Chapter 4
SAS Macros: A Quick Start

4.1 Introduction: Why Macros?

This chapter provides readers unfamiliar with the SAS macro language a quick review of the basic elements of SAS macro programming. This is not a comprehensive guide, but rather it contains the minimum requirements to write reasonably efficient macros.

In providing this brief introduction, we adopted a learning strategy based on *What do I need to learn to be able to . . . ?* The "*. . .*" represents different scenarios we intend to cover in our data preparation journey in this book. In these scenarios, we will need to write effective macros to facilitate performing the following tasks:

1. Write a SAS code with variable names and dataset names that are themselves parameters to allow reuse of the code with other datasets and variables. This requires familiarity with replacing text in regular SAS code. Furthermore, the text may need additional manipulation before using it in the final code. For example, the name of the SAS library will need to be concatenated to the name of the dataset before being referenced in a `DATA` step.
2. Perform some calculations before replacing the text. These calculations could be performed on either a string or numerical values. We may also need to call some of the mathematical functions provided by SAS. For example, we may wish to count the number of variables in the dataset and iterate on each of them to determine some univariate statistics as part of the EDA process.
3. Allow the code to be controlled by some programming logic, such as conditional execution of some parts of it depending on the nature of the data. For example, we may decide to determine the type of a variable as either categorical or continuous and treat each type differently.

4.2 The Basics: The Macro and Its Variables

Writing SAS macros can be simplified by thinking of it as a way to replace the text of the SAS code with dynamic content. So let us start with a simple example. The following DATA step code reads the dataset DS1 into DS2:

```
DATA DS2;
 SET DS1;
RUN;
```

We can turn this simple code to create a macro that copies the contents of the dataset (call it DS_Original) to a new dataset, (DS_Copy). The way to do that is to use macro variables to express the names of the datasets in the DATA step instead of the static names DS1 and DS2. We can accomplish this as follows:

```
DATA &DS_Copy;
 SET &DS_Original;
  RUN;
```

Notice that we used the operator & to let SAS substitute the "contents" of the variables DS_Copy and DS_Original in those exact locations in the code. Then all we need to do each time we want to use this code is to *set* the value of these variables to the actual names of the datasets. This is accomplished using the keyword %LET . Note the percent sign that precedes the keyword LET. This percent sign always precedes *all* SAS macro keywords. The syntax used to define a macro variable and set its value is

```
%LET DS_Original = DS1;
%LET DS_Copy = DS2;
```

Now, we can define the two variables DS_Original and DS_Copy and give them their respective values as the names of the datasets DS1 and DS2, respectively, and we are ready to use the code that copies DS_Original into DS_Copy.

Finally, we add one more twist to this example. In order to *package* the copying code into a *SAS macro*, we enclose the code between two statements: one uses the keyword %MACRO ; the second ends the macro definition using the keyword %MEND. The syntax for the %MACRO statement includes the name of the macro and the list of parameters being passed to this macro enclosed in parentheses, as in the following example:

```
%MACRO CP_DS(DS_Original, DS_Copy);
... (the code goes here)
%MEND CP_DS;
```

The final code of the macro, then, is as follows:

```
%MACRO CP_DS(DS_Original, DS_Copy);
DATA &DS_Copy;
  SET &DS_Original;
  RUN;
%MEND CP_DS;
```

Note that the name of the macro in the %MEND statement is optional and will be ignored in most of our code.

And just as in all programming languages, we first define the function and then use it. In the case of SAS macros, we first define the macro and then use it. To use a macro, we need only to invoke its name preceded by the % and pass the appropriate values to its parameters. So, in our example, we invoke the dataset copying macro as follows:

```
%CP_DS(DS1, DS2);
```

The limitations on the names that can be used to define macro variables and macros are the same limitations that apply to general SAS programming language. Although recent versions of SAS allow longer variable names, we advise avoiding excessively long variable names. We also recommend that the user consult the most recent documentation on SAS macro language to review the limitations on naming macros and macro variables.

Within the body of the SAS macro, we can include almost any regular SAS code. However, this code will be executed only when the macro is invoked.

Variables declared within a macro are *local*, in the sense that their scope of definition is limited within the macro and cannot be reached from the open code area. Once the program execution leaves the macro, these variables will no longer be defined. Variables that are used as arguments in the macro name are local by definition. Macro variables declared outside any macro are *global*, and all macros can *see* these variables and use them. It is a good practice to limit the number of such variables and try to define all the variables within the macros to limit the chances of errors that can result from unintentionally overwriting values to the same variables. And as might be expected, a list of macro variables can be declared local or global by preceding them with the keywords %LOCAL and %GLOBAL, as follows:

```
%LOCAL Local_Var1 Local_Var2;     /* Local variables  */
%GLOBAL Global_Var1 Global_Var2;  /* Global variables */
```

We will get back to the issue of local and global macro variables later in this chapter. Please make sure that you read the section on common macro patterns before you start writing macros that call other macros.

To summarize the basic features of macros:

- Macro variables are defined using the `%LET` statement, which assigns a text to the contents of the variable. The value of the variable can be restored in the code using the `&` operator.
- Macros can be defined by enclosing the macro code within the statements `%MACRO` and `%MEND`. The name of the macro is mandatory for the `%MACRO` statement but optional for the `%MEND` statement.
- Variables within a macro can be declared global or local. By default all variables created within a macro as well as the macro arguments are local. Variables created in the open code area are global.

4.3 Doing Calculations

The SAS macro language provides a wide variety of commands to facilitate calculations. The simplest one is the function `%EVAL()`, which evaluates an *integer* expression. For example, to increase the value of a counter by 1, we can write:

```
%LET Counter = 1;
%LET Counter = %EVAL(&Counter+1);
```

Again, notice that we did not forget to use the `&` before the variable `Counter` inside the expression to let the macro compiler substitute its previous value. The result of the above calculation would be to set the value of the variable `Counter` to 2.

A useful command to print the value of any macro variable to the SAS Log is the command `%PUT`. To display the final value of the variable `Counter` of the preceding calculation, we would use the `%PUT` at the end of the code. The final code would be as follows:

```
%LET Counter = 1;
%LET Counter = %EVAL(&Counter+1);
%PUT &Counter;
```

To do floating point calculations, the function `%SYSEVALF()` is used. The following example shows how to use this function to calculate the area of a circle using its radius.

```
%LET Radius = 10;
%LET PI = 3.14159;

%LET Area = %SYSEVALF(&PI * &Radius * &Radius );
%PUT &Area;
```

Finally, SAS provides a very useful macro function: %SYSFUNC(,). This function can take two arguments. The first is any regular SAS function, and the second is the SAS format to use to present the result. The following example sets the value of the variable Date_Today to the current date in words.

```
%let Date_Today = %Sysfunc(Date(), worddate.);
%put &Date_Today;
```

The second formatting argument is optional and can be omitted if no formatting is required. There are some restrictions on the SAS functions that can be called using this function. Please refer to the SAS Macro manual or SAS online help to learn about SAS functions that cannot be called using %SYSFUNC(). The good news is that all regular mathematical functions and data conversion functions can be used with %SYSFUNC(). The importance of this function is that it extends the scope of the numerous SAS functions to the macro language.

4.4 Programming Logic

Almost all programming languages provide a set of statements for creating programs that express complex logic through three basic categories of statement.

1. Iterations through a set of statements a predefined number of times. This is similar to the DATA step DO loop in regular SAS language.
2. Statements that create loops that keep iterating until or while a certain condition is valid. This is similar to the DO–WHILE and DO–UNTIL loops in the SAS DATA step.
3. Statements that make decisions based on checking the validity of conditions. This is similar to IF–THEN–ELSE conditions used in the SAS DATA step.

All these categories of statement are supported in the SAS macro language. The syntax is almost the same as that used in the DATA step, with the exception that in the case of macros we use the % before *all* the key words.

For example, the syntax for a DO loop in the macro language is

```
%do i=1 %to 10;
... (some statements go here)
%end;
```

The DO–WHILE, DO–UNTIL, and IF–THEN–ELSE statements share the fact that they require the testing of some logical condition. For example, define two macro variables and write a simple IF–THEN statement.

```
%LET A=10;
%LET B=20;
```

```
%IF &A>&B %THEN %LET C = A;
          %ELSE %LET C = B;
```

The code defines two variables, A and B. It then compares them and sets a variable C to the name of the larger one. Notice that we used the & operator to restore the values of the variables A and B before comparing their values. If we did not do so, the condition A > B would compare the two "texts" "A" and "B" and would result in an error.

A logical condition in SAS macros must compare two numerical values. In fact, it compares two integer values. If one of the values is real (i.e., contains numbers to the right of the decimal point), we must use the function %SYSEVALF() to enclose the condition. For example, let us assign noninteger values to A and B and rewrite the code.

```
%LET A=5.3;
%LET B=12.5;

%IF %SYSEVALF(&A>&B) %THEN %LET C=A;
                     %ELSE %LET C=B;
```

The syntax for the %DO–%WHILE and %DO–%UNTIL loops is straightforward. The following examples show how to use these two statements.

```
%macro TenIterations;
/* This macro iterates 10 times and writes
   the iteration number to the SAS Log. */
%let Iter=0;                /* intialize i to zero */
%do %while (&Iter<10);      /* loop while i<10     */
 %let Iter=%Eval(&iter+1);  /* increment i         */
 %put &iter;                /* write it to the log */
%end;
%mend;
```

This example performs the same function but uses the %DO–%UNTIL structure.

```
%macro TenIterations;
  %let Iter=0;
  %do %Until (&Iter =10 );
    %let Iter = %eval(&Iter +1);
    %put &Iter;
%end;
%mend;
```

Note the use of a different logical condition to stop the iterations in the two macros.

Before we move on to the next topic, we emphasize a very important fact about the preceding programming logic structures in SAS macros:

> All the programming logic statements, such as %IF–%THEN–%ELSE, %DO–%UNTIL, and %DO–%WHILE, are valid *only within* the body of a macro. We cannot write these statements in open program code.

4.5 Working with Strings

The SAS macro language provides a long list of functions to manipulate string values. These functions allow for removing quotation marks, trimming strings, searching for sequences of characters in them, and so on. Of this long list, the most used functions are %LENGTH() and %SCAN(). The following is a brief description of these two functions.

The %LENGTH() function, as expected, evaluates the length of a string. For example, consider the following code:

```
%let X=this is a string.;
%let L = %length(&X);
%put Length of X is: &L characters;
```

The code would result in the variable L storing the value 17. (Remember that the period at the end of the string stored in the variable X and the spaces count as characters).

The function %SCAN() is used to extract the next word from a string starting from a certain position. The delimiters used to separate words can be specified as one of the arguments. The syntax of %SCAN is as follows:

```
%SCAN (String to search, word number, list of delimiters)
```

The first argument is the string to examine and it is mandatory. The word number is optional (1 by default) and determines the word number to extract. For example, using three extracts the third word in the expression. The list of delimiters is also optional; the default delimiters are the space, the comma, and the semicolon. The following code is a macro that decomposes a string into its words and prints them in the log.

```
%macro Decompose(List);
%let i=1;
%let condition = 0;

%do %until (&condition =1);
  %let Word=%scan(&List,&i);
  %if &Word =  %then %let condition =1;
               %else %do;

  /*** This part prints the word to the log */
       %put &Word;
  /***/

  %let i = %Eval(&i+1);
               %end;  /* end of the do */
  %end; /* end of the until loop */
%mend;

%let MyList = This is a list of several words;
%Decompose (&MyList);
```

Another common string macro function, which is usually useful when comparing two strings, is the `%UPCASE()` function. As the name implies, it converts all the characters of the input string to uppercase. This makes it easy to compare two strings while ignoring the case. The function `%LOWCASE()` converts all the characters to lowercase. Neither of these functions are real functions; they are called "macro calls"; they are built-in macros provided by SAS. However, we do not need to be concerned with this fine distinction.

4.6 Macros That Call Other Macros

When macro programs become more complex, it is easier to divide the program into several macros and have a main macro that calls all the bits and pieces. For example, the following code shows the macro `Main()`, which calls two other macros.

```
%macro Main();

 ... /* some statements */
 %local x1 x2 x3;
 %let x1=7; %let x2=89; %let x3=92;
 %Func1(&x1, &x2);
 ... /* more statements */
 %Func2(&x3, &x2);
 ... /* more statements */
%mend;
```

The only precaution is that the macros `Func1()` and `Func2()` shown in the code have to be defined *before* using them in the macro `Main()`. This can be achieved by simply putting the definition of `Func1()` and `Func2()` *above* the code of `Main()` in the same file. This way, during execution, the SAS compiler will compile the macros `Func1()` and `Func2()` before attempting to compile and execute `Main()`.

When the called modules, such as `Func1()` and `Func2()`, are themselves calling other modules, or when they are large macros with lengthy code, then it is more convenient to store the macros in different files. In this case, we can *include* a particular macro before it is used by another macro using the statement `%INCLUDE`, as follows:

```
%INCLUDE 'C:\Macros\Macro_Func1.sas';
```

This statement instructs SAS to include the contents of the file `C:\Macros\Macros_Funcl.SAS`. The filename could be enclosed in either single quotes (') or double quotes ("). When double quotes are used, we can substitute in them the values of some macro variables. For example, we could define the directory name as a variable and then add the filename to it, as follows:

```
%let Dir=C:\Macros\;
%include "&Dir.Macro_Func1.sas";
```

Note the dot at the end of the macro variable `Dir`. It tells the compiler to resolve the value of `Dir` before appending `Macro_Func1.SAS` to it.

In this way, we can write general programs that would work on, say, both UNIX and Windows file structures using simple logic, as follows:

```
%let OS=WIN; /* specify the Operating system */
/* if in UNIX use: %let OS=UNIX; */

%let dirWin=C:\MyMacros\;
%let dirUNIX=/myroot/sasfiles/mymacros/;

%if &OS=WIN  %then %let Dir=&dirWin;
%if &OS=UNIX %then %let Dir=&dirUNIX;

%include "&Dir.macro_A.sas";
```

We recommend that each macro be written in its own file so that when we need to call this file in a program or a macro, we can simply call it using the `%INCLUDE` statement. For example, the `Main()` macro mentioned would call its submodules `Func1()` and `Func2()`, as follows:

```
%include "C:\Macros\Func1.sas";
%include "C:\Macros\Func1.sas";

%macro Main();
 ...
 %Func1(&x1, &x2);
 ...
 %Func2(&x3,&x2);
 ...
%mend;
```

Finally, the `%INCLUDE` statement can be used to combine any usual SAS code that is spread over several physical files. Many of the macros presented in this book use other macros. The dependency between these macros is provided in Section A.2.

4.7 Common Macro Patterns and Caveats

In writing SAS macros, certain common tasks always arise. This section provides the details of some of these patterns.

4.7.1 Generating a List of Macro Variables

This task arises, for example, when we decompose a list of string tokens, such as the list of variables to be included in a model, and we want to keep these variables in a list of macro variables. The required variables should be easy to increment through. For example, we would like to generate a list of variables called `X1, X2,` ... up to `XN,` where `N` is their number, an integer, and is already stored in a macro variable.

This is usually achieved using the following pattern:

```
%do i=1 %to &N;
  %let X&i = { here goes the definition of variable x(i) }
   ... {other statements}
%end;
```

The loop shows that we iterated through the *N* values and generated a list of variables using the `%LET x&i` statement. The SAS macro compiler resolves the values of the variables generated by this statement first, such that the pattern is equivalent to writing *N* `%LET` statements as

```
  %let X1 = ...
  %let X2 = ...
   ... { up to XN}
```

In other parts of the macro, to access the values stored in these variables, we use a `%DO` loop to iterate through them. The values stored in each of these variables are accessed through the *double ampersand*, `&&`, convention, as follows:

```
%do j=1 %to &N;
   /* store the value of Var_j in a new variable y_j */
  %let y_&j = &&Var_&j;
   ... (other statements)
%end;
```

In the preceding code, the second part of the `%LET` statement (`&&Var_&j`) reads as follows: resolve the value of `j` first, then resolve the value of the resulting variable. For example, during the third iteration of the `%DO` loop (`j=3`), the part `&&Var_&j` will first resolve `j` to the value of 3 then resolves the value of `Var_3`. This logic is illustrated in Figure 4.1 when the value of the variable `Var_3` is the string `My Text`.

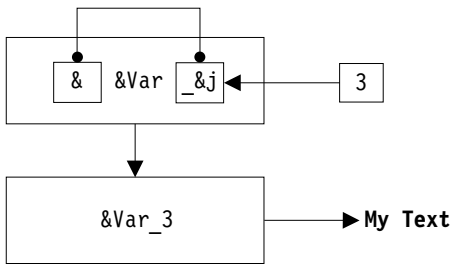

Figure 4.1 Resolution of macro variables with double coding.

4.7.2 Double Coding

Double coding means having a macro variable that holds the name of another macro variable. For example, we wish to have two macro variables that hold the names of the categories of a variable, such as the values `Female` and `Male` of the variable `Gender`. We can do that as follows:

```
%let V1=Female;
%let V2=Male;
```

Now, let's have another macro variable called `X` that could have either the values `V1` or `V2`.

```
... (some statements)
%let X=V1;
... (some statements)
%let X=V2;
... (some statements)
```

At a later stage in the macro, we need to resolve the value of the macro variable included in `X` directly, that is the value of `Gender` (`Male` or `Female`). We do that as follows:

```
%let Xg= &&&X;
```

The three ampersands are interpreted over two steps, as follows:

- First resolve the value of `&X` to give either `V1` or `V2`.
- Then resolve the resulting value (`V1` or `V2`;) as a macro variable. This should give the required value (`Female` or `Male`).

The double coding pattern is useful in cases that involve automatically generated macro variables.

4.7.3 Using Local Variables

Some of the macros presented in this book involve calling other macros. In these cases, we have to be careful to declare *all* the variables in each macro as *local*. Failing to do this could lead to errors that are difficult to debug.

For example, consider the macro `Parent` that calls another macro, `Child`, as follows:

```
%macro Child;
 %do i=1 %to 2;
   %put --- Now in Child --Message &i;
 %end;
 %mend Child;

%macro Parent;
 %do i=1 %to 5;
```

```
   %Child;
 %end;
%mend Parent;
```

The macro `Child` appears to be an innocent macro that prints only two messages to the SAS Log.

If we invoke the macro `Parent`, we would expect a total of 10 messages to be printed to the SAS Log (5 times for the loop in `Parent` X, twice in each call of `Child`). However, we will discover that attempting to run the macro `Parent` will result in an endless loop of messages on the SAS Log. (Try it yourself and be prepared to press the Stop button!)

The reason for such a problem is that the macro `Child` *modifies* the value of the counter variable `i`, which is used in a loop in the macro `Parent`. Every time `Child` is called, `i` loops from 1 to 2. But the loop in `Parent` will not stop until `i` reaches a value of 5, which it will never do.

To remedy this situation, we declare `i` a local variable in *both* macros (at least in `Child`, but we opt for both to be on the safe side). Therefore, a good code that will behave as expected would be as follows:

```
%macro Child;
%local i;
 %do i=1 %to 2;
   %put --- Now in Child --Message &i;
 %end;
 %mend Child;
\midskip
%macro Parent;
%local i;
 %do i=1 %to 5;
   %Child;
 %end;
%mend Parent;
```

Note that if `i` were declared as local only in `Parent`, it would still run properly; but it is actually a sloppy programming practice because we allow a submodule to create global variables that could have an impact later on some other macro.

This behavior is especially critical for variables used as loop counters. Some symbols are popular for that function such as $i, j, k, \ldots$, and so on. Therefore, we recommend that while you are writing a macro, declare *all* the variables as local the minute they are created. In this way, you minimize the chances that any of them gets forgotten. Therefore, I actually consider it good programming practice to use several `%LOCAL` statements in a macro, although some die-hard idealists insist on putting all the `%LOCAL` declarations at the begining of the macro.

4.7.4 From a DATA Step to Macro Variables

We should not think of macro variables and macros as clever tools to control the standard SAS programs, such as `DATA` steps and the different `PROC`s. We regularly need to feed back

values read from datasets into macro variables. The way to do this is to use the system call CALL SYMPUT. (Note the CALL part of it.) In these cases, we are not really interested in processing the data in the DATA step by generating a new dataset; we are merely interested in reading the values in sequence from the dataset until we reach the required condition to set the value of the macro variable.

For example, assume that we want to find the observation number of the last observation that has a missing value in a variable called X in a dataset called DUMP. Furthermore, we need to store this value in a macro variable we will call LastMiss. We do this using the NULL dataset of SAS, which has the fixed name _NULL_, as follows:

```
DATA _NULL_;
 SET DUMP;
 IF X=. THEN CALL SYMPUT('LastMiss',_N_);
run;
```

In the DATA step, we used the system call SYMPUT, which stores the value of the SAS variable _N_, that counts the observations read so far to the dataset _NULL_. This call will be executed only if the value of variable X is missing. If variable X does not have missing values, then the macro variable LastMiss will not be created. In case the dataset DUMP has several observations with X missing, the variable LastMiss will store the observation number of the *last* one, which is what we want to do.

This pattern appears very often to read values from a dataset into macro variables. Often we use it in situations where we know that the IF condition will occur *only once* in the entire dataset or when the dataset has only one observation.

4.8 Where to Go From Here

With this introduction to SAS macro language, we believe that you can write relatively complex macros that can turn simple code into powerful analysis tools. To explore more features of SAS macro programming, consult the *SAS Macro Programming Language Manual* or one of the many books available on the subject. For suggestions on good books dedicated to writing SAS macros, please refer to the official SAS website for guidance (*www.sas.com*).

This Page Intentionally Left Blank