Type Less, Do More: Have SAS® do the typing for you

Jeanina Worden, PPD, Austin TX

ABSTRACT

Typing less when you're a SAS® programmer seems counterintuitive; however when repetitive tasks leave you with the realization only five words differentiate the last twenty lines of code, the concept becomes clearer. There are numerous ways to accomplish these tasks, such as "hardcoding" and copy-and-paste, however they carry with them increased risk in terms of additional time required for updates, and the lack of assurance all constraints are accounted for. Therefore the most common "go to" solution is the macro; however that too can quickly result in a hand cramping amount of code. This paper shows how CALL EXECUTE can instead be used to dynamically code repetitive tasks, populating the required constraints from the actual metadata ensuring all available constraints are accounted for and reducing the need for updates if the database changes, doing it all with less coding. For simplicity purposes PROC PRINT is used in the examples however the code can be changed to perform any function where the repeating code differs by a single dataset or variable name.

INTRODUCTION

Nothing causes the proverbial hand cramp faster for a programmer than repeating the same code over and over again. There are several situations where repeating the same code, with minor changes, is required; comparing all datasets within two libraries, creating individual XPT files per dataset, or even doing multiple transposes. Sure you can keystroke your way through that long list of variables when transposing an extra wide dataset, "hardcoding" the necessary code, in its entirety, for each constraint. Or there's the ever popular use of "proc copy-and-paste" to produce a mind boggling stack of code. And we can't forget the option of macro-tizing the code and using the constraint within the call of the macro, then repeating the call for each instance. But what do you do when the database changes? How do you ensure nothing was missing? Any and all of these options work as solutions however they do present similar challenges; difficult maintenance and the risk of omission. This paper shows examples that makes increasing use of metadata and SAS® tools to dynamically code repetitive tasks, populating the required constraints from the actual metadata ensuring all available constraints are accounted for and reducing the need for updates if the database changes, doing it all with less coding. For simplicity purposes the same example, PROC PRINT, is used for each example however the code can be changed to perform any function where the repeating code differs by a single dataset or variable name.

HOW TO PRODUCE HAND CRAMPS

We've all had them; those requests to write a program that involve the same programmatic task to be performed numerous times; comparing two libraries, creating individual XPT files for each dataset, creating ISO dates across all datasets for all date variables, transposing vital sign tests. This paper uses the example of printing each dataset in a library using PROC PRINT and can easily be done by coding each task individually as follows:

```
proc print data=SASHELP.RETAIL;
run;

proc print data=SASHELP.SHOES;
run;

proc print data=SASHELP.COMPANY;
run;
```

"Hardcoding" each step individually will get the job done, but tends to be keystroke laden. "Proc Copy-and-paste" is an alternative as it tends to save some typing by merely copying the first block of code and modifying the data for each repeating block. However both options bring with them similar risks due to programs often being developed prior to database reaching a truly stable, locked form, so the likelihood of the code having to be changed or updated prior to final production is usually high. If a new dataset is added to the library the program must be updated to include the code to print each new addition. In addition to the maintenance to keep the code current, the more difficult challenge is the risk of omission. Leaving a needed constraint or instance out does not produce an error but instead relies heavily on validation processes to ensure correctness. And while the copy-and-paste method saves some typing, it adds the risk of missing, or not fully completing, a change that was needed between the blocks. Additional time must be used to address these challenges which can impact a project's overall budget.

So how can you program these blocks to ensure accuracy of the database structure even after a change takes place? Developing a macro that processes the code in a dynamic way using the metadata tables available within the background of SAS® is a good place to start. There are two specifically that can help; DICTIONARY.COLUMNS and DICTIONARY.TABLES. Either table can be accessed using PROC SQL and provides information about the current session of SAS®. COLUMNS is similar to the contents generated by PROC CONTENTS and contains all available library names (libnames), dataset names (memnames), variables (name), labels and variable characteristics within one table. The TABLES dataset contains some of the same variables such as library names and dataset names, but expands to more information on the dataset level like labels, creation/modification dates, and record totals. One thing to note is that these tables are a reflection of the current session, so using these to access a permanent library is preferred. If the WORK library is used the datasets needed must always be created in that library prior to running this code. Also, if you use the COLUMNS table to capture only the dataset name, you must use the DISTINCT option on the SELECT line as this table contains records for each variable.

So we start by creating a new dataset that contains the constraint items we will need, in this case we want the datasets contained in the library SASHELP:

```
*** Create dataset containing dataset names;
proc sql;
       create table dsnames as
       select memname
       from dictionary.tables
       where libname = 'SASHELP';
quit;
```

Of course this code can be further manipulated and expanded to meet the need, for example additional code could be added to the WHERE statement if further subsetting was required. The key point is that the resulting dataset contains the varying constraints that will be used in the following part of the program. And that each time this code is ran all dataset names will be captured so if changes occur, datasets are added or deleted, no future coding would be necessary.

PULLING OUT WHAT'S NEEDED

Now that we have a source that we can pull the changing constraints from, what do we do? As stated previously, developing a macro to loop through the constraints to populate one instance of the code is an option. However, by putting the constraint as part of the call, the macro option still carries with it the same issues of maintenance and risk of omission. So instead, we can create a macro variable that contains the dataset name but changes as the names change. To do this we can use CALL SYMPUT to create a numbered macro variable:

```
*** Create a macro variable called DSN<#> and one for the max count of datasets;
data _null_;
       set dsnames end=last;
       call symput('DSN'||left(_n_),trim(MEMNAME));
       if last then call symput('count',_n_);
run;
```

First we set the new dataset and identify the end of the file with LAST. Then we use CALL SYMPUT to create two macro variables; DSN and COUNT. By concatenating the _N_ to the macro variable DSN a numbered version of the macro variable (DSN1, DSN2…) is created that corresponds to record number of the dataset name it represents. If this was not used a variable called DSN would be created but would only hold the final dataset name as it would be replaced until the end of the file was reached. For this reason the IF statement in the COUNT assignment could be left out as the COUNT variable would only contain the final record number, hence the max number of records in the file. However the IF statements enables the assignment of COUNT by directly going to the record identified as LAST.

PUTTING IT TO USE

Now we create one instance of the code that the varying names will be used in:

```
*** Create macro to loop through datasets;
%macro MAKE_DSN;

%do i = 1 %to &count;
       ****Proc contents datasets *;
       proc print data=SAShelp..&&dsn&i;
       run;
%end;
```

```
%mend MAKE_DSN;
%MAKE_DSN;
```

The macro is called one time and the variations are applied from within the macro itself by creating a DO loop to loop through the list until the maximum number is reached, as assigned in COUNT. The item to note in this section is the use of the double ampersands. The first time the macro variable is reached, it resolves to the macro variable name based on the current 'I' value, for instance the first iteration would be &DSN1. It then resolves the macro variable itself and applies the dataset name it corresponds to.

THE WAY OF THE MACRO, PART DUEX (A LITTLE LESS CODING)

CREATING A LIST

A second option, also using a macro, provides a bit less coding by first changing how the assignment of the macro variable, DSN, is done and creating a single list within the macro variable of the dataset names needed. We can still take advantage of the DICTIONARY.TABLES but instead of creating a table we can create a single macro variable, like this:

```
*** Create a list containing dataset names;
proc sql;
       select distinct 'SAShelp.'||trim(memname) into: DSN separated by ' '
       from dictionary.tables
       where libname = 'SASHELP';
quit;
```

Using INTO:, within the SELECT statement identifies the macro variable name after the semicolon, and specifying the delimiting character as the separate, in this case a space, results in a list of all dataset names within the single macro variable.

PUTTING IT TO USE

The entire list can then be fed into the macro:

```
%macro get_list;
       %do i = 1 %to %sysfunc(countw(&dsn,%str()));
              proc print
                     data=%scan(&dsn,&i,%str());
              run;
       %end;
%mend;
%get_list;
```

This example takes advantage of the version 9 function, COUNTW. The syntax for this function is COUNTW(<string>,<chars>,<modifier>); where the STRING is the list of words that are to be counted, and in this example is the DSN macro variable. The optional CHARS, and MODIFIER, work together to identify the delimiting characters that allow SAS® to recognize where a word starts, counting only the words between the delimiters. Modifiers can also be used to identify all characters not in the list as the delimiters. For this example, since we used a space when creating the DSN macro variable, we can simply leave this blank by using the %STR(). The SAS® documentation for this function can be referenced for further explanation and for a list of modifiers that can be used. Once we have the number of datasets in the list we can proceed to the varying code. We now use the SCAN function to go through the list to set the dataset name to the one located at the nth position according to 'I'. The syntax for SCAN is SCAN(<*string*>, *count*<,*charlist* <,*modifiers*>>); where STRING is the list of words, again using the DSN macro variable. In the above code, 'I' represents the count or position the word is located within the list, and we again use %STR() to identify MODIFIER since we used a blank. More detailed explanations can again be obtained from the SAS® documentation for SCAN.

SAVING THE BEST FOR LAST (THE LEAST CODE OF ALL)

While both of these macros are viable options to address the risks when repeating code, the amount of coding is really not reduced with either option. Also, some would balk at the use of macros at all. Due to their unease of using macros, or their feelings on the "overuse" of macros, some programmers would prefer a "no macros" solution. There are also the groups that just plain prefer the DATA step over diving into PROC SQL. Well the follow example can be used to accommodate these groups, and results in an option with the least amount of typing of all, one DATA _NULL_ utilizing CALL EXECUTE. The syntax for CALL EXECUTE is:

```
CALL EXECUTE('/"argument'/");
```

Where the argument is a string enclosed in single or double quotes that resolves to the code or a macro invocation that is used to build the block. It can also contain the name of a DATA step variable which is not enclosed in quotes. As with many of the other SAS routines, the quote you use matter. If ARGUMENT is contained within double quotes it is sent to the input stack during compile time and the data step pauses while the argument resolves. If ARGUMENT is contained within single quotes, such as when it resolves to SAS® statements, it is executed at the end of the DATA step.

FIRST THINGS FIRST
First it is a good idea to start with a "trial run". Create one instance of the repeating code and submit to ensure you get the result you want. For our example we would use:

```
proc print data=SASHELP.RETAIL;
run;
```

So the only thing that would change would be the dataset name, "RETAIL".

BUILDING THE _NULL_
We can now start building the DATA _NULL_ step, however since we are not using SQL we need another way of accessing the database structure dynamically. Instead of using the DICTIONARY.TABLES table to create our list of dataset names we look to the SASHELP library itself and the table VTABLE (if you need variables you can use VCOLUMN). Like the DICTIONARY tables this dataset contains information of the current SAS® session on a dataset, or memname, level. We first subset the table to only include the library we want to pull the dataset names from. Like this:

```
data _null_;
      set SAShelp.vtable (where=(libname="SASHELP"));
```

Then we create the repeating PROC PRINT code within the CALL EXECUTE statements (split for ease of review) to write the code dynamically:

```
            call execute('proc print');
            call execute ('data=SASHELP.'||trim(memname)||';');
            call execute ('run;');
```

And finally, don't forget the RUN statement. The RUN within the final line of the CALL EXECUTE is for the PROC PRINT, not the DATA _NULL_.

```
run;
```

In this example, as the dataset is set, the statements are built, and rebuilt, using the datasets as they are fed in from VTABLE. As each record is executed, CALL EXECUTE "writes" the code:

```
Proc print
data = SASHELP.<memname>;
run;
```

There is no need for a DO loop as this process will continue to execute till it reaches the end of the file.

CONCLUSION

When repeating similar code is required to complete a task there are numerous ways to accomplish it. For instance, the code can be "hardcoded" by creating the necessary code, in its entirety, for each constraint. Or, a single instance of the code can be copied for each of the subsequent instances, changing only the differing constraints. And there is always the option of macro-tizing the code and using the constraint within the call of the macro, then repeating the call for each instance. While any and all of these would work as solutions they do present similar challenges; difficult maintenance and the risk of omission. Most often code is written prior to the database being in a truly stable, locked form, increasing the likelihood of the code having to be changed or updated prior to final delivery. This creates challenges for maintenance of such programs as it requires wading through each instance for updates and/or changes. All of these options also carry with them the risk of omission, leaving a needed constraint or instance out, which would not produce an error but require heavily on validation processes to ensure correctness. Additional time must be used to address these challenges which can impact a study's overall budget. Instead, we can look to SAS® itself to help create leaner, more dynamic code, reducing post development time for changes and updates and providing the assurance all constraints are accounted for. To do this we first identified the DICITONARY

tables as a resource that could be used with PROC SQL to create a new dataset that contained all the dataset names in the current session of SAS®. Then we used CALL SYMPUT, within a DATA _NULL_, to create two macro variables, a numbered DSN variable and a COUNT variable that held the maximum number of datasets. We then created a macro that used those variables to loop through the dataset names and dynamically execute the code with the changing constraints. As an alternative, a second macro example was given that, instead of creating two macro variables, one macro variable was created containing a space delimited list of all the dataset names. Within the alternative macro, COUNTW was used to count the words in the list and assigned that to the DO statement as the end point. That was followed by the code for the repeating tasks using the SCAN function to pull the dataset names out based on the location of the name within the list in accordance to the 'I' value. To further compress the overall amount of code, and to provide a non-macro, non-SQL option, the final example starts by pulling the dataset names from the SASHELP table VTABLES into a new dataset. Then DATA _NULL_ is used to set the new dataset, and CALL EXECUTE statements created to hold the SAS® code needed for the task. As each dataset is set into the DATA _NULL_, the code is "written" using the current record's information.

REFERENCES

ACKNOWLEDGMENTS

I'd like to first thank Toby Dunn for reminding an "old dog" there are still tricks to learn. I'd also like to thank everyone at PPD for their support and especially those folks that put up with my numerous requests to read the latest draft (you know who you are).

CONTACT INFORMATION

Your comments and questions are always welcome. Please contact the author at:

Jeanina (Nina) Worden
PPD
7551 Metro Center Drive, Ste 100
Austin, TX 78744
Phone: (512) 747-5012
Fax: (512) 440-2971
E-mail: jeanina.worden@ppdi.com