCITY Function" on page 990
- "ZIPFIPS Function" on page 993
- "ZIPNAMEL Function" on page 996
- "ZIPSTATE Function" on page 998

ZIPNAMEL Function

Converts ZIP codes to mixed case state names.

Category: State and ZIP Code

Syntax

ZIPNAMEL(*zip-code*)

Required Argument

zip-code
specifies a numeric or character expression that contains a five-digit zip code.

Tip If the value of *zip-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPNAMEL assumes that the value is 01040.

Details

The Basics

If the ZIPNAMEL function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPNAMEL returns the name of the state or U.S. territory that corresponds to its five-digit zip code argument. ZIPNAMEL returns mixed-case character values up to 20 characters long.

How the Zip Code Is Translated to the State Postal Code

To determine which state corresponds to a particular zip code, this function uses a zone table that consists of the start and end zip code values for each state. It then finds the corresponding state for that range of zip codes. The zone table consists of start and end zip code values for each state to allow for exceptions, and does not validate zip code values.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the name of the city in mixed-case and the two-character postal code that corresponds to its five-digit zip code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit zip code argument.
- ZIPNAMEL returns the mixed-case name of the state or U.S. territory that corresponds to its five-digit zip code argument.
- ZIPSTATE returns the upper-case two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit zip code argument.

Example

The following SAS statements produce these results.

SAS Statement	Result
`state1=zipnamel('27511');` `put state1;`	`North Carolina`
`state2=zipnamel('01040');` `put state2;`	`Massachusetts`
`state3=zipnamel(1040);` `put state3;`	`Massachusetts`
`state4=zipnamel(59017);` `put state4;`	`Montana`

`length zip $10.;`
`zip='90049-1392';`
`zip=substr(zip,1,5);`
`state5=zipnamel(zip);`
`put state5;`
	`California`

See Also

Functions:

- “ZIPCITY Function” on page 990
- “ZIPFIPS Function” on page 993
- “ZIPNAME Function” on page 994
- “ZIPSTATE Function” on page 998

ZIPSTATE Function

Converts ZIP codes to two-character state postal codes.

Category: State and ZIP Code

Syntax

ZIPSTATE(*ZIP-code*)

Required Argument

ZIP-code
specifies a numeric or character expression that contains a valid five-digit ZIP code.

Tip If the value of *ZIP-code* begins with leading zeros, you can enter the value without the leading zeros. For example, if you enter 1040, ZIPSTATE assumes that the value is 01040.

Details

The Basics

If the ZIPSTATE function returns a value to a variable that has not yet been assigned a length, by default the variable is assigned a length of 20.

ZIPSTATE returns the two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument. ZIPSTATE returns character values in uppercase.

Note: ZIPSTATE does not validate the ZIP code.

How the ZIP Code Is Translated to the State Postal Code

To determine which state corresponds to a particular ZIP code, this function uses a zone table that consists of the start and end ZIP code values for each state. It then finds the corresponding state for that range of ZIP codes. The zone table consists of start and end ZIP code values for each state to allow for exceptions, and does not validate ZIP code values.

With very few exceptions, a zone does not span multiple states. The exceptions are included in the zone table. It is possible for new zones or new exceptions to be added by the U.S. Postal Service at any time. However, SAS software is updated only with each new release of the product.

Army Post Office (APO) and Fleet Post Office (FPO) Postal Codes

The ZIPSTATE function recognizes APO and FPO ZIP codes. These military ZIP codes correspond to their exit bases in the United States. The ZIP codes are contained in the SASHELP.ZIPMIL data set. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=SASHELP.ZIPMIL;
run;
```

Output from the CONTENTS procedure provides the date of the last update, along with the contents of the SASHELP.ZIPMIL data set.

Note: You can download the latest version of the SASHELP.ZIPMIL data set from the **Technical Support Web site**. Select **Zipcode Dataset** from the Name column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set.

Determining When the State Postal Code Table Was Last Updated

Except for APO and FPO addresses, the SASHELP.ZIPCODE data set contains postal code information that is used with ZIPSTATE and other ZIP code functions. To determine when this data set was last updated, execute PROC CONTENTS:

```
proc contents data=SASHELP.ZIPCODE;
run;
```

Output from the CONTENTS procedure provides the date of the last update, along with the contents of the SASHELP.ZIPCODE data set.

Note: You can download the latest version of the SASHELP.ZIPCODE data set from the **Technical Support Web site**. Select **Zipcode Dataset** from the Name column to begin the download process. You must execute the CIMPORT procedure after you download and unzip the data set.

Comparisons

The ZIPCITY, ZIPNAME, ZIPNAMEL, and ZIPSTATE functions accept the same argument but return different values:

- ZIPCITY returns the mixed-case name of the city and the two-character postal code that corresponds to its five-digit ZIP code argument.
- ZIPNAME returns the uppercase name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPNAMEL returns the mixed-case name of the state or U.S. territory that corresponds to its five-digit ZIP code argument.
- ZIPSTATE returns the upper-case two-character state postal code (or world-wide GSA geographic code for U.S. territories) that corresponds to its five-digit ZIP code argument.

Example

The following SAS statements produce these results.

SAS Statement	Result
`state1=zipstate('27511');` `put state1;`	NC
`state2=zipstate('01040');` `put state2;`	MA
`state3=zipstate(1040);` `put state3;`	MA

SAS Statement	Result
`state4=zipstate(59017);` `put state4;`	`MT`

`length zip $10.;`
`zip='90049-1392';`
`zip=substr(zip,1,5);`
`state5=zipstate(zip);`
`put state5;`
	`CA`

See Also

Functions:

- “ZIPCITY Function” on page 990
- “ZIPFIPS Function” on page 993
- “ZIPNAME Function” on page 994
- “ZIPNAMEL Function” on page 996

Chapter 3

References

References

Abramowitz, Milton, and Irene Stegun. 1964. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables – National Bureau of Standards Applied Mathematics Series #55*. Washington, USA: U.S. Government Printing Office.

Amos, D. E., S. L. Daniel, and K. Weston. "CDC 6600 Subroutines IBESS and JBESS for Bessel Functions I(v,x) and J(v,x), $x \geq 0$, $v \geq 0$." 1977. *ACM Transactions on Mathematical Software* 3: 76–95.

Aho, A. V., J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Reading, USA: Addison-Wesley Publishing Co..

Cheng, R. C. H. "The Generation of Gamma Variables." 1977. *Applied Statistics* 26: 71–75.

Duncan, D. B. "Multiple Range and Multiple F Tests." 1955. *Biometrics* 11: 1–42.

Dunnett, C. W. "A Multiple Comparisons Procedure for Comparing Several Treatments with a Control." . *Journal of the American Statistical Association* 50: 1096–1121.

Fishman, G. S. "Sampling from the Poisson Distribution on a Computer." 1976. *Computing* 17: 145–156.

Fishman, G. S. 1978. *Principles of Discrete Event Simulation*. New York, USA: John Wiley & Sons, Inc.

Fishman, G. S., and L. R. Moore. "A Statistical Evaluation of Multiplicative Congruential Generators with Modulus $(2^{31} - 1)$." 1982. *Journal of the American Statistical Association* 77: 1, 29–136.

Knuth, D. E. 1973. *The Art of Computer Programming, Volume 3. Sorting and Searching*. Reading, USA: Addison-Wesley.

Hochberg, Y., and A. C. Tamhane. 1987. *Multiple Comparison Procedures*. New York, USA: John Wiley & Sons, Inc.

Williams, D. A. "A Test for Differences Between Treatment Means when Several Dose Levels are Compared with a Zero Dose Control." 1971. *Biometrics* 27: 103–117.

Williams, D. A. "The Comparison of Several Dose Levels with a Zero Dose Control." 1972. *Biometrics* 28: 519–531.

Vincenty, T. “Direct and Inverse Solutions of Geodesics on the Ellipsoid with Application of Nested Equations.” 1975. 22: 88–93.

Appendix 1

Tables of Perl Regular Expression (PRX) Metacharacters

General Constructs

Table A1.1 *General Constructs*

Metacharacter	Description
( )	indicates grouping.
non-metacharacter	matches a character.
{ } [ ] ( ) ^ $. | * + ? \	to match these characters, override (escape) with \.
\	overrides the next metacharacter.
n	matches capture buffer *n*.
(?:...)	specifies a non-capturing group.

Basic Perl Metacharacters

The following table lists the metacharacters that you can use to match patterns in Perl regular expressions.

Table A1.2 *Basic Perl Metacharacters and Their Descriptions*

Metacharacter	Description
\a	matches an alarm (bell) character.
\A	matches a character only at the beginning of a string.
\b	matches a word boundary (the position between a word and a space): • "er\b" matches the "er" in "never" • "er\b" does not match the "er" in "verb"

Metacharacter	Description
\B	matches a non-word boundary: • "er\B" matches the "er" in "verb" • "er\B" does not match the "er" in "never"
\cA-\cZ	matches a control character. For example, \cX matches the control character control-X.
\C	matches a single byte.
\d	matches a digit character that is equivalent to [0−9].
\D	matches a non-digit character that is equivalent to [^0−9].
\e	matches an escape character.
\E	specifies the end of case modification.
\f	matches a form feed character.
\l	specifies that the next character is lowercase.
\L	specifies that the next string of characters, up to the \E metacharacter, is lowercase.
\n	matches a newline character.
\num $num	matches capture buffer *num*, where *num* is a positive integer. Perl variable syntax ($num) is valid when referring to capture buffers, but not in other cases.
\Q	escapes (places a backslash before) all non-word characters.
\r	matches a return character.
\s	matches any white space character, including space, tab, form feed, and so on, and is equivalent to [\f\n\r\t\v].
\S	matches any character that is not a white space character and is equivalent to [^\f\n\r\t\v].
\t	matches a tab character.
\u	specifies that the next character is uppercase.
\U	specifies that the next string of characters, up to the \E metacharacter, is uppercase.
\w	matches any word character or alphanumeric character, including the underscore.

Metacharacter	Description
\W	matches any non-word character or non-alphanumeric character, and excludes the underscore.
ddd	matches the octal character *ddd*.
\x*dd*	matches the hexadecimal character *dd*.
\z	matches a character only at the end of a string.
\Z	matches a character only at the end of a string or before newline at the end of a string.

Metacharacters and Replacement Strings

You can use the following metacharacters in both a regular expression and in replacement text, when you use a substitution regular expression:

- \l
- \u
- \L
- \E
- \U
- \Q

These metacharacters are useful in replacement text for controlling the case of capture buffers that are used within replacement text. For an example of how these metacharacters can be used, see “Replacing Text” on page 45

For a description of these metacharacters, see Table A1.2 on page 1003 .

Other Quantifiers

The following table lists other qualifiers that you can use in Perl regular expressions. The descriptions of the metacharacters in the table include examples of how the metacharacters can be used.

Table A1.3 *Other Quantifiers*

Metacharacter	Description
\	marks the next character as either a special character, a literal, a back reference, or an octal escape: • “\n” matches a newline character • “\\” matches “\” • “\(“ matches”(“

Metacharacter	Description
|	specifies the *or* condition when you compare alphanumeric strings. For example, the construct x|y matches either x or y: • "z|food" matches either "z" or "food" • “(z|f)ood” matches “zood” or “food”
^	matches the position at the beginning of the input string.
$	matches the position at the end of the input string.
period (.)	matches any single character except newline. To match any character including newline, use a pattern such as "[.\n]".
(pattern)	specifies grouping. Matches a pattern and creates a capture buffer for the match. To retrieve the position and length of the match that is captured, use CALL PRXPOSN. To retrieve the value of the capture buffer, use the PRXPOSN function. To match parentheses characters, use "\(" or "\)".

Greedy and Lazy Repetition Factors

Perl regular expressions support “greedy” repetition factors and “lazy” repetition factors. A repetition factor is considered greedy when the repetition factor matches a string as many times as it can when using a specific starting location. A repetition factor is considered lazy when it matches a string the minimum number of times that is needed to satisfy the match. To designate a repetition factor as lazy, add a ? to the end of the repetition factor. By default, repetition factors are considered greedy.

The following table lists the greedy repetition factors. The descriptions of the repetition factors in the table include examples of how they can be used.

Table A1.4 *Greedy Repetition Factors*

Metacharacter	Description
*	matches the preceding subexpression zero or more times: • zo* matches "z" and "zoo" • * is equivalent to {0,}
+	matches the preceding subexpression one or more times: • "zo+" matches "zo" and "zoo" • "zo+" does not match "z" • + is equivalent to {1,}
?	matches the preceding subexpression zero or one time: • "do(es)?" matches the "do" in "do" or "does" • ? is equivalent to {0,1}

Metacharacter	Description
{n}	matches at least *n* times.
{n,}	matches a pattern at least *n* times.
{n,m}	m and n are non-negative integers, where n<=m. They match at least *n* and at most *m* times: • "o{1,3}" matches the first three o's in "fooooood" • "o{0,1}" is equivalent to "o?" You cannot put a space between the comma and the numbers.

The following table lists the lazy repetition metacharacters.

Table A1.5 *Lazy Repetition Factors*

Metacharacter	Description
*?	matches a pattern zero or more times.
+?	matches a pattern one or more times.
??	matches a pattern zero or one time.
{n}?	matches exactly *n* times.
{n,}?	matches a pattern at least *n* times.
{n,m}?	matches a pattern at least *n* times but not more than *m* times.

Class Groupings

The following table lists character class groupings. You specify these classes by enclosing characters inside brackets. These metacharacters share a set of common properties. To be successful, the character class must always match a character. The negated character class must always match a character that is not in the list of characters that are designated inside the brackets. The descriptions of the metacharacters in the table include examples of how the metacharacters can be used.

Table A1.6 *Character Class Groupings*

Metacharacter	Description
[...]	specifies a character set that matches any one of the enclosed characters: • “[abc]” matches the “a” in “plain”

Metacharacter	Description
[^...]	specifies a negative character set that matches any character that is not enclosed: • “[^abc]” matches the “p” in “plain”
[a-z]	specifies a range of characters that matches any character in the range: • “[a-z]” matches any lowercase alphabetic character in the range “a” through “z”
[^a-z]	specifies a range of characters that does not match any character in the range: • "[^a-z]" matches any character that is not in the range "a" through "z"
[[:alpha:]]	matches an alphabetic character.
[[:^alpha:]]	matches a nonalphabetic character.
[[:alnum:]]	matches an alphanumeric character.
[[:^alnum:]]	matches a non-alphanumeric character.
[[:ascii:]]	matches an ASCII character. Equivalent to [\0–\177].
[[:^ascii:]]	matches a non-ASCII character. Equivalent to [^\0–\177].
[[:blank:]]	matches a blank character.
[[:^blank:]]	matches a non-blank character.
[[:cntrl:]]	matches a control character.
[[:^cntrl:]]	matches a character that is not a control character.
[[:digit:]]	matches a digit. Equivalent to \d.
[[:^digit:]]	matches a non-digit character. Equivalent to \D.
[[:graph:]]	is a visible character, excluding the space character. Equivalent to [[:alnum:][:punct:]].
[[:^graph:]]	is not a visible character. Equivalent to [^[:alnum:][:punct:]].
[[:lower:]]	matches lowercase characters.
[[:^lower:]]	does not match lowercase characters.
[[:print:]]	prints a string of characters.
[[:^print:]]	does not print a string of characters.
[[:punct:]]	matches a punctuation character or a visible character that is not a space or alphanumeric.
[[:^punct:]]	does not match a punctuation character or a visible character that is not a space or alphanumeric.

Metacharacter	Description
[[:space:]]	matches a space. Equivalent to \s.
[[:^space:]]	does not match a space. Equivalent to \S.
[[:upper:]]	matches uppercase characters.
[[:^upper:]]	does not match uppercase characters.
[[:word:]]	matches a word. Equivalent to \w.
[[:^word:]]	does not match a word. Equivalent to \W.
[[:xdigit:]]	matches a hexadecimal character.
[[:^xdigit:]]	does not match a hexadecimal character.

Look-Ahead and Look-Behind Behavior

Look-ahead and look-behind are ways to look ahead or behind a match to see whether a particular text occurs. The text that is found with look-ahead or look-behind is not included in the match that is found. For example, if you want to find names that end with “Jr.”, but you do not want “Jr.” to be part of the match, you could use the regular expression /.*(?=Jr\.)/. For the value "John Wainright Jr.", the regular expression will find "John Wainright" as a match because it is followed by "Jr."

Table A1.7 *Look-Ahead and Look-Behind Behavior*

Metacharacter	Description
(?=...)	specifies a zero-width, positive, look-ahead assertion. For example, in the expression *regex1 (?=regex2)*, a match is found if both *regex1* and *regex2* match. *regex2* is not included in the final match.
(?!...)	specifies a zero-width, negative, look-ahead assertion. For example, in the expression *regex1 (?!regex2)*, a match is found if *regex1* matches and *regex2* does not match. *regex2* is not included in the final match.
(?<=...)	specifies a zero-width, positive, look-behind assertion. For example, in the expression *(?<=regex1) regex2*, a match is found if both *regex1* and *regex2* match. *regex1* is not included in the final match. Works with fixed-width look-behind only.
(?<!...)	specifies a zero-width, negative, look-behind assertion. Works with fixed-width look-behind only.

Comments and Inline Modifiers

The metacharacters in this table contain a question mark as the first element inside the parentheses. The characters after the question mark indicate the extension.

Table A1.8 *Comments and Inline Modifiers*

Metacharacter	Description
(?#text)	specifies a comment in which the text is ignored.
(?imsx)	specifies one or more embedded pattern-matching modifiers. If the pattern is case insensitive, you can use (?i) at the front of the pattern. An example is `$pattern="(?i)foobar";`. Letters that appear after a hyphen (-) turn the modifiers off.

Selecting the Best Condition by Using Combining Operators

The elementary regular expressions (for example, \a and \w) that are described in the preceding tables can match at most one substring at the given position in the input string. However, operators that perform combining in typical regular expressions combine elementary metacharacters to create more complex patterns. In an ambiguous situation, these operators can determine the best match or the worst match. The match that is the best is always chosen.

Table A1.9 *Best Match Using Combining Operators*

Metacharacter	Description
ST	in the following example, specifies that AB and A'B', and A and A' are substrings that can be matched by S, and that B and B' are substrings that can be matched by T: • If A is a better match for S than A', then AB is a better match than A'B'. • If A and A' coincide, then AB is a better match than AB' if B is a better match for T than B'.
S|T	specifies that when S can match, it is a better match than when only T can match. The ordering of two matches for S is the same as for S. Similarly, the ordering of two matches for T is the same as for T.
S{repeat-count}	matches as SSS . . . S (repeated as many times as necessary).
S{min,max}	matches as S{max}|S{max-1}| . . . |S{min+1}|S{min}.
S{min,max}?	matches as S{min}|S{min+1}| . . . |S{max-1}|S{max}.
S?, S*, S+	same as S{0,1}, S{0, big-number}, S{1,big-number}, respectively.
S??, S*?, S+	same as S{0,1}?, S{0, big-number}?, S{1,big-number}?, respectively.
(?=S), (?<=S)	considers the best match for S. (This is important only if S has capturing parentheses, and back references are used elsewhere in the whole regular expression.)

Metacharacter	Description
(?!S), (?<!S)	unnecessary to describe the ordering for this grouping operator because only whether S can match is important.

Index

B

C

D

E

I

N

O

P

Q

R

S

V

W

Y

Z