

Methods for Repeated Macro Calls

Nicholas Nerren, PPD, Wilmington, NC

ABSTRACT

The SAS[®] Macro Facility is useful for defining SAS statements that need to be executed repeatedly, but the best method of implementing repeated macro calls depends on the circumstances. This paper presents several methods for repeatedly calling a macro and gives the advantages and disadvantages of each. Methods covered include CALL EXECUTE and the PARMBUFF option, as well as techniques for calling macros when the parameters or number of macro calls are driven by data.

INTRODUCTION

When writing a SAS program, it is not uncommon to find yourself with a task that must be performed several times. The SAS Macro Facility is a useful tool for repeated execution, but even after you have decided to create a macro there are multiple options for how to call it. Like most programming challenges, there is no one “correct” solution. In fact, what might be an elegant solution in one situation is cumbersome in another. To be prepared for all circumstances, the prudent SAS programmer must be familiar with a variety of ways to repeatedly call code and be able to decide which best fits the task at hand.

This paper covers five methods for repeatedly calling code and discusses some advantages and disadvantages for each. By becoming familiar with these techniques, you will learn what questions to ask about your programming needs and how to select a method that fits your situation.

METHOD ONE: PLAIN CODE

When faced with a simple task that needs to be repeated only a few times, the most straight-forward approach might be to write plain code without macros. For example, if you needed to print three data sets, you might write one PRINT procedure for every data set:

```
proc print data = one;
run;
proc print data = two;
run;
proc print data = three;
run;
```

ADVANTAGES AND DISADVANTAGES

Although this method may seem rudimentary, it can be an excellent choice for simple operations. By not using the macro facility, the code is more accessible and options such as MPRINT are not required to help read the log to resolve issues. If the task was something more complex than a simple data set print, however, repeating the code once for every data set to be processed could quickly add up to a lot of code, making the program lengthy and harder to read. If the program is designed to be updated as new data sets need to be printed, new code must be written to handle them; this manual update takes time and runs the risk of introducing new errors into a valid program.

METHOD TWO: SIMPLE MACRO CALLS

Suppose instead of simply printing data sets, we now need to manipulate the data with a DATA step prior to printing. If we updated the program from method one, we would have something like this:

```
data one;
  set mylib.one;
  patient = inv||pat;
  factor = 0.85*base;
run;
proc print data = one;
run;

data two;
```

```

    set mylib.two;
    patient = inv||pat;
    factor = 0.85*base;
run;
proc print data = two;
run;

data three;
    set mylib.three;
    patient = inv||pat;
    factor = 0.85*base;
run;
proc print data = three;
run;

```

As the data step grows or other procedures are added prior to printing, our code becomes cumbersome. It is likely that a typo could result in one data set not receiving the same manipulation as the others, since the code is typed out once for every data set. To shorten the code and ensure that the same operations are performed to each data set processed, we might choose to convert our program into a macro:

```

%macro process(dataset);
    data &dataset;
        set mylib.&dataset;
        patient = inv||pat;
        factor = 0.85*base;
    run;
    proc print data = &dataset;
    run;
%mend process;

%process(one);
%process(two);
%process(three);

```

The process macro takes the data set named in its sole parameter, processes it, and prints it. Each data set that we need to treat in this manner is passed to the macro in a separate macro call.

ADVANTAGES AND DISADVANTAGES

By converting our program to a macro, we have reduced the size of the code and no longer need to retype the data step and proc print for every data set to be processed. Any updates to the data step can now be made in one location instead of three. If another data set needs to be processed, it is a simple matter of adding one line of code to call the macro an additional time.

If another programmer needs to update this code and is not familiar with the macro facility, he might find this a more difficult task. For this simple macro that uses one parameter, it is perhaps reasonable that a programmer with no macro training could derive the correct syntax for adding a macro call or updating the data step. However, as the macro grows in complexity and size, it requires more sophistication to maintain the program without error. In addition, we must now use options such as MPRINT and SYMBOLGEN to help read the log.

Suppose we know in advance that we will need to add additional macro calls as more data sets become available. Although adding additional macro calls is still a relatively simple matter at this point, it still requires manual intervention, which both takes time and puts the program at risk for error. It would be desirable to have a program that can determine what data sets are available and call the macro once for each, regardless of how many data sets exist.

METHOD THREE: THE EXECUTE ROUTINE

If we wish to enhance our program to remove the need to manually add macro calls, one possibility is to use the EXECUTE routine. In this example, we determine what data sets exist in a library, perform the required manipulations on each, and finally print the results.

```

%macro process(dataset);
    data &dataset;
        set mylib.&dataset;
        patient = inv||pat;
        factor = 0.85*base;
    run;

```

```

proc print data = &dataset;
run;
%mend process;

proc contents data mylib._all_ out=content(keep=libname memname) noprint;
run;
data _null_;
set content;
by libname memname;
if first.memname = 1 then call execute('%process('||memname||')');
run;

```

In this example, the macro program is unchanged from before, but the number and content of invocations of the process macro are dynamically driven. The CONTENTS procedure compiles a list of all data sets and variables in the library defined as mylib and stores in it a data set called content. The variables libname and memname store the name of the library and the name of the data set, respectively. The data step cycles through this list, building a macro call every time another value of memname (and thus another data set in mylib) is encountered. Since we use the data step to build macro calls and not modify the data, we can use the _null_ keyword to cycle through the data without creating an output data set.

After the data step has finished, the macro calls are executed, creating the new variables patient and factor, then printing the resulting data for each data set in the mylib library.

ADVANTAGES AND DISADVANTAGES

The main benefit that this method has over the previous method is that the programmer is no longer burdened with updating the code manually every time new data sets become available. This both saves time and removes the possibility that the programmer will overlook a data set. The tradeoff is that the code is less accesible, especially for programmers unfamiliar with the EXECUTE routine.

When using this method, it is important to keep in mind the differences between a direct macro call and those built with the EXECUTE routine. In this method, the program creates the code for all macro calls and does not execute them until after the data set terminates. This has no effective impact on the macro in this example, but could lead to unintentional problems with more complex macros, especially those that rely on the value of macro variables defined within the macro to conditionally execute code. Care must be taken that the macro used will not be negatively impacted by these execution timing differences.

METHOD FOUR: CALLING A MACRO WITH SAS FILE I/O FUNCTIONS

If the macro to be repeatedly called is not suitable for the EXECUTE routine, one alternative is to use SAS File I/O functions to call the macro:

```

%macro process(dataset);
data &dataset;
set mylib.&dataset;
patient = inv||pat;
factor = 0.85*base;
run;
proc print data = &dataset;
run;
%mend process;

%macro runit(dataset,var);
%let dsid = %sysfunc(open(&dataset));
%do i = 1 %to %sysfunc(attrn(&dsid,nobs));
%let thisobs = %sysfunc(fetchobs(&dsid,&i));
%let thisvar = %sysfunc(getvarc(&dsid,%sysfunc(varnum(&dsid,&var))));
%process(&thisvar);
%end;
%let rc = %sysfunc(close(&dsid));
%mend runit;

proc contents data mylib._all_ out=content(keep=libname memname) noprint;
run;
proc sort data=content nodup;
by libname memname;
run;

```

```
%runit(content,memname);
```

In this example, a helper macro, `runit`, is defined to call the main process macro. Because the I/O functions are not used in a data step, the `%sysfunc` function is used to cause the functions to execute. The program begins by compiling a list of the data sets found in the `mylib` library. Next, the `SORT` procedure removes duplicate entries -- a step that could have been used in the previous method instead of using `first.memname`. Then the `runit` macro is called, passing the name of the file containing the list of data sets (`content`) and the name of the variable holding the data set names (`memname`). The `runit` macro first opens the data set passed to it and reads in the number of observations. The `%do` loop cycles through the observations of the data set, taking the value of `memname` and passing it to the process macro. After all observations have been processed, the macro closes the content data set. The process macro call could be substituted with the actual body of the process macro if desired, reducing the code to one macro.

ADVANTAGES AND DISADVANTAGES

This purpose behind this method is similar to that of method four: we are dynamically constructing a list of data sets, then calling a macro that manipulates and prints each. The main difference is that in this method each macro call is executed during the `%do` loop. By avoiding the `EXECUTE` routine, the process macro can use the values of macro variables defined within the `EXECUTE` argument.

The disadvantage with the method is that, once again, the code is more complex. Programmers that are familiar with the `EXECUTE` routine may not be as comfortable with SAS File I/O functions, which can be hard to read if one does not understand the syntax.

METHOD FIVE: MACRO PARAMETER BUFFER (PARMBUFF)

One alternative to using SAS File I/O functions is to pass the list of data sets to process as parameters to a helper macro, which then calls the process macro for each. Since the number of data sets is unknown until run time, the `parmbuff` option is used to hold the macro parameters and cycle through them.

```
%macro process(dataset);
  data &dataset;
    set mylib.&dataset;
    patient = inv||pat;
    factor = 0.85*base;
  run;
  proc print data = &dataset;
  run;
%mend process;

%macro runit() / parmbuff;
%let count = 1;
%let dataset = %scan(&syspbuff,&count,%str(,));
%do %while (&dataset ^= );
  %process(&dataset);
  %let count = %eval(&count+1);
  %let dataset = %scan(&syspbuff,&count,%str(,));
%end;
%mend runit;

proc contents data mylib._all_ out=content(keep=libname memname) noprint;
run;
proc sql noprint;
  select distinct(memname) into :list separated by ','
  from content;
quit;

%runit(&list);
```

As with the previous method, we begin by constructing a list of data sets in the `mylib` library. We now compile them into a comma-delimited list and store them in a single macro variable (`list`) which is passed to the `runit` macro. The `parmbuff` option assigns the parameters used in the invocation of `runit` (including the parentheses) to a macro variable called `syspbuff`. Inside the `runit` macro, the macro variable `count` is a counter used to iterate through the data sets in `syspbuff`; `dataset` contains the name of the current data set we are processing. The `scan` function reads in the first data set and the `do while` loop begins. This loop calls the process macro with the current data set, then increments `count` and reads in the name of the next data set. When a blank is read in, we exit the `do while` loop and the program ends. As with method four, the body of the process macro could be substituted for the process macro call, resulting in only one macro in the program.

ADVANTAGES AND DISADVANTAGES

This method removes the I/O functions from methods four and trades them for macro commands. As with method four, we retain the ability to dynamically determine the macro calls and avoid the timing issues of the EXECUTE routine. We do so at the expense of having a program that takes some deciphering to tell what it does. For programmers who are used to macros but do not use I/O functions regularly, this code may be more accessible, but still takes effort to read.

CONCLUSION

This paper has presented five methods for repeatedly executing code. As with most programming decisions, there is no “right way” and “wrong way” to program, but some methods are more applicable in certain situations. Before deciding which method to use, there are several important factors to consider:

- How many times will the code be executed?
- Will the number of executions change? If so, how often?
- How often will the code need to be updated?
- How much processing needs to be performed per iteration? (How long will the code be?)
- Will other programmers need to read and modify the code? If so, what is a reasonable assumption of their familiarity with the elements in the methods considered?

Asking these questions before you start programming will help you to choose the best strategy for your task and save time down the road.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Nicholas Nerren
PPD, Inc.
929 North Front Street
Wilmington, NC 28401-3331
Work Phone: 910-558-2974
E-mail: nicholas.nerren@wilm.ppdi.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.