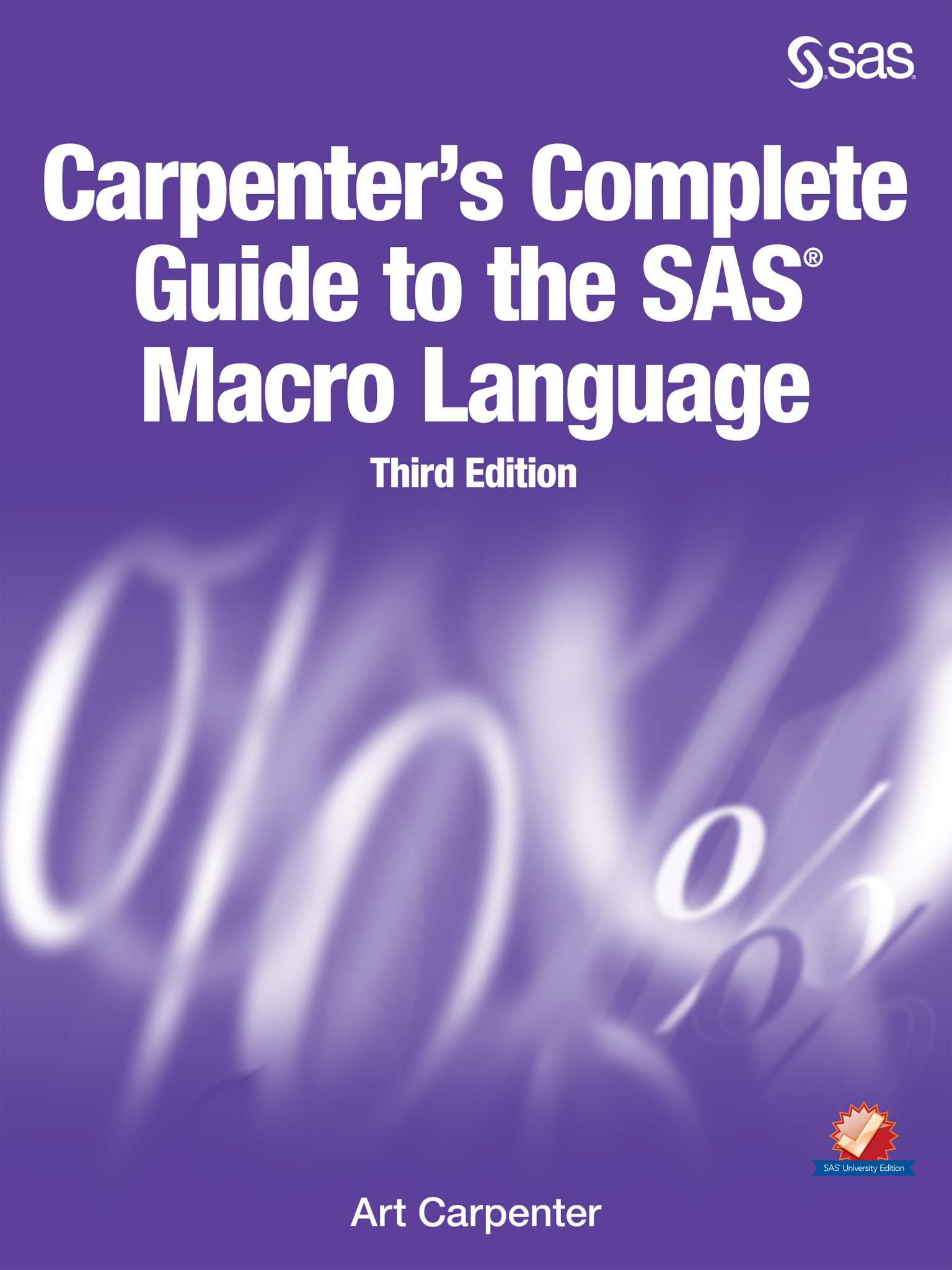
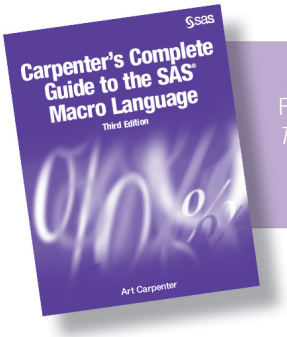
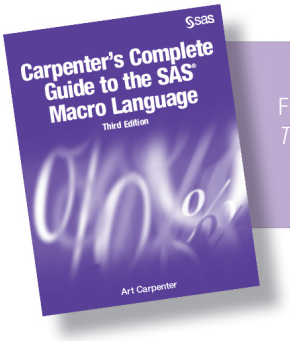

sas
Carpenter's Complete Guide to the SAS® Macro Language
Third Edition
SAS University Edition
Art Carpenter



Contents



Chapter 8: Discovering Even More Macro Language Elements

In this chapter a second look is taken at a number of types of macro language elements, such as functions and options that have been introduced throughout this book. Here you will find elements of the macro language that tend to be less commonly used, not necessarily because they are less important, but for the most part, the elements noted in this chapter have a narrower focus and therefore a more limited utility.

As you read through this chapter you will notice that the examples tend to highlight the usage of the element being described. The examples are not intended to be 'practical' in and of themselves, but are instead designed to demonstrate certain aspects of the elements being discussed.

For the examples in this chapter and indeed for all of the code examples throughout the book, if you want to execute these sample programs, then be sure to follow the setup instructions. Remember that all of the data sets and programs are available for download, so you do not need to retype either the code or the data.

For instructions on accessing and setting up the programs and data, see the "Example Code and Data" section within this edition's "About This Book" front matter.

SEE ALSO: A number of these newer features are discussed by Langston (2015a).

8.1 Even More Macro Functions

Although you will probably not reach for the functions in this section often, when you need them, you will tend to *really* need them. They enable you to interface with the operating system and to track the progress of your macro application.

8.1.1 Accessing System Environmental Variables Using %SYSGET

Just as SAS uses macro variables, operating systems use a similar system of symbolic variables known as *environmental variables*. SAS takes advantage of these environmental variables in a number of ways, and this is usually to store information that has some connection between SAS and the operating system. Sometimes we would like to access the information stored in these environmental variables, and we can do just that by using the %SYSGET macro function, which is similar to the SYSGET DATA step function.

Environment variables can be set either through the operating system or by SAS, and since these variables can provide a link between SAS and the operating system, they can be a valuable interface tool when writing macros.

SYNTAX:

`%SYSGET(`*`environmentalvariablename`*`)`

VALUE RETURNED:

Value held by the environmental variable

Probably the most difficult part about using this function is knowing what environmental variables exist, and how the information that those environmental variables hold will be helpful. A number of environment variables are available to the user; however, they vary by operating system, and can be additionally tailored when SAS is invoked. Your SAS Companion and SAS Online Doc go into some detail on setting environment variables, either through SAS or through the operating system.

Environmental Variables Created by the Configuration File

When the configuration file is executed at SAS initialization, a number of environmental variables are created. In the configuration file, under Windows, the keyword SET is used to name the environmental variables. Here is a portion of a SASv9.cfg file (SAS9.4 under Windows) that creates the SASAUTOS environmental variable:

```
-SET SASROOT "C:\Program Files\SASHome2\SASFoundation\9.4" ❶

-SET SASAUTOS (
        "!SASROOT\core\sasmacro" ❷
        "!SASROOT\aacomp\sasmacro"
        "!SASROOT\accelmva\sasmacro"
        "!SASROOT\assist\sasmacro"
. . . portions of this statement are not shown . . .
```

❶ The SASROOT environmental variable is defined.

❷ The SASROOT environmental variable is used in the definition of the SASAUTOS environmental variable. The SASAUTOS environmental variable can be used as a *fileref* in SAS programs. It can also be retrieved using the %SYSGET function. The value stored in SASAUTOS can be surfaced by using

the %SYSGET function, and a portion of the SAS Log showing the usage of %SYSGET with the SASAUTOS environmental variable is shown here:

```
275  %put %sysget(sasautos);
(                    "!SASROOT\core\sasmacro"
"!SASROOT\aacomp\sasmacro"
"!SASROOT\accelmva\sasmacro"          "!SASROOT\assist\sasmacro"
. . . portions of the LOG are not shown . . .
```

You can see some of the environment variables that SAS has created and their current values by viewing the value of the SET system option in SASHELP.VOPTIONS.

Program 8.1.1a: Viewing Selected Environmental Variables

```
proc print data=sashelp.voption(where=(optname='SET'));
run;
```

Finding the SAS Executable File Location

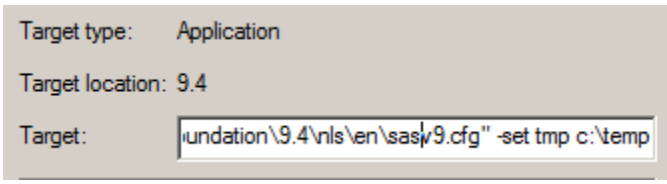
If you are writing code that will be used across operating systems or for different versions of SAS, you may need to know the location of the SAS executable file. For some operating systems, like Windows, this location information is stored in the !SASROOT environment variable, and the %SYSGET function can be used to determine this value directly. To create a macro variable that contains the full path to the executable file for the current OS and version of SAS, the %SYSGET is used to retrieve the current value of !SASROOT.

```
%let sasloc = %sysget(sasroot)\sas.exe;
```

Accessing Environmental *Librefs* and *Filerefs*

One common use of environmental variables is to associate locations (paths or directories) with a name. Usually, the LIBNAME or FILENAME statements are used to create this association from within SAS, but if the association is created outside of SAS, the programs can become more location independent and may require less maintenance when moved from machine to machine. You can ask SAS to interpret an environmental variable as a *libref* or *fileref*. The SASAUTOS environmental variable is used as a *fileref* in the SASAUTOS system option when setting up the autocall library (see Section 10.1.3). Under Windows you can set the environmental variable in the properties section of the SAS shortcut by using the –SET initialization option (see Section 14.6 for more on SAS initialization options).

Figure 8.1.1: Using the –SET Initialization Option to Create an Environmental Variable



Once it has been created, this environmental variable can be used as a *libref*, even though it will not show up on your list of libraries. TMP now refers to the directory C:\TEMP.

Program 8.1.1b: Using an Environmental Variable as a *libref*

```
proc print data=tmp.oldtest;
run;
```

The method used to set the environmental variables will vary among operating systems. Consult the SAS Companion for your OS.

Returning the Name and Path of the Currently Executing Program

When executing an existing SAS program from the Enhanced Editor within the Display Manager (Windows), SAS knows where in the operating system the program is stored and what its name is. You can retrieve that information by using the SAS_EXECFILEPATH and SAS__EXECFILENAME environmental variables. Program 8.1.1c shows how the name of the executing program is returned by using %SYSGET in a FOOTNOTE statement.

Program 8.1.1c: Returning the Executing Program Name

```
footnote1 justify=c
     "The executing program is: %sysget(sas_execfilename)";
proc print data=sashelp.class;
run;
```

The environmental variable SAS_EXECFILEPATH contains both the name of the file and the physical path to that file. The %GRABPATH macro shown in Program 8.1.1d uses these two environmental variables together to return the path without the program name.

Program 8.1.1d: Returning the Path of the Executing Program

```
%macro grabpath ;
   %* return the path of the currently executing program;
   %qsubstr(%sysget(SAS_EXECFILEPATH), ❸
            1, ❹
            %length(%sysget(SAS_EXECFILEPATH))- ❺
                  %length(%sysget(SAS_EXECFILEname))
            )
%mend grabpath;

footnote1 justify=c "The path to the executing program is: %grabpath";
proc print data=sashelp.class;
run;
```

❸ The %QSUBSTR function is used to grab the path portion.

❹ The grab starts in the first position and continues until the name of the program.

❺ The length of the whole path (including the program name) less the length of the name of the program yields the width of the path portion of the text contained in the SAS_EXECFILEPATH environmental variable.

You can use the operating system itself to surface the currently defined environmental variables. Under Windows environmental variables are defined using the SET command. When used without an argument the SET command lists all the currently defined environmental variables. The %SYSEXEC statement can be used to issue the SET command as shown in Program 8.1.1e.

Program 8.1.1e: Listing Current Environmental Variables and Their Values

```
options noxwait;
%sysexec set > c:\temp\environvar.txt; ❻
```

❻ The SET command is issued without an argument, and the results are written to the specified file.

A portion of the file (`C:\temp\environvar.txt` ❻) shows some of the current environmental variable values.

```
SASCFG=C:\Program Files\SASHome2\SASFoundation\9.4\nls\en
SASHOME=C:\Program Files\SASHome2
SASROOT=C:\Program Files\SASHome2\SASFoundation\9.4
SAS_EXECFILENAME=Carpenter_17835TW_Program8.1.1e.sas
```

If you are using SAS Enterprise Guide or SAS Studio, the macro variable &_SASPROGRAMFILE can be used. This macro variable returns the full path and filename of the SAS program that is currently being run. This macro variable is available only for SAS program files that are saved on the same server on which your SAS Studio code or SAS Enterprise Guide session is running.

MORE INFORMATION: The SYMEXIST DATA step function (see Section 8.5.3) can be used to determine if an environmental variable has been defined.

SEE ALSO: Levin (2001) and Lund (2001a, 2001b) use the %SYSGET macro function. Carpenter (2008) discusses the %GRABPATH macro in more detail, as well as other ways to access system environmental variables. Pahmer (2014) uses %SYMGET to retrieve the name of the executing program.

8.1.2 %SYSMEXECDEPTH and %SYSMEXECNAME

When you have developed a series of nested macros (macros that call other macros), it can sometimes become important to be able to determine which macros are being called and in which order. The nesting depth and the name of the executing macro at each depth can be surfaced using the %SYSMEXECDEPTH and %SYSMEXECNAME functions. These two functions are usually used together, however it is not necessary to do so.

SYNTAX:

`%SYSMEXECDEPTH`

VALUE RETURNED:

The number of nesting levels (0 for open code)

SYNTAX:

`%SYSMEXECNAME(`*`level_number`*`)`

VALUE RETURNED:

Name of the called macro at the specified nesting level

The macro %SHOWMACNEST in Program 8.1.2 uses the %SYSMEXECDEPTH and the %SYSMEXECNAME functions to highlight the nesting structure of nested macros.

Program 8.1.2: Using the %SYSMEXECDEPTH and %SYSMEXECNAME Functions

```
%macro ShowMacNest;
   %local i;
   %do i = 1 %to %sysmexecdepth; ❶
      %put Level &i, Macro name is: %sysmexecname(&i); ❷
   %end;
%mend showmacnest;
```

❶ The %SYSMEXECDEPTH function returns the total number of nesting levels. Here this value is used as the upper bound for a %DO loop.

❷ The %SYSMEXECNAME function returns the macro name for the specified nesting level (in this case the level is &I).

The use of %SHOWMACNEST is demonstrated in the nested macros shown here. In this example, the macro %ONE calls %TWO, which calls %THREE, which calls %SHOWMACNEST.

```
%macro one;
   %put in one;
   %two
%mend one;
%macro two;
   %put in two;
```

```
   %three
%mend two;
%macro three;
   %put in three;
   %showmacnest ❸
%mend three;

%put Level 0: %sysmexecname(0); ❹
%one
```

❸ %SHOWMACNEST is called from within the macro %THREE.

❹ Nesting level = 0 is used to indicate open code.

The SAS Log shows the various nesting levels:

```
Level 0:OPEN CODE ❹
1280  %one
in one
in two
in three
Level 1, Macro name is:ONE ❷
Level 2, Macro name is:TWO ❷
Level 3, Macro name is:THREE ❷
Level 4❶, Macro name is:SHOWMACNEST ❷
```

MORE INFORMATION: The automatic macro variable &SYSMACRONAME, which surfaces the name of the currently executing macro is described in Section 8.3.5.

SEE ALSO: Langston (2013) describes a macro that checks for the existence of a specified macro.

8.1.3 Assessing Macro Existence and Execution Status with %SYSMACEXEC and %SYSMACEXIST

When you are executing an application that has a series of macros that call other macros, it is not always easy to determine which macro is currently executing or sometime even if a macro definition currently exists. Fortunately, we are not without tools to help us. In Section 8.1.2 the %SYSMEXECDEPTH and %SYSMEXECNAME functions are used to show nesting structure.

The %SYSMACEXEC and %SYSMACEXIST functions can be used to determine if a macro is currently executing or if it has been compiled.

SYNTAX:

`%SYSMACEXIST(`*macro_name*`)`

VALUES RETURNED:

1 if the macro has been compiled and resides in the WORK.SASMACR catalog

0 if the macro definition is not in WORK.SASMACR

SYNTAX:

`%SYSMACEXEC(`*macro_name*`)`

VALUES RETURNED:

Determines if the named macro is currently executing

The macro %MACEXEC in Program 8.1.3 checks to see if the specified macro has been compiled and whether it is currently executing.

Program 8.1.3: Determine If a Macro Has Been Compiled and If It Is Executing

```
options sasmstore=macro3 mstored; ❶

%macro one/store; ❷
   %put in one;
   %two
%mend one;
%macro two;
   %put in two;
   %three
%mend two;
%macro three;
   %put in three;
   %macexec(one) ❸
   %macexec(three) ❹
   %macexec(silly) ❺
%mend three;
%macro Macexec(macname);
   %if %sysmacexist(&macname) %then
   %put %upcase(&macname) exists in WORK.SASMACR; ❻
   %else %put %upcase(&macname) does not exist in WORK.SASMACR;
   %if %sysmacexec(&macname) %then
%put %upcase(&macname) is currently executing; ❼
%mend macexec;
%one ❽
```

❶ Turn on the ability to use stored compiled macros so the interaction with this type of library can be demonstrated.

❷ Store the compiled version of %ONE in the stored compiled macro library.

❸ %ONE is executing but the compiled macro is not in the WORK catalog.

❹ %THREE is executing and the compiled macro is in the WORK catalog.

❺ %SILLY does not exist and has not been compiled.

❻ Check to see if the macro has been compiled.

❼ Check to see if the macro is currently executing.

❽ The macro %ONE is called, which in turn will call the other macros.

```
1376  %one ❽
in one
in two
in three
ONE does not exist in WORK.SASMACR
ONE is currently executing ❶ ❼
THREE exists in WORK.SASMACR ❹ ❻
THREE is currently executing ❹ ❼
```

Only the macro %THREE is detected in the WORK.SASMACR catalog by the %SYSMACEXIST function, while both the %ONE and %THREE macros are detected as executing by the %SYSMACEXEC function.

8.1.4 Determining Product Availability Using %SYSPROD

The %SYSPROD macro function can be used to determine if a particular SAS product has been licensed at your site. The function argument is the name of the product that you want to check for. It will also let you know if you have used it to query for a product that the function does not recognize.

SYNTAX:

`%SYSPROD(`*`product_name`*`)`

VALUE RETURNED:

1	if the product is available
0	if the product is not available
−1	if the product name is not recognized

In Program 8.1.4 the macro %CHECKPROD uses the %SYSPROD macro function to check the availability of a specified SAS product.

Program 8.1.4: Using the %SYSPROD Function

```
%macro Checkprod(prod=);
   %if %sysprod(&prod)=1 %then %put &prod is available;
   %else %if %sysprod(&prod)=0 %then %put &prod is not available;
   %else %if %sysprod(&prod)=-1 %then %put &prod is unknown;
%mend checkprod;
```

The SAS Log shows that SASGRAPH is not an acceptable code for a SAS product, while GRAPH is:

```
1448  %checkprod(prod=sasgraph)
sasgraph is unknown
1449  %checkprod(prod=graph)
graph is available
1450  %checkprod(prod=gis)
gis is not available
```

One of the disadvantages of this function is that it expects that the SAS products use specific codes, and it is not obvious what those codes are. Worse, the documentation only lists a few of the codes for some of the more common products. Some of the commonly used codes for the %SYSPROD function are as follows:

- AF
- ASSIST
- BASE
- CALC
- CONNECT
- CPE
- EIS
- ETS
- FSP
- GIS
- GRAPH
- IML
- INSIGHT
- LAB
- OR
- PH-CLINICAL
- QC
- SHARE
- STAT
- TOOLKIT

8.1.5 Checking Up on Macro Variable Scopes

There are three macro functions that can be used to determine if a macro variable exists and if so, what symbol table it resides in.

SYNTAX:

```
%SYMEXIST(macro_variable_name)
%SYMGLOBL(macro_variable_name)
%SYMLOCAL(macro_variable_name)
```

The %SYMEXIST function is used to determine whether a macro variable exists. The %SYMGLOBL and %SYMLOCAL functions are used to determine whether a macro variable resides in either the global or a local table, respectively. Each of these functions returns a true/false (1 or 0). If either %SYMGLOBL or %SYMLOCAL is true %SYMEXIST will necessarily be true as well. The macro %SYMCHKUP in Program 8.1.5 returns a 0 if the macro variable does not exist, 1 if it is global, 2 if it is local, and 3 if there is both a global and local instance of the macro variable.

Program 8.1.5: Checking the Scope of a Macro Variable

```
%macro symchkup(mvar);
   %local  ___rc;
   %let ___rc = %eval( %symglobl(&mvar)    ❶
                  + %symlocal(&mvar)*2);  ❷
   &___rc
%mend symchkup;

%* Test;
%put DNE has a rc of %symchkup(DNE);  ❸
%let silly=global; ❹
%put SILLY has a rc of %symchkup(silly); ❺
```

The SAS Log shows that the %SYMCHKUP macro detects the presence of macro variables of various scopes:

```
172  %mend symchkup;
173  %put DNE has a rc of %symchkup(DNE);
DNE has a rc of 0 ❸
174  %let silly=global; ❹
175  %put SILLY has a rc of %symchkup(silly);
SILLY has a rc of 1 ❺
```

❶ %SYMGLOBL will return a 0 or a 1.

❷ %SYMLOCAL will return a 1 if the macro variable exists in any of the existing local tables. This value is multiplied by 2 and the result is added into &RC.

❸ The macro variable &DNE does not exist and %SYMCHKUP returns a 0.

❹ &SILLY is defined in the global symbol table, but does not exist in any local table

❺ %SYMCHKUP returns a 1 indicating that the macro variable exists in the global symbol table.

Because %SYMLOCAL detects a macro variable in any local table, a macro variable that exists in multiple local tables will only be detected once.

MORE INFORMATION: Additional examples of the use of these functions can be found in Section 9.2.1. Similar functions can be found in the DATA step (see Section 8.5.3).

SEE ALSO: Mason (2016) uses %SYMEXIST to check for the existence of macro variables.

8.2 Even More Macro Statements

There are a number of macro language statements that have not been introduced in other sections of this book. Most of these are less commonly used, either because they are not needed as often or as you will see, because of author bias. A few others were only briefly introduced elsewhere and are described in more detail in this section.

8.2.1 Extending the Use of %SYMDEL

The %SYMDEL statement, which was introduced in Section 2.7, is intended to be used to delete macro variables from the GLOBAL symbol table. The statement accepts a list of macro variables that are referenced directly (without the ampersand).

SYNTAX:

```
%SYMDEL list_of_variables </option>;
```

The %SYMDEL statement in Program 8.2.1a removes the macro variables &NADA and &DSN from the GLOBAL symbol table

Program 8.2.1a: Deleting Two Macro Variables Using %SYMDEL

```
%symdel nada dsn;
```

By default a warning is issued if an attempt is made to delete a macro variable that does not exist, however the NOWARN option can be used to suppress this warning.

```
%symdel nada dsn/nowarn;
```

As is shown in Program 8.2.1b, you can use indirect references to specify the macro variable or variables that are to be deleted. Program 8.2.1b demonstrates a usage of an indirect list.

Program 8.2.1b: Using a Macro Variable to Reference a List

```
%let nada=;
%let dsn=clinics;
%let macvarlist = nada dsn xyz;
%symdel &macvarlist / nowarn;
```

%SYMDEL does not offer a lot of flexibility if you want to delete all the macro variables in the global symbol table. However, by first creating a list of all the macro variables, and then using that list as in Program 8.2.1b, you can indeed do so. The code in Program 8.2.1c enables you to dynamically delete all the macro variables in the global symbol table using %SYMDEL.

Program 8.2.1c: Deleting All Macro Variables from the Global Symbol Table

```
proc sql noprint;
   select distinct name
      into :maclist separated by ' '
         from dictionary.macros
            where upcase(SCOPE) eq 'GLOBAL'
                and name ne 'maclist'
/*                and name ne 'SYS_SQL_IP_ALL'*/
/*                and name ne 'SYS_SQL_IP_STMT'*/
    ;
   quit;
%put &=maclist;
%symdel &maclist maclist;
%put _global_;
```

The SQL step places a couple of read-only automatic macro variables in the global symbol table. Since they are read-only they cannot be deleted and the attempt will cause an error.

```
ERROR: Attempt to delete automatic macro variable SYS_SQL_IP_ALL.
ERROR: Attempt to delete automatic macro variable SYS_SQL_IP_STMT.
```

You could prevent this error by excluding these macro variables in the WHERE clause in the SQL step (logic commented out in Program 8.2.1c).

Although it seems less of a problem in the current versions of SAS, the use of a macro variable in the %SYMDEL statement may cause an error due to a timing conflict between the compilation and execution of the statement. If the timing problem is encountered, it can be solved in a couple of different ways. The first is to use quoting functions to control what is resolved first. If you do encounter a problem when using a list such as was done in Programs 8.2.1b and 8.2.1c, you can delay the execution by quoting the %SYMDEL statement keyword.

Program 8.2.1d: Using Quoting to Delay Execution

```
%let nada=;
%let dsn=clinics;
%let maclist = nada dsn;

%unquote(%nrstr(%symdel) &maclist / nowarn);
```

The %NRSTR prevents resolution of the %SYMDEL until after &MACLIST has been resolved. Once &MACLIST has been resolved, the %UNQUOTE removes the quotes and %SYMDEL will be applied to the resolved list of macro variables.

Another solution is to delete the macro variables one at a time by using the CALL EXECUTE routine from within a DATA step. Several variations of this solution have been presented, including ones by SAS Technical Support. The macro %DELVARS shown in Program 8.2.1e, which uses SASHELP.VMACRO and the CALL EXECUTE routine to delete all the macro variables with `SCOPE='GLOBAL'`, is very similar to a macro of the same name, which can be found in SAS Sample 26154.

Program 8.2.1e: Using %SYMDEL with CALL EXECUTE

```
%macro delvars;
  data vars;
    set sashelp.vmacro;
    where scope='GLOBAL' & substr(name,1,3) ne 'SYS';
    if name ne lag(name) then output vars;
    run;
  data _null_;
    set vars;
    call execute('%symdel '||trim(left(name))||'/nowarn;');
    run;
%mend delvars;

%let nada=;
%let dsn=clinics;
%delvars
%put _global_;
```

Notice that since SASHELP.VMACRO is a VIEW that points back to the symbol table(s), it cannot be used in the same DATA step as the CALL EXECUTE. Again, this is a timing issue—a CALL EXECUTE timing issue this time.

If you try to delete macro variables that are not on the global table (perhaps because the variables do not exist or they exist only on a local table), you will get a warning indicating that the macro variable was not found. This warning is suppressed by using the /NOWARN option.

MORE INFORMATION: The %SYMDEL statement was introduced in Section 2.7.

SEE ALSO: Watts (2003a) has an example of the PROC SQL step that prepares a list of GLOBAL macro variables for deletion. Similar examples and discussions have appeared on SAS-L by several authors. Discussion of the use of %SYMDEL and variations of the macro %DELVARS can also be found on the SAS Technical Support page under the FAQ section relating to macros.

diTommaso (2003) also discusses the use of %SYMDEL with a CALL EXECUTE.

An alternative to deleting macro variables that has more flexibility can be found on sasCommunity.org: http://www.sascommunity.org/wiki/Deleting_global_macro_variables.

Langston (2015b) demonstrates the use of %SYMDEL.

8.2.2 Using the %GOTO and %*label* Statements Appropriately

The %GOTO and %*label* statements are included in this book because you might encounter them someday in someone else's code (warning: subtle author bias may be encountered in this subsection). These statements, like other directed branching statements, enable you to create code that is **very** unstructured. So far (when I have tried hard enough), I have always been able to find better ways of solving a problem (both in coding SAS and in my personal life) other than by using GOTO and %GOTO type statements. My first choice is to use alternative logic, thereby avoiding the use of these statements.

Like the DATA step GOTO statement, %GOTO (or %GO TO) causes a logic branch in the processing. The branch destination will be a macro label (%*label*). Therefore, the argument associated with the %GOTO must resolve to a known %*label*.

SYNTAX:

```
%GOTO label;
```

or

```
%GO TO label;
%LABEL:
```

The *label* associated with the %GOTO statement must resolve to a macro label that you have defined somewhere within the macro using the %*label* statement. The label may be explicitly or implicitly named. In the following example, the label is named explicitly. After execution of the %GOTO statement, the next statement to be executed will be the statement following the %NEXTSTEP: label:

```
%GOTO NEXTSTEP;
...code not shown...

%nextstep:
...code not shown...
```

In code that uses %GOTO, it is not unusual for the %GOTO statement to include a label that contains a reference to a macro variable that must be resolved before the %GOTO is executed. In the following example &STEP must resolve to a defined macro label—for example, NEXTSTEP—before the branch can take place. This is often referred to as a *directed* or *computed* %GOTO.

```
%let step = nextstep;

%GOTO &STEP;
```

Because the macro label is preceded by a %, the new user often uses a % with the *label* in the %GOTO statement, as in this statement:

```
%GOTO %NEXTSTEP;
```

Rather than branching to the specified %*label*, however, a call to execute the macro %NEXTSTEP will be issued before the %GOTO can be executed. Generally, this will result in an error, but it could work if the macro %NEXTSTEP resolves to the name of a macro label.

In the following example, %GOTO is used to determine which of two DATA steps will be executed. The macro labels are explicitly defined in the %GOTO statements. Notice that the %*label* statement is followed by a colon and *not* a semicolon.

Program 8.2.2a: Using %GOTO with Explicit Labels

```
%macro mkwt(dsn);
%* Point directly to the label;
%if &dsn = MALE %then %goto male;
  data wt;
  set female;
  wt = wt*2.2;
  run;
%goto next;
%male:
  data wt;
  set male;
  run;
%next:
%mend mkwt;
```

You can rewrite this example to use implicit labels that reflect the incoming macro variable (&DSN). This makes the use of the %IF unnecessary.

Program 8.2.2b: Using %GOTO with Implicit Labels

```
%macro make(dsn);
%* Point indirectly to the label;
%goto &dsn;  %* DSN takes on either MALE or FEMALE;
%female:
  data wt;
  set female;
  wt = wt*2.2;
  run;
%goto next;
%male:
  data wt;
  set male;
  run;
%next:
%mend make;
```

Admittedly, this is a rather simplistic case, but you can generally rewrite programs that use %GOTO to avoid the use of the %GOTO altogether.

Program 8.2.2c: Avoiding the Use of the %GOTO

```
%macro smart(dsn);
  %*AVOID GOTO WHEN POSSIBLE;
  data wt;
  set &dsn;
  %if &dsn=FEMALE %then wt = wt*2.2;;
  run;
%mend smart;
```

A common use of %GOTO is to avoid execution of portions of a macro by skipping to the macro's %MEND statement. To illustrate the point, Program 8.2.2d is a rather silly example of this technique.

Program 8.2.2d: Using %GOTO to Skip to the End of a Macro

```
%macro modfem(dsn);
   %* Execute only for Females;
   %if &dsn ne FEMALE %then %GOTO skip;
   data &dsn;
      set &dsn;
      wt = wt*2.2;;
      run;
   %skip:
%mend modfem;
%modfem(MALE)
```

We could rewrite the %MODFEM macro to avoid the DATA step by using a %DO block just as easily as by skipping to the end of the macro.

When conditions warrant macro termination, rather than skipping to the end of the macro with a %GOTO, the %RETURN statement (see Section 5.4.5) can be used.

Program 8.2.2e: Using %RETURN to Terminate the Execution of a Macro

```
%macro modfem(dsn);
   %* Execute only for Females;
   %if &dsn ne FEMALE %then %return;
   data &dsn;
      set &dsn;
      wt = wt*2.2;;
      run;
%mend modfem;
```

MORE INFORMATION: The %GOTO statement is used in %TRIM, an autocall macro supplied by SAS, which is discussed in Section 10.6.5.

SEE ALSO: The %GOTO statement and %*label* are used by Wang (2003) in a %WINDOW example. Lund (2003a) uses %GOTO to skip the execution of a macro.

8.2.3 Using %WINDOW and %DISPLAY

Through the use of the %WINDOW statement, the macro language provides the programmer with a tool that can be used to establish a basic user interface. Similar to the WINDOW statement in the DATA step, %WINDOW can be used to create and display message boxes and to collect information from the user that can then be placed into macro variables.

The %WINDOW statement can be used to do the following:

- display a window
- control window attributes including size and color
- make use of existing key and menu definitions
- display existing macro variable values

- define and assign values to macro variables

Once a window has been defined with the %WINDOW statement, it can then be displayed by using the %DISPLAY statement. The %WINDOW and %DISPLAY statements can be used in open code.

SYNTAX:

```
%WINDOW window-name <attributes and display characteristics>;
%DISPLAY window-name <display control options>;
```

Because %WINDOW can be used to create macro variables, it can be useful when having the user specify execution time specific parameters without editing the program. The macro %DSNPROMPT in Program 8.2.3a defines and then displays a macro window, which prompts the user for the name of a data set within the declared library.

Program 8.2.3a: Using %WINDOW to Prompt for a Data Set Name

```
%macro dsnprompt(lib=sasuser);
%* prompt user to for data set name;
%window verdsn color=white    ❶
  #2 @5 "Specify the data set of interest" ❷
  #3 @5 "for the library &lib"  ❸
  #4 @5 'Enter Name: '
         dsn 20 ❹ attr=underline required=yes ❺;

%display verdsn; ❻

title1 "8.2.3a Print the &lib..&dsn data set";
proc print data=&lib..&dsn;
run;
%mend dsnprompt;

%dsnprompt(lib=macro3)
```

When the %DSNPROMPT macro is executed, the VERDSN macro window will be defined and displayed.

❶ The VERDSN window will have a white background with no specifications for size.

❷ The text (in single or double quotes) is to be displayed at row 2 and column 5 of the window. The same notation for row (#) and column (@) is used as in the PUT and INPUT statements.

❸ Macro variables can be included in the text (see caveat below).

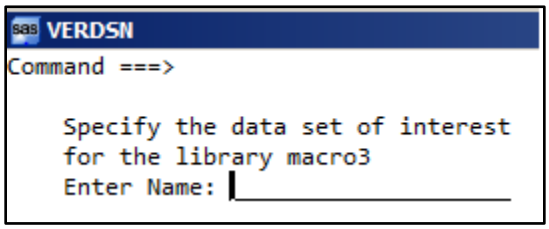
❹ The user is prompted for the name of a data set which is placed into &DSN.

❺ Attributes can be assigned to the display of the macro variable.

❻ Although defined by the %WINDOW statement, the VERDSN window is not displayed until the %DISPLAY statement is executed.

The VERDSN window defined and displayed in the %DSNPROMPT macro is shown in Figure 8.2.3a.

Figure 8.2.3a: Prompting for a Data Set Name



CAVEAT: In Program 8.2.3a the VERDSN window is defined when the macro is executed. It is during this definition phase that &LIB at ❸ is resolved. If &LIB is redefined at a later time (after the %WINDOW statement has executed), then %DISPLAY will still show the original value, not the current value of &LIB. Appendix 2 has additional examples of instances where macro variables are resolved during the compilation phase.

There are a number of attributes that can be associated with a macro window. The window definition shown in Program 8.2.3b specifies the background and foreground colors, the location, and the size of the window.

Program 8.2.3b: Prompting for a Two-Digit Year

```
%window getyear
   color=magenta ❼
   icolumn=15 ❽
   irow=10
   columns=30 ❾
   rows=15
   #3 @3  'Enter the two digit year' color=white ❼
   #4 @5 yr 2 color=white attr=underline required=yes ❿
   #5 @3 'Then press "Enter" '  color=white
   ;
%display getyear;
```

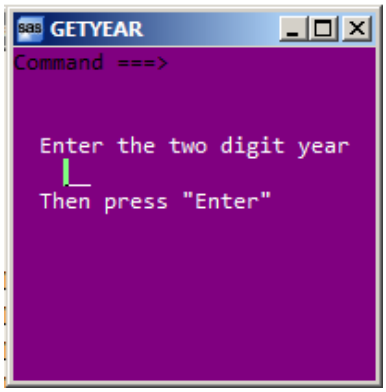
❼ Color attributes are specified using the COLOR= option for both the background and foreground colors.

❽ The upper left corner is located using the ICOLUMN= and IROW= options.

❾ The window size is control by using COLUMNS= to specify width and the ROWS= option to specify height.

❿ The incoming macro variable (&YR) value is allowed two characters and is required.

Figure 8.2.3b: Prompting for a Two-Digit Year



SEE ALSO: Many of the options associated with %WINDOW are introduced and discussed by Alden (2000) and Mace (1997, 1998, 2000, 2002, and 2003). These papers provide very nice overviews as well as detailed (and in most cases more sophisticated) examples of macro windows.

Gau (1999) presents an example of the %WINDOW to manage programs. Ren (1999), Parker (2000), Dynder, Cohen, and Cunningham (2000), Fahmy (2003), Huang (2003), Wang (2003), Parker (2003), and Rhoads and Letourneau (2002) each use a macro window to create user interfaces. Plath (2002) creates and executes a series of macro windows to collect information from the user.

Access control to macros is achieved through the use of the WINDOW statement in an example by Shilling and Kelly (2001).

Glass and Hadden (2016) use the %WINDOW and %DISPLAY statements to collect information from the user of a SAS program.

8.2.4 Extending %SYSEXEC with Examples

The %SYSEXEC statement enables you to execute operating system commands and statements from within the macro language. The macro %MAKEDIR in Program 8.2.4 can be used to verify that a directory exists, and, if it does not already exist, to create it. The only parameter used by %MAKEDIR is the directory path to be checked.

Program 8.2.4: Verify that a Directory Exists Using %SYSEXEC

```
%macro makedir(newdir);
  %local rc;
  %* Make sure that the directory exists;
  %let rc = %sysfunc(fileexist(&newdir)); ❶
  %if &rc=0 %then %do;
     %put Creating directory &newdir;
     %* Make the directory;
     %sysexec  md &newdir; ❷
  %end;
  %else %put Directory &newdir already exists;
%mend makedir;

options noxwait; ❸
%makedir(c:\tempzzz)
```

❶ The FILEEXIST function is used to see if the "file," which in this case is actually a directory, exists. The return code from this function is 0 if the file is not found.

❷ A return code of 0 indicates that the directory does not exist and should be created. %SYSEXEC is used to execute the Windows MD (make directory) command.

❸ Under Windows the X statement and the %SYSEXEC statement both open a DOS window. Without specifying the NOXWAIT system option you will need to close that window manually.

The advantage of the %SYSEXEC macro statement over the X statement is that you do not need to leave the macro environment before executing the operating system command. By using the %SYSEXEC statement the %MAKEDIR macro mimics a macro function. If an X statement had been used the macro would have had a more limited utility.

When the command issued by %SYSEXEC is executed by the operating system, the success or failure of that OS command is returned to SAS as a code, which is stored in the automatic macro variable &SYSRC. A value of 0 indicates success. In the code that follows, the %MAKEDIR macro attempts to create a directory on the Z: drive, which for this example does not exist.

```
%makedir(z:\tempzzz)
%put Return Code: &sysrc ;
```

The SAS Log shows that &SYSRC will contain a 1 indicating that the directory was not created:

```
62   %makedir(z:\tempzzz)
Creating directory z:\tempzzz
63   %put Return Code: &sysrc ;
Return Code: 1
```

MORE INFORMATION: The %SYSEXEC statement is introduced in Section 5.4.3 and is used in Program 8.1.1e.

SEE ALSO: Dynder, Cohen, and Cunningham (2000) use a %WINDOW interface to generate a series of directories. The FILEEXIST function is also used by Lund (2003a) to check and establish directories.

Jia(2015) uses %SYSEXEC to create a directory.

8.2.5 Deleting Macro Definitions with %SYSMACDELETE

Unless stored compiled macro libraries are being used, the compiled macro is stored in the WORK.SASMACR catalog. As a general rule, it does not matter how many macros are in this catalog or even whether a given macro already exists if it is to be recompiled, however there are instances when it does make a difference (Sun and Carpenter, 2011). Although you can *generally* delete entries from the WORK.SASMACR catalog manually using the Display Manager or other SAS interfaces, this is neither a recommended nor a supported technique.

The %SYSMACDELETE statement is the supported tool for deleting compiled macros from the WORK.SASMACR catalog.

SYNTAX:

```
%SYSMACDELETE macro_name </NOWARN>;
```

You can only delete one macro for each instance of the %SYSMACDELETE statement, and the NOWARN option can be used to suppress warnings if you try to delete a macro that is either currently executing or does not exist in the WORK.SASMACR catalog.

Program 8.2.5: Using the %SYSMACDELETE Statement to Delete a Macro Definition

```
%macro silly0;
%* silly0;
%mend silly0;
%macro silly1;
%* silly1;
%mend silly1;
%macro silly2;
%* silly2;
%mend silly2;
%sysmacdelete silly0; ❶
%sysmacdelete silly1 silly2; ❷
%sysmacdelete silly3; ❸
%sysmacdelete silly3 /nowarn; ❹
```

After execution of Program 8.2.5 the SAS Log shows the following:

```
113  %sysmacdelete silly0; ❶
114  %sysmacdelete silly1 silly2; ❷
WARNING: Extraneous argument text on %SYSMDELETE call ignored:  SILLY2
115  %sysmacdelete silly3; ❸
WARNING: Attempt to delete macro definition for SILLY3 failed. Macro
definition not found.
116  %sysmacdelete silly3 /nowarn; ❹
```

❶ The definition for %SILLY0 is deleted.

❷ A warning is issued, because of the second name (%SILLY2). Even with the warning, the definition of the first macro named (%SILLY1) is deleted.

❸ %SILLY3 does not exist and a warning is issued in the SAS Log.

❹ %SILLY3 does not exist, but a warning is not issued because of the /NOWARN option.

SEE ALSO: Langston (2015b) demonstrates the use of %SYSMACDELETE.

8.2.6 Making Macro Variables READONLY

In Section 5.4.2 the %GLOBAL and %LOCAL statements are introduced along with the concept of macro variable collisions. These collisions occur when a macro variable assignment inadvertently overwrites the value of another macro variable with the same name in a different symbol table. The READONLY options on the %GLOBAL and %LOCAL statements are designed to mitigate some of the issues associated with macro variable collisions. They are not a panacea, however they can be very helpful when you need to protect one or more of your macro variables.

SYNTAX:

```
%GLOBAL/READONLY varname=value;
%LOCAL/READONLY varname=value;
```

In Program 8.2.6 the macro variable &MYPATH is declared to be global and to be READONLY.

Program 8.2.6: Declaring a Global Macro Variable READONLY

```
%global/readonly mypath = &path; ❶
%put _user_; ❷
%let mypath = abc; ❸
%global/readonly mypath = abc; ❸
%symdel mypath;❸
```

❶ The macro variable &MYPATH is declared to be a read-only global macro variable and assigned the value stored in &PATH

❷ %PUT is used to show the user-defined macro variables. Notice that the SAS Log does not indicate that &MYPATH has been declared to be READONLY.

❸ Once declared to be READONLY the macro variable &MYPATH cannot be assigned a new value, nor can it be deleted from the global symbol table.

Program 8.2.6 (SAS Log): Showing the Usage of the /READONLY Option

```
127  %global/readonly mypath = &path; ❶
128  %put _user_; ❷
GLOBAL MYPATH C:\Primary
GLOBAL PATH C:\Primary
129  %let mypath = abc; ❸
ERROR: The variable MYPATH was declared READONLY and cannot be modified or
re-declared.
130  %global/readonly mypath = abc; ❸
ERROR: The variable MYPATH was previously declared as READONLY and cannot
be re-declared.
131  %symdel mypath; ❸
ERROR: The variable MYPATH was declared READONLY and cannot be deleted.
```

When using the READONLY option on the %GLOBAL or %LOCAL statements, you cannot assign values to more than one macro variable at a time. In the following code an attempt is made to assign values to the macro variables &A, &B, and &C.

```
%global/readonly a=a b=b c=c;
```

In actuality only one macro variable is assigned &A, and %PUT _USER_ shows that &A contains the value of `a b=b c=c`.

CAVEAT: In Program 8.2.6 &MYPATH is declared to be a READONLY macro variable in the global symbol table. This declaration also precludes the use of this macro variable name in any local symbol table as well. In fact, the declaration of a READONLY macro variable prevents the use of that name in any other symbol table. READONLY macro variables persist until the end of the SAS session in which they are created.

8.3 Even More Automatic Macro Variables

A number of automatic macro variables were introduced in Section 2.6 as well as elsewhere within this book. Depending on how you use SAS and how you use the macro language, these macro variables will have varying utility to you. However, you need to have an understanding of what is available to you so that you can take full advantage of the ones that are actually valuable to you.

You can view the list of currently defined automatic macro variables along with their values by using the %PUT statement and the _AUTOMATIC_ option.

```
%put _automatic_;
```

This section describes some of the less commonly used, but no less valuable, automatic macro variables.

8.3.1 Passing VALUES into SAS Using &SYSPARM

The value of the SYSPARM system option can be loaded during the SAS initialization phase. Because this option can be used as a SAS initialization option, the value itself can be supplied before SAS is executed. This gives us the ability to pass values into SAS from an outside process or program. The value stored in this system option can be retrieved in a number of ways including the SYSPARM( ) DATA step function and the automatic macro variable &SYSPARM.

Because this option is most useful when its value is loaded when SAS is initially executed, it is most commonly used when SAS is executed in a batch execution environment. The SYSPARM initialization option specifies a character string that can be passed into SAS programs. The maximum length of this macro variable is 32K characters.

In the following example, you would like your programs to automatically direct your data to either a test or production library. To make this switch, assign &SYSPARM the value TST or PROD when you start the SAS session.

Assume that an Open VMS SAS session is initiated with:

```
$ sas/sysparm=tst
```

A typical LIBNAME statement on Open VMS which uses this value might be:

```
libname projdat "usernode:[study03.gx&sysparm]";
```

The resolved LIBNAME statement becomes:

```
libname projdat "usernode:[study03.gxtst]";
```

The syntax that you use to load a value into &SYSPARM depends on the operating environment that you are using. See the SAS Companion for your operating environment for more information. When using a shortcut under Windows, `-sysparm tst` appears on the TARGET LINE in the Properties Window of the shortcut. In JCL, the option is used on the SYSIN line.

The DATA step function SYSPARM() can also be used to retrieve the value of the SYSPARM system option. Depending on how this function is used it might not return the same value as &SYSPARM. The differences between using &SYSPARM directly and the SYSPARM () function are demonstrated in the

following example. Notice in this example that the value of the SYSPARM option contains a macro variable reference.

Initialize SAS using the -SYSPARM option.

```
"C:\... path not shown ...\9.4\sas.exe" -sysparm &aaa; ❶
```

Once initialized, &SYSPARM can be used throughout the session.

Program 8.3.1a: Returning &SYSPARM Values

```
%let aaa = AAAAA;
data try2;
a = "&sysparm"; ❷
b = sysparm();  ❸
put a=;
put b=;
run;
```

The SAS Log shows how the values are assigned to the variables A and B:

```
a=AAAAA; ❷
b=&aaa; ❸
```

❶ Usually, as in this example, the value for &SYSPARM is set when SAS is first invoked. Since at this point the code has not even been sent to the word scanner, the macro processor is not called, and therefore, no attempt is made to resolve &AAA. As a result, &SYSPARM contains the characters &aaa. If &SYSPARM contains a blank, and therefore more than one word, **double** quotes should be used.

❷ In the data set TRY2 the variable A is a character variable, which has a length of 5, and contains the value "AAAAA". Before a value can be assigned to the variable A, &SYSPARM is first resolved to &aaa. This is in turn resolved to AAAAA, and it is this value that is then stored in the data set variable A.

❸ While the variable B is also a character variable, it will, by default, have a length of 200 (this is the default length returned when using the SYSPARM function). Since the SYSPARM() function is executed during the DATA step execution phase, the value "&aaa" will be written directly to the variable B and no attempt will be made to resolve the macro reference.

If &SYSPARM contains more than 200 characters be sure to use the LENGTH statement to set the length of the variable created by the SYSPARM function, otherwise longer values will be truncated.

There is an interesting relationship between the –SYSPARM initialization option, the automatic macro variable &SYSPARM, and the SYSPARM system option. It turns out that updating any one of the three, changes the values of the others. This is good because this means that regardless of which method you use to retrieve the stored value, you will always get the same value. Program 8.3.1b demonstrates this relationship by showing that changing either changes both.

Program 8.3.1b (SAS Log): Showing the Relationship between &SYSPARM and the SYSPARM System Option

```
4    %let sysparm=something; ❹
5    options sysparm=' '; ❺
6    %put &=sysparm;
SYSPARM=  ❻
7    %let sysparm=def; ❼
8    data a;
9       x = sysparm(); ❽
10      y = getoption('sysparm'); ❾
11      z = "&sysparm"; ❿
12      put x= y= z=;
```

```
13      run;

x=def y=def z=def  ❽ ❾ ❿
```

❹ We make sure that &SYSPARM has a value using a %LET statement. This value could also have been set using the SAS initialization option (-SYSPARM).

❺ Changing the value of the SYSPARM system option also changes the value of &SYSPARM as is shown using a %PUT at ❻.

❻ The value of &SYSPARM is written to the SAS Log. The value of &SYSPARM was changed when the system option was updated ❺.

❼ The value of &SYSPARM is changed again.

❽ The SYSPARM function returns the value stored in the SYSPARM system option. The default length of X is $200.

❾ The GETOPTION function returns the value of the SYSPARM system option. The default length of Y is $200.

❿ The &SYSPARM macro variable is resolved before the assignment is made. The length of Z will be $3.

CAVEAT: Not all operating systems are the same. Consult your SAS Companion for details or limitations on the number of characters that can be passed into &SYSPARM.

SEE ALSO: Johnson (2001) has an example that parses several words out of a single &SYSPARM value. Wong (2002) shows examples of both the SYSPARM macro variable and the SYSPARM() function.

8.3.2 Learning More about Deciphering Errors

When you encounter problems with the execution of various components of your macro, there are a number of automatic macro variables that you can inspect to try to get a handle on the coding problem.

SEE ALSO: Hughes (2016a) uses &SYSERR and &SYSERRORTEXT to examine errors associated with a failed SORT step. Billings (2015) describes a strategy for detecting errors, including the use of &SYSERR and &SQLRC.

&SYSERR, &SYSERRORTEXT, and &SYSWARNINGTEXT

The automatic macro variable &SYSERR, introduced in Section 2.6.3, is likely to be one of the first automatic macro variables that you might want to check. Because the codes stored in &SYSERR are cryptic, the automatic macro variables &SYSERRORTEXT and &SYSWARNINGTEXT contain the latest error and warning messages written to the SAS Log.

In a variation of the PROC DATASETS example shown in Section 2.6.3, the macro %COPYALL shown in Program 8.3.2a will check &SYSERR to see if the copy was successful. The resulting error codes are written to the SAS Log.

Program 8.3.2a: Checking for PROC Step Errors

```
%macro copyall(inlib=, outlib=);
proc datasets memtype=data;
   copy in=&inlib out=&outlib;
   quit;
%if &syserr>5 %then %do;
   %put ERROR: &=syserrortext; ❶
   %put ERROR: &=syserr; ❷
   %abort;
%end;
%put Copy was successful;
```

```
%mend copyall;
%copyall(inlib=combine, outlib=combtemp)
```

Because the COMBINE *libref* does not exist, the PROC step must fail. The SAS Log shows that &SYSERR takes on a value greater than 5, and &SYSERRORTEXT displays an explanation of this code:

```
158  %copyall(inlib=combine, outlib=combtemp)

ERROR: Libref COMBINE is not assigned.
NOTE: Statements not processed because of errors noted above.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE DATASETS used (Total process time):
      real time           0.02 seconds
      cpu time            0.03 seconds

ERROR: SYSERRORTEXT=Libref COMBINE is not assigned. ❶
ERROR: SYSERR=1008 ❷
ERROR: Execution terminated by an %ABORT statement.
```

❶ The text associated with the error is written to the SAS Log.

❷ The return code, which is stored in &SYSERR is written to the SAS Log.

When the %ABORT statement executes the %COPYALL macro terminates, and in this case the %PUT indicating that the copy was successful will not be executed.

Although the DATASETS procedure returns multiple codes (0=success, 1-4 are warnings, and greater than 4 are errors), most steps return a 0/1. This means that you will generally have a %IF statement that checks for &SYSERR values > 0:

```
%if &syserr>0 %then %do;
```

Clearly, using &SYSWARNINGTEXT and &SYSERRORTEXT to parrot back messages to the SAS Log is not particularly helpful. However, if you parse the contents of &SYSERRORTEXT for specific text you can have your macro take specific action. The %IF statement shown here (which is taken from Program 8.3.2b which is not shown), detects that a *libref* has not been established and calls a macro that creates it.

Program 8.3.2b (Partial): Checking Error Text to Make Decisions

```
%if %bquote(&syserrortext) =%bquote(Libref %upcase(&inlib) is not
assigned.) %then %makelib(&inlib);
```

CAVEAT: Be very careful when making decisions based on the text values that are contained in &SYSERRORTEXT and &SYSWARNINGTEXT. These are READONLY macro variables and they are not reset between step boundaries. Their values only change when a new error or warning is encountered. This means that the value of one of these macro variables could easily have been set in some prior step or even from an earlier program if you are running interactively. This makes the text checking such as was done in Program 8.3.2b somewhat impractical – unless, of course, you are very careful.

MORE INFORMATION: The %ABORT statement is introduced in Section 5.4.4. This statement includes options that determine the overall impact of this statement.

SEE ALSO: Shtern (2014) uses the CANCEL option on the %ABORT statement to terminate the SAS session.

Failure to copy can occur when a data set is locked. Hughes (2014a) carefully describes various locking situations as well as a macro to detect and avoid failures due to locks. Other descriptions of data set locks can be found in Graham and Osowski (2013) and Galligan (2011).

&SYSCC

Step condition codes can also be examined using &SYSCC. Unlike &SYSERR, which is a READONLY variable, the value of &SYSCC can be reset by the user. In Program 8.3.2c the %RUNCHECK macro is used as a special batch process RUN statement, which automatically terminates the SAS process if the condition code exceeds the specified value for that step. In this program the %ABORT statement (see Section 5.4.4) includes the use of the ABEND option, which, when running interactively outside of a SAS/AF session, causes the SAS session to end.

Program 8.3.2c: Checking Condition Codes Using &SYSCC

```
%macro RunCheck(codeval);
run;    /* terminate the step */  ❸
%if &syscc > &codeval %then %do;  ❹
   %put ERROR: Condition Code &syscc exceeds &codeval;
   %put Aborting Process;
   %abort abend;
%end;
%else %if &syscc>0 %then %do; ❺
   %put WARNING: Condition Code &syscc within limits;
   %let syscc=0;
%end;
%else %do;
   %let syscc=0;
%end;
%mend runcheck;

proc print data=sashelp.class;
   var name ht wt; ❻
   %runcheck(500)
proc print data=sashelp.class;
   var name height weight;
   %runcheck(0)
```

❸ Terminate the previous step with a RUN; statement.

❹ If the value of &SYSCC exceeds the specified tolerance write a message to the SAS Log and terminate the process using a %ABORT statement.

❺ Although &SYSCC exceeds 0, if it is not above the tolerance level for the step, therefore a warning is written to the SAS Log.

❻ The variables HT and WT are not on the data set SASHELP.CLASS. &SYSCC takes on the value of 3000, which exceeds the tolerance and the SAS session is aborted.

Function Return Codes and the SYSMSG Function

The success or failure of function calls can also be evaluated within the macro language. In Program 8.3.2d the LIBNAME function is used to assign the *libref* TEMXX to a location (`C:\TEMPXX`) which does not exist. The SYSMSG() function returns the text associated with the most recent function call.

Program 8.3.2d (SAS Log): Showing a Function Return Code and Its Message

```
180  %let rc = %sysfunc(libname(temxx,c:\tempxx)); ❼
181  %put &rc %sysfunc(sysmsg()); ❽
-70008 NOTE: Library TEMXX does not exist. ❾
```

❼ Since the LIBNAME function does not normally return a value, we can instead capture its return code in &RC.

❽ Write the function's return code and its associated message to the SAS Log.

❾ The SYSMSG() function will return the text associated with the call to the LIBNAME function.

MORE INFORMATION: The code in Program 8.3.2d is used in Programs 11.2.6a and 12.3.3 where the SYSMSG function writes error messages when the LIBNAME function fails. There are two automatic macro variables that will capture the success or failure of LIBNAME and FILENAME statements, see Section 8.3.6.

Capturing SQL Step Boundary Errors Using &SQLRC

Because PROC SQL executes at the statement level, we may need to be able to evaluate the success or failure of individual statements within a PROC SQL step. To do this we can use the automatic macro variable &SQLRC. This macro variable is reset following the execution of each PROC SQL statement.

Program 8.3.2e (SAS Log): Showing Errors at SQL Boundaries

```
226  proc sql ;
227  %put &=sqlrc;
SQLRC=0
228  create table class as
229     select *
230        from sashelp.clss; ❶
ERROR: File SASHELP.CLSS.DATA does not exist.
231  %put &=sqlrc;
SQLRC=8 ❷
232  create table class as
233     select *
234        from sashelp.class;
NOTE: Table WORK.CLASS created, with 19 rows and 5 columns.

235  %put &=sqlrc;
SQLRC=0 ❸
236  quit;
NOTE: The SAS System stopped processing this step because of errors. ❹
```

❶ The incoming data set name has been misspelled.

❷ &SQLRC contains a nonzero value indicating something other than success.

❸ The data set is spelled correctly and &SQLRC contains a 0.

❹ Being able to capture the return code within a step can become important. Notice here that although the NOTE indicates that the step was stopped, it clearly was not as the data set WORK.CLASS was created.

SEE ALSO: Additional detail about using automatic macro variable return and completion codes can be found in a very detailed paper by Hughes (2014b).

Examining Errors Codes Stored in &SYSINFO

While all procedure steps can be checked using the &SYSERR macro variable, some procedures, routines, statements and functions will also provide a return code in the automatic macro variable &SYSINFO. PROC COMPARE is one of those steps.

In the COMPARE step in Program 8.3.2f two very different data sets are compared. &SYSERR detects that warnings were issued, while &SYSINFO has a more specific return code.

Program 8.3.2f: Examining &SYSINFO

```
proc compare data=sashelp.shoes comp=sashelp.class;
run;
%put &=syserr;
%put &=sysinfo;
```

The SAS Log shows the values of &SYSERR and &SYSINFO:

```
271  %put &=syserr;
SYSERR=4
272  %put &=sysinfo;
SYSINFO=3073
```

Possible values for &SYSINFO can be found in the documentation for the procedures that use this macro variable.

SEE ALSO: Cheng et. al. (2015) uses &SYSINFO with a PROC COMPARE step.

8.3.3 Taking Advantage of the Parameter Buffer

A buffer is a temporary storage location that can be used to store or pass information. Macro parameter buffers enable you to create macros with a variable number of parameters. The PARMBUFF (or PBUFF) switch is used to turn the parameter buffer on, and the automatic macro variable &SYSPBUFF is used to hold the buffer's value.

The /PARMBUFF switch is used on the %MACRO statement to turn on the ability to load the macro variable &SYSPBUFF when the macro is called. The macro %DEMO in Program 8.3.3a demonstrates the process that you will use when taking advantage of &SYSPBUFF.

Program 8.3.3a: Demonstrating the Use of the PARMBUFF Switch on the %MACRO Statement

```
%macro demo(a=1, b=2)/parmbuff; ❶
  %put buffer holds |&syspbuff|; ❷
  %put &=a; ❸
  %put &=b; ❹
%mend demo;

%demo(a=aa)  ❺

%demo(a=silly, d=unknown) ❻
```

The macro %DEMO is called twice, and the SAS Log shows the following:

```
35   %demo(a=aa) ❺
buffer holds |(a=aa)| ❷
A=aa ❸
B=2 ❹
36
37   %demo(a=silly, d=unknown) ❺
buffer holds |(a=silly, d=unknown)| ❷
A=silly ❸
B=2 ❹
```

❶ The macro statement shows two keyword parameters and the /PARMBUFF switch.

❷ This %PUT writes the contents of &SYSPBUFF to the SAS Log. The value of &SYSPBUFF contains the parameters of the macro call, including the parentheses, just as they are coded. This includes extra spaces, commas, and other characters.

❸ The value of &A has been passed into the macro as a keyword parameter, and as is usual, it is stored in the macro variable.

❹ Since &B is not included in the macro call, its value is not included in &SYSPBUFF, and the value of &B remains at its default value.

❺ The macro is called a second time, and the macro parameter list is passed to the macro and stored, including the parentheses, in &SYSPBUFF.

❻ Since the parameter values being passed to the macro are coming in through the buffer, you can call the macro using parameters that do not exist. This call to %DEMO runs without error. However, the parameter &D will not be defined unless you explicitly write code to parse &SYSPBUFF. This enables you to build a macro with a variable number of parameters, and this is done in the macro %IN in Program 8.3.3c.

Although &SYSPBUFF is an automatic macro variable it is stored in the local symbol table. This means that a local symbol table will always exist when using the /PARMBUFF switch. Although on the local table, &SYSPBUFF is an automatic macro variable and can be shown using the `%PUT _AUTOMATIC_;` statement rather than the `%PUT _LCOAL_;` statement.

Program 8.3.3b shows that &SYSPBUFF is local, but it also highlights inconsistencies when using the %SYMGLOBL and %SYMLOCAL functions. These inconsistencies have been fixed in SAS 9.4M3.

Program 8.3.3b: Showing that &SYSPBUFF Is Local

```
%macro test/pbuff;
  %put &syspbuff;
  %put %symexist(syspbuff);
  %put %symglobl(syspbuff);
  %put %symlocal(syspbuff);
%mend test;
%test(abc)
%put &=syspbuff;
```

The SAS Log for %TEST shows that:

```
92   %test(abc)
(abc)
1
1
0
93   %put &=syspbuff;
WARNING: Apparent symbolic reference SYSPBUFF not resolved.
```

The PARMBUFF option is used in Program 8.3.3c to create a macro function that can be used to build a highly flexible IN operator for the DATA step IF statement. It enables you to check a character variable against a list of values of varying lengths. In addition, you can optionally match only the first few characters of the string. It does this by building an IF expression of the following form:

```
if (code eq 'a1' or code eq 'a2' or code eq 'a3') then....
```

The variable that is to be checked (code) is the first parameter in the macro call and the remaining parameters form the values (a1, a2, and a3).

Program 8.3.3c: Using PARMBUFF to Build a Flexible IN Operator

```
%macro in() / parmbuff; ❶
  %local parms var numparms infunc i thisparm;
  %let parms = %qsubstr(&syspbuff,2,%length(&syspbuff)-2);  ❷

  %let var = %scan(&parms,1,%str(,)); ❸

  %let numparms = ❹ %eval(%length(&parms) -
                    %length(%sysfunc(compress(&parms,%str(,)))));

  %let infunc = &var eq %scan(&parms,2,%str(,)); ❺
```

```
  %do i = 3 %to (&numparms + 1); ❻
    %let thisparm = %scan(&parms,&i,%str(,)); ❼
    %let infunc = &infunc or &var eq &thisparm; ❽
  %end;

  (&infunc) ❾
%mend;
```

Source: Pete Lund, Looking Glass Analytics

❶ The macro is specified without any parameters. The information needed by the macro will come in through &SYSPBUFF. Although the name %IN is currently not reserved (SAS 9.4), it *will* become a reserved word in the future. It is recommended that %IN not be used as a macro name so as to improve compatibility with future releases of SAS.

❷ The parentheses that are automatically included with &SYSPBUFF are stripped off.

❸ The first parameter is the variable name that will be checked.

❹ Count the number of parameters by counting the number of commas. In this statement the commas are counted by comparing the length of &PARM with its length after the commas have been compressed out. The number of parameters, &NUMPARMS, is one too small because there is one more parameter than there are commas and the first parameter is actually the variable name.

The number of parameters could have also been calculated using the COUNTW function, which was not available when this macro was originally written.

```
%let numparms = %sysfunc(countw(&parms));
```

❺ The macro variable &INFUNC will be used to hold the expression that is being built. This statement creates the first expression by equating the name of the variable, &VAR, with the first parameter value (second word in the list).

❻ Loop through the remaining parameters (this macro expects at least two comparison parameters).

❼ Select the next value from the parameter list.

❽ Add this comparison onto the growing list in &INFUNC.

❾ Use &INFUNC to pass the list of comparisons back to the IF statement.

In the call to the %IN macro below, an IF statement will be built that will check a variable (CPT) against the following character values '4300', '4301', '44xx', '451x'. Here x represents a wildcard value, which is established by placing the colon operator before those values that include partial strings. Notice that these are character values and the quotes are passed into the macro.

```
If %in(cpt,'4300','4301',:'44',:'451') then...
```

The previous macro call would generate the following code:

```
if (cpt eq '4300' or cpt eq '4301' or cpt eq :'44' or cpt eq :'451') then
...
```

Since &SYSPBUFF is being used to pass the parameters, the macro call can contain any number of comma-separated values.

The macro %ORLIST in Program 8.3.3d is similar to the macro %IN in Program 8.3.3c as it also uses the PARMBUFF switch; however, it parses &SYSPBUFF differently. The %DO %WHILE loop is used to pass through the list of parameter values and one by one the variable/value pairs are added to &ORLIST.

Program 8.3.3d: Parsing &SYSPBUFF with the %QSCAN Function

```
%macro ORlist() / pbuff;
   %local datvar i parm orlist;
   %let datvar = %qscan(&syspbuff,1,%str(%(,)); ❶
   %let i = 1;
   %do %while(%qscan(&syspbuff,&i+1,%str(,%(%))) ne %str());
      %let parm = %qscan(&syspbuff,&i+1,%str(,%(%))); ❷
      %if &i=1 %then %let orlist = &datvar=&parm; ❸
      %else %let orlist = &orlist or &datvar=&parm;
      %let i = %eval(&i + 1);
   %end;
   &orlist ❹
%mend orlist;
```

The macro builds a series of logical comparisons separated by the Boolean OR operator. Typical usage would be within an IF statement. Here a %PUT is used to show in the SAS Log what the IF statement would look like after the macro is called:

```
157  %put If %orlist(cpt,'4300','4301',:'44',:'451') then...;
If cpt='4300' or cpt='4301' or cpt=:'44' or cpt=:'451' then...
```

❶ The variable name is retrieved as the first word. The %(is used to mark the open parenthesis as a word delimiter along with the comma.

❷ %QSCAN is used to separate the values. Notice the use of %(and %), as well as the comma, to designate the open and close parentheses as word delimiters (this prevents them from becoming a part of the first and last words selected by %QSCAN. (See Section 7.1.9 for a discussion of the marking of special characters.)

❸ The list of comparisons is temporarily stored in &ORLIST.

❹ The resulting list of comparisons is passed back, and replaces the macro call with the resultant list of comparisons.

SEE ALSO: Mace (1999) briefly discusses the automatic macro variable &SYSPBUFF. A more detailed discussion of %IN and other user-written macro functions can be found in Lund (1998, 2000a, 2000b, and 2001c). The /PARMBUFF switch and &SYSPBUFF macro variable are used by Lund (2000a) to build a formatted comment for the SAS Log.

8.3.4 Using &SYSNOBS as an Observation Counter

When processing a DATA step it is often handy to be able to capture the number of observations written to the new data set. The macro variable &SYSNOBS will contain the number of observations written to the last data set closed by a DATA step. In Program 8.3.4 the macro variable &SYSNOBS will contain the number of observations in WORK.WANT.

Program 8.3.4: Using &SYSNOBS to Indicate the Number of Observations in a Data Set

```
data want;
   set sashelp.class(where=(name>'B'));
   run;
%put &=sysnobs;
```

Similar to &SQLOBS, which counts observations in SQL steps, this macro variable will be reset by the next DATA step, but it will not be reset by procedure steps, even an SQL step that creates a data table.

If your DATA step creates more than one table, the observation count of only one of the tables (the last one to be closed) is reflected in &SYSNOBS. Generally, this will be the right most table listed in the DATA statement.

MORE INFORMATION: The macro %OBSCNT (see Program 11.2.6) will also return the number of observations in a data set.

8.3.5 Using &SYSMACRONAME

The automatic macro variable &SYSMACRONAME contains the name of the most local macro that is currently executing. Section 8.1.3 examines the use of four different macro functions that can be used to surface names as well as nesting levels. However, if you only need to know the name of the innermost currently executing macro, then &SYSMACRONAME is available for your use.

Program 8.3.5 demonstrates how the value stored in &SYSMACRONAME changes depending on which macro is executing. When there are nested macro calls only the name of the inner most macro is revealed.

Program 8.3.5: Showing the Name of the Currently Executing Macro

```
%macro inner;
%put inner &sysmacroname;
%mend inner;
%macro test;
%put in test before inner: &sysmacroname;
%inner
%put back in test: &sysmacroname;
%mend test;
%test
%put in open code: &sysmacroname;
```

The SAS Log shows that the value of &SYSMACRONAME is updated as the macro being executed changes:

```
246  %test
in test before inner: TEST
inner INNER
back in test: TEST
247  %put in open code: &sysmacroname;
in open code:
```

MORE INFORMATION: Macro functions that can be used to determine macro nesting as well as the name of the currently executing macro are described in Section 8.1.3.

SEE ALSO: McMullen (2012) uses &SYSMACRONAME in a macro that tests data assertions.

8.3.6 Using &SYSLIBRC and &SYSFILRC

Whenever you attempt to create a *libref* or a *fileref* a return code is generated. You can view the success or failure of the operation by examining this return code, which is stored in either &SYSLIBRC or &SYSFILRC. Success is indicated by the return of a 0. A nonzero integer is returned when the LIBNAME or FILENAME statement is not successful.

Program 8.3.6 demonstrates various aspects of the use of the &SYSLIBRC macro variable (usage of &SYSFILRC is similar).

Program 8.3.6: Checking the Success of a LIBNAME Statement

```
* This library location does not exist;
libname mytemp "c:\temploc";
%put Zero is success: &syslibrc;
```

The SAS Log shows that in this usage the location `'c:\temploc'` does not exist and that a nonzero value (-70008) is returned:

```
38   libname mytemp "c:\temploc";
NOTE: Library MYTEMP does not exist.
39   %put Zero is success: &syslibrc;
Zero is success: -70008
```

The LIBNAME and FILENAME functions (see Program 8.3.2d) also have a return code, however these functions do not update the corresponding automatic macro variables. These are updated only by the LIBNAME and FILENAME statements. This includes when these functions are executed using %SYSFUNC.

You can change or reset the values of these macro variables directly, however they can only be reset to integers. Fractional values are truncated, and you will generate an error if you try to insert a value that cannot be converted to a number. Interestingly, scientific notation does not generate an error, nor does it convert to the correct value.

8.4 Even More System Options

The macro programmer should at least be aware that there are a number of less commonly used system options that affect the operation and performance of the macro language. You can list the system options that apply to the macro language by using the GROUP= option on the PROC OPTIONS statement.

Program 8.4: Displaying System Options Related to the Macro Language

```
proc options group=macro;
   run;
```

MORE INFORMATION: Some of the primary system options used with the macro language were introduced in Section 3.3. Additional system options that can be used with autocall macro libraries are discussed in Section 10.4.2.

8.4.1 Memory Control Options

Typically, macro symbol tables, and therefore the values of macro variables are stored in memory. When the memory required to store the value of a macro variable is not available, SAS will instead write the macro variable to a catalog (under Windows the catalog is named WORK.SAS0ST0). In this catalog each macro variable is a separate entry (that is, it has an entry type of MSYMTAB).

MVARSIZE

MVARSIZE specifies the maximum size that an individual macro variable can take on before it is written to disk. The default size for SAS9.4 under Windows is 64K bytes, which is also the same as the maximum size of a macro variable.

MSYMTABMAX

MSYMTABMAX specifies the maximum memory that is available for all symbol tables. When this value is exceeded the macro variables are written to disk. The default size for Windows and UNIX is about 4 megabytes (one megabyte for z/OS). To improve performance increase this limit if you have either a large number of variables or if the variables themselves are large.

By adjusting the values of these options, you can control where macro variables and symbol tables will be written. Usually, these options are not of general concern, but they can be useful if you have either large symbol tables or large macro variables and you are limited either in available memory or available disk space.

SEE ALSO: DiIorio (1999) uses the MVARSIZE option to force macro variables into a catalog where they can be removed.

8.4.2 Preventing New Macro Definitions with NOMCOMPILE

The MCOMPILE option should almost always be left on (its default value). When NOMCOMPILE is specified you will not be able to compile new macros. The only time I have found this option to be helpful was with an application that was being executed in a controlled environment and user-defined macros were highly discouraged.

SEE ALSO: Sun and Carpenter (2011) discuss the use of this option along with others when attempting to develop a controlled environment.

8.5 Even More DATA Step Functions and Statements

The DATA step has a number of ways to interface with the macro language. Often you will use the macro language to write DATA step code; however, there are a number of DATA step tools that can be used to create macro variables and to execute macro code. The CALL SYMPUTX routine (introduced in Section 6.1) and the CALL EXECUTE routine (introduced in Section 6.5) are prime examples of DATA step routines that work with macro language elements that write to symbol tables. This section describes some other DATA step functions that you, as a macro programmer, should know.

8.5.1 DOSUBL Function

In Section 6.5 the CALL EXECUTE routine is introduced and discussed. Of special interest are the timing issues associated with that routine. CALL EXECUTE gives us the ability to immediately execute macro statements from within the DATA step. However, because of the timing of events when using this function, the results can be 'different' from what you might otherwise expect (see Section 6.5.3 for more detail on timing issues).

The DOSUBL function is similar to the CALL EXECUTE routine in that it can be used to immediately submit and execute macro code from within a DATA step. However, many of the timing issues associated with the CALL EXECUTE routine are eliminated by this function. DOSUBL is not a replacement for CALL EXECUTE; rather, it is a different way of solving the problem of the execution of code from within the DATA step.

SYNTAX:

```
rc = DOSUBL(argument);
```

Program 6.5.3b was used to illustrate the timing differences between macro language elements and non-macro language elements in code submitted through CALL EXECUTE. When a macro is called through CALL EXECUTE macro code is executed immediately (for the entire macro) while non-macro code (including masked macro code) is placed in a stack for later execution. Because this dichotomy is step

independent, the behavior is very different than all other macro/non-macro executions which respect the step boundary. When using the DOSUBL function to execute a macro, the step boundaries are respected.

Program 8.5.1 repeats the example that was used for CALL EXECUTE in Program 6.5.3b, except that DOSUBL is used instead of CALL EXECUTE.

Program 8.5.1: Event Timing Associated with the DOSUBL Function

```
%macro test;
data _null_; ❸
   put 'Calling SYMPUTX';
   call symputx('x3',100);
   run;

%put Ready to compile DATA step for NEW; ❹
data new; ❼
   %put Compiling NEW; ❺
   put 'Executing NEW';
   y = &x3; ❻
   run;
title 'Data NEW'; ❽
proc print data=new;
   run;
%mend test;

data _null_; ❶
   rc= dosubl('%test'); ❷
   put rc=; ❾
   run;
```

❶ During DATA step execution the DOSUBL function calls the macro %TEST. The calling DATA step is suspended and %TEST is immediately executed.

❷ The DOSUBL function contains a call to the %TEST macro. Notice that %TEST is enclosed in single quotes.

❸ The DATA step in %TEST is immediately executed and the macro variable &X3 is defined and given the value of 100. If %TEST had been called using CALL EXECUTE, this DATA step would have been placed in a stack for execution after the calling DATA step had completed execution, and &X3 would remain undefined until then.

❹ The %PUT is executed after the DATA step completes. If a CALL EXECUTE had been used, this %PUT would have executed before the preceding DATA step.

❺ This %PUT is executed as the DATA step (❼) is being compiled.

❻ The macro variable &X3 is resolved during the compilation of the DATA step.

❼ After compilation the DATA step is executed.

❽ The title is defined and the PROC PRINT executes.

❾ The calling DATA step resumes execution and the PUT executes. RC=0 indicates that the DOSUBL executed successfully.

```
Calling SYMPUTX ❸
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.01 seconds


Ready to compile DATA step for NEW ❹
Compiling NEW ❺
Executing NEW
NOTE: The data set WORK.NEW has 1 observations and 1 variables. ❼
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
```

```
      cpu time            0.01 seconds


NOTE: Writing HTML Body file: sashtml.htm
NOTE: There were 1 observations read from the data set WORK.NEW. ❽
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.46 seconds
      cpu time            0.29 seconds


rc=0 ❾
NOTE: DATA statement used (Total process time): ❶
      real time           0.73 seconds
      cpu time            0.40 seconds
```

If you want to execute a macro from within a DATA step and the macro is not specifically designed to execute with CALL EXECUTE, consider using DOSUBL.

SEE ALSO: The documentation for DOSUBL has a nice example that shows how one of the limitations of CALL EXECUTE can be overcome by using DOSUBL. Henderson (2014) and Parker (2015) both use the DOSUBL function to generate code for PROC STREAM.

8.5.2 Deleting Macro Variables with CALL SYMDEL

When executing within the DATA step it is possible to delete macro variables from the global symbol table through the use of the CALL SYMDEL routine. Much like the %SYMDEL macro statement (see Sections 2.7 and 8.2.1), this routine only deletes macro variables from the global symbol table.

SYNTAX:

```
CALL SYMDEL(macrovariablename<,NOWARN>);
```

The use of the CALL SYMDEL routine is shown in Program 8.5.2, which creates both local and global macro variables and then attempts to delete them using CALL SYMDEL. A warning is issued when an attempt is made to delete a macro variable that does not exist. The optional second argument can be set to NOWARN, which eliminates this warning.

Program 8.5.2: Using the CALL SYMDEL Routine

```
%global City;
%let city = Los Angeles; ❶
%macro test;
%local city; ❷
%let city=Anchorage;
%let state=Alaska;
data _null_;
   put 'Delete Global &CITY';
   call symdel("city"); ❸
   put 'There is no Global &CITY to delete';
   call symdel('city'); ❹
   put '&state does not exist in the global table';
   call symdel('state','nowarn'); ❺
   run;
%* show that the local version of &CITY still exists;
%put Within TEST &=city; ❻
%mend test;
%test
%* is there a global version of &CITY?;
%put &=city; ❼
```

❶ The macro variable &CITY is established in the global symbol table.
❷ A local version of &CITY is also established.
❸ The global version of &CITY is deleted.
❹ The second attempt to delete &CITY will result in a warning, since &CITY no longer exists on the global table. The local version of &CITY is ignored.
❺ &STATE only exists on the local table, but the NOWARN option prevents the warning from being issued in the SAS Log.
❻ Show that the local version of &CITY remains unaffected.
❼ Show that the global version of &CITY has been eliminated

Program 8.5.2 (SAS Log): Showing Results of the Use of CALL SYMDEL

```
Delete Global &CITY ❸
There is no Global &CITY to delete
WARNING: Attempt to delete macro variable CITY failed. Variable not found.
❹
&state does not exist in the global table ❺
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds


Within TEST CITY=Anchorage ❻
WARNING: Apparent symbolic reference CITY not resolved. ❼
114  %* is there a global version of &CITY?;
115  %put &=city;
city
```

MORE INFORMATION: The %SYMDEL macro statement is introduced in Section 2.7.

8.5.3 Using SYMEXIST, SYMGLOBL, and SYMLOCAL

If your DATA step is going to create a macro variable, it could be important to know if that macro variable already exists in either a local or global symbol table. Each of these functions can assist with that determination.

SYNTAX:

SYMEXIST(*macrovariablename*)

SYMLOCAL(*macrovariablename*)

SYMGLOBL(*macrovariablename*)

VALUES RETURNED:

Each of these functions returns either a

1 macro variable is found

0 macro variable is not found

The SYMEXIST function only will tell you whether the macro variable exists in some scope. SYMGLOBL (note the spelling of this function) checks only the global symbol table, while SYMLOCAL checks each of the local symbol tables.

Program 8.5.3: Detecting Macro Variables and Their Scope

```
%global City state; ❶
%let city = Los Angeles;
%let state= CA;
%macro test;
%local city; ❷
%let city=Anchorage;
data _null_;
  var='state'; ❸
  if symexist("city") then do;
    if symlocal('city') then put 'macro variable city exists locally'; ❹
    if symglobl('city') then put 'macro variable city exists globally';❺
    if symglobl(var) then put 'macro variable ' var ' exists globally';❻
end;
  run;
%mend test;
%test
```

In the DATA step in %TEST a check is made to determine if a macro variable exists in some table and if it does secondary checks are used to determine if it is a local or global table. In this case &CITY is in both symbol tables. The SAS Log shows that all three IF statement expressions are true.

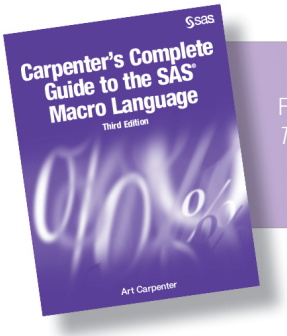
```
214  %test

macro variable city exists locally ❹
macro variable city exists globally ❺
macro variable state  exists globally ❻
```

❶ The macro variables &CITY and &STATE are added to the global symbol table.
❷ The macro variable &CITY is also added to the local table.
❸ The DATA step variable VAR is created with the value of 'state'.
❹ %SYMLOCAL shows that there is a local version of the macro variable &CITY.
❺ %SYMGLOBL shows that there is a global version of the macro variable &CITY.
❻ A variable name (VAR) that resolves to the name of a macro variable (STATE) is used to show that &STATE exists on the global table.

Remember that these are DATA step functions and as such they work with strings that take on the names of macro variables or variables that resolve to the name of a macro variable.

MORE INFORMATION: Section 8.1.5 discusses the macro versions of these functions. SYMEXIST is used in Program 9.3a to check for the existence of a macro variable before retrieving it using SYMGET.



Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

X

Y

Z

About This Book

Purpose

This book was written to provide a comprehensive look at the SAS Macro Language. It takes the reader from the most basic introduction through the most advanced topics in the macro language. Regardless of your current level of understanding of the macro language, this book contains new ways of looking at the language as a tool for improving your SAS programs.

Is This Book for You?

Written for all levels of macro language understanding, this book takes the reader from an introduction that assumes no macro language knowledge—and indeed little SAS knowledge—to advanced topics for the advanced SAS programmer. Regardless of where you are in your journey with SAS, you will find this book helpful if you are at all interested in improving your coding skills and in incorporating macro language elements in your programs.

Prerequisites

Although SAS programming strength is not a prerequisite per se, the stronger you are in the overall use of SAS, the easier it will be for you to learn the macro language. That said, even someone just starting to learn SAS can take advantage of the fundamentals of the macro language. The book is written with the expectation that readers will come to it with widely varying levels of understanding of SAS. If you are just starting with the macro language, the concepts in this book should provide you challenge for years to come.

What's New in This Edition

The third edition not only incorporates recent changes and additions to the macro language, but, more important, it greatly expands the sections on using list processing techniques, writing dynamic applications, and writing data-driven macros.

Scope of This Book

This is a comprehensive book on the SAS macro language. It covers all aspects of the macro language from basic concepts, to how the macro language thinks and interacts with the rest of SAS, and then on to the most advanced macro programming techniques.

The book itself is divided into four primary parts, the first three of which roughly translate into levels of complexity.

Part 1

Part 1 is a very basic introduction to the macro language, including explanations of why one should want to go to the trouble of learning the language. It contains basic explanations and basic syntax. This part of the book is designed for the user who is new to the macro language, and it is written to be read sequentially. It is *not* written as a syntax or reference guide.

Part 2

Part 2 covers the beginning and intermediate usage and organization of the macro language. In these chapters macro language statements and functions are introduced and demonstrated. Macro quoting is explained, and you will learn to write and use your own macro functions.

Part 3

Part 3 presents in detail the writing of dynamic applications. Various aspects of list processing are covered in detail. Throughout the examples in these chapters, you will discover numerous data-driven programming techniques.

Part 4

Part 4 includes more examples and many miscellaneous macro language topics.

About the Examples

Software Used to Develop the Book's Content

Although this book was written using the latest maintenance release, SAS 9.4 M4, virtually all of the content and examples will be applicable to all users of SAS, regardless of SAS release, their operating system, or their SAS interface (batch, SAS Display Manager, SAS Enterprise Guide, SAS Studio, or SAS University Edition).

Example Code and Data

There are nearly 400 example programs that accompany this book. These programs use either standard SAS data sets such as those in the SASHELP library, or data sets that are included with the downloaded programs. The SAS example programs are organized by chapter, and the names of the programs correspond to the section number. For example, the first program in Section 6.3.2 is Program 6.3.2a, which can be found in the downloadable Chapter 6 SAS programs with the filename Carpenter_17835TW_Program6.3.2a.sas.

You can access the example code and data for this book by linking to its author page at http://support.sas.com/publishing/authors. Select "Art Carpenter." Then look for the cover thumbnail of this book, and select "Example Code and Data" to display the SAS programs that are included in this book.

If you are unable to access the code through the website, send email to saspress@sas.com.

SAS University Edition

 This book is compatible with SAS University Edition.

If you are using SAS University Edition you can use the code and data sets provided with this book. This helpful link will get you started: http://support.sas.com/publishing/import_ue.data.html.

Exercises and Solutions

Exercises designed to test your understanding of the material discussed in this book can be found at the end of Chapters 2 through 7. Programs used in these exercises can be found among the other example programs (see "Example Code and Data" above). Answers to the exercise questions, as well as solutions to programming tasks, can be found in Appendix 1 of this book.

Additional Help

Although this book illustrates many analyses regularly performed in businesses across industries, questions specific to your aims and issues may arise. To fully support you, SAS Institute and SAS Press offer you the following resources:

- For questions about topics covered in this book, contact the author through SAS Press:
  - Send questions by email to saspress@sas.com; include the book title in your correspondence.
  - Submit feedback on the author's page at http://support.sas.com/author_feedback.
  - More information on the book is available on sasCommunity.org at http://www.sascommunity.org/wiki/Category:Carpenters_Complete_Guide_to_the_SAS_Macro_Language,_Third_Edition.
- For questions about topics in or beyond the scope of this book, post queries to the relevant SAS Support Communities at https://communities.sas.com/welcome.
- A great deal of macro-language-related content can be found on http://www.sascommunity.org/wiki/Category:Macro_Language.
- SAS Institute maintains a comprehensive website with up-to-date information. One page that is particularly useful to both the novice and the seasoned SAS user is its Knowledge Base. Search for relevant notes in the "Samples and SAS Notes" section of the Knowledge Base at http://support.sas.com/resources.
- Registered SAS users or their organizations can access SAS Customer Support at http://support.sas.com. Here you can pose specific questions to SAS Customer Support; under "Support" click "Submit a Problem." You will need to provide an email address to which replies can be sent, identify your organization, and provide a customer site number or license information. This information can be found in your SAS logs.

Keep in Touch

We look forward to hearing from you. We invite questions, comments, and concerns. If you want to contact us about a specific book, please include the book title in your correspondence.

Contact the Author through SAS Press

- By email: saspress@sas.com
- Via the Web: http://support.sas.com/author_feedback

Purchase SAS Books

For a complete list of books available through SAS, visit sas.com/store/books.

- Phone: 1-800-727-0025
- Email: sasbook@sas.com

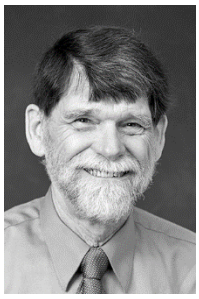
Subscribe to the SAS Learning Report

Receive up-to-date information about SAS training, certification, and publications via email by subscribing to the SAS Learning Report monthly eNewsletter. Read the archives and subscribe today at http://support.sas.com/community/newsletters/training!

Publish with SAS

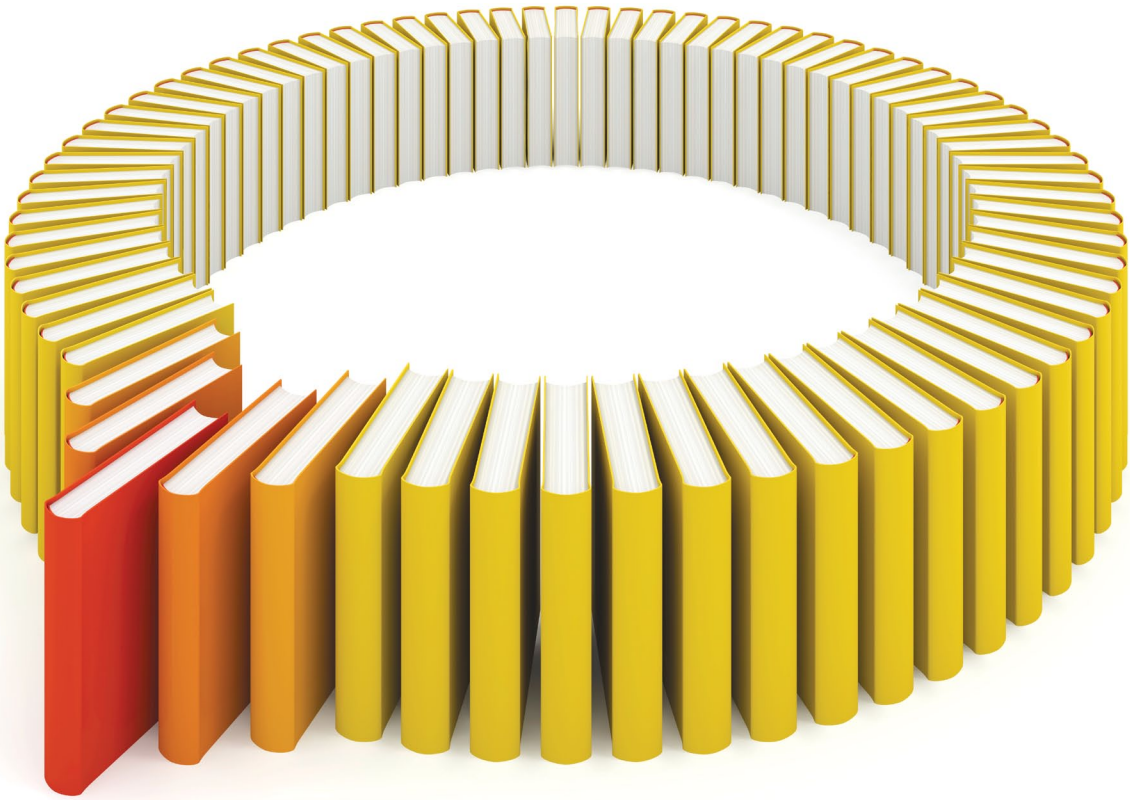
SAS is recruiting authors! Are you interested in writing a book? Visit http://support.sas.com/saspress for more information.

About the Author



Art Carpenter, an independent consultant and statistician, has been a SAS user since 1977. His impressive list of publications includes *Carpenter's Guide to Innovative SAS® Techniques*; *Carpenter's Complete Guide to the SAS® REPORT Procedure*; *Carpenter's Complete Guide to the SAS® Macro Language, Third Edition*; *Annotate: Simply the Basics*; his co-authored *Quick Results with SAS/GRAPH® Software*; and two chapters in *Reporting from the Field*. He also has served as the general editor of Art Carpenter's SAS Software Series. As an Advanced SAS Certified Professional, Art has presented more than a hundred papers, posters, and workshops at SAS Global Forum, SAS Users Group International (SUGI) conferences, and various SAS regional conferences. Art has received several best-contributed-paper awards, and he has served in a variety of leadership roles for local, regional, national, and international users groups, including conference chair and executive board member of the SAS Global Users Group.

Learn more about this author by visiting his author page at http://support.sas.com/carpenter. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.



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/store/books
for additional books and resources.

sas
THE POWER TO KNOW®