Paper AD-87

Unlock SAS® Code Automation with the Power of Macros

William “Gui” Zupko II, Federal Law Enforcement Training Centers

ABSTRACT

SAS code, like any computer programming code, seems to go through a life cycle depending on the needs of that code. Often, SAS programmers need to determine where a code might be in that life cycle and, depending on what that code is used for, choose to maintain, update, or reuse SAS code in current or future projects. These SAS programmers need to decide what the best option for the code is. Simple code that has few variables or options is easy to leave hard-coded, as it is a quick fix for the programmer to maintain and update this code. Complex code, which can have multiple variables and options, can be difficult to maintain and update. This paper goes through the process a SAS programmer might encounter and talk about times when it is useful and necessary to automate SAS code. Then, it explores useful macro SAS code that helps in maintenance and updating utilities, talking about when an option is appropriate and when it is not. The following SAS code will be utilized: %macro, %let, call symput(x), symget, %put, %if, %do, :into, ods output, %eval, option statements, and %sysfunc.

INTRODUCTION

Many times when I am writing code, I am struck at the truthfulness of the software life cycle. Many times, code that I have written seems to go through a repeating cycle of creating, modifying, and dying. As SAS programmers, it is imperative that we know where in this cycle our code is and what is the strength and weakness of the code in that particular cycle. Many times, when I am asked to create a program, my first question in the requirements gathering is trying to find out how often a program is going to be used. Knowing the life cycle of my code can help me not only prioritize assignments, but also determine the amount of effort involved. For a SAS programmer, those two attributes, time and effort, are the currency with which we ply our trade.

TIME AND EFFORT

Time is an interesting constraint for us to consider. Given any day, there are only so many hours that one can be programming, and even then, programming seems to happen in spurts, not straight through. I love to program in SAS, but I have found that without breaks, the coding gets erratic and starts to develop continuity holes. Time for SAS programmers has several different main components when dealing with any one specific program:

1. Creating Code
2. Updating Code
3. Maintaining Code

Creating code is the actual creation of the program, sitting down and writing it out. Updating code is making sure that code is looking at the correct source data and variables. Maintaining code is making those changes to code to optimize/eliminate solutions/problems that arise with code.

When time is paired with effort, we find some interesting correlations. When writing any segment of code, I am constantly evaluating the following things:

- Repeatability
- Complexity
- Usability

Repeatability is the amount of times I might see code snippets appear. For example, I often am using date manipulation, and to make things easier on my audience and for me, I use the same date formats, just copying and pasting the same 6 lines of code. Complexity is how difficult the code is to write. Sometimes, especially with nesting, it gets difficult to know where one starts and the other stops. Finally, usability is how useful I see the code in future projects with minor changes, unlike repeatability, which needs no modification.

So, we have our two constraints, time and effort, both of which need to be managed effectively as SAS programmers. However, I have found that time is an absolute constraint; it is impossible to get more than 24 hours in a day. Effort is a dynamic constraint, as in I, the SAS programmer, determine how much effort to give a particular program. But, in some cases where code is usable for other projects, I can increase the complexity of the code and reap huge time savings. This can sometimes be difficult to justify, because this can also increase the time on the front-end. While difficult, if you are sometimes wondering if there is a way to reuse code, then often, there is.

MACRO TOOLS-INCREASE COMPLEXITY, SAVE TIME

SAS runs everything in memory. Those variables or code that are stored solely in that memory are macros. One of the greatest powers of macros is their ability to lessen future effort in maintenance and updating. However, when code is solely in macros, the complexity increases, sometimes exponentially. Anything can be coded in SAS, but sometimes the code to accomplish that can be both herculean and labyrinthine.

The following subsections will detail different macro tools, functions, and abilities that are useful for maintenance and repeatability. While I will not be going into detail about the syntax of some of these, they are all available on support.sas.com.

%INCLUDE

%include is an interesting function that is only useful for repeatability. If there is code that is often used, such as formatting data sets or creating parameters, this function takes SAS code written in one SAS program and reads it into the current program.

%MACRO

%macro is the call to start a macro program. Unlike %include, %macro is written within the program unless it has been saved as a global macro or part of a macro library. Once the macro is named, the code will not run unless called by %(macro name). Like %include, it can be used to call things up time after time. However, it is only within %macro that certain macro functions can be used, such as %if, %else, %return, and %do.

%IF

Like the if statement in a DATA step, %if is a conditional that can be used for macro variables. %if can be used to make dynamic macro statements, conditional code, or set parameters. If %if uses the %then %do, SAS code can be written to either run or not run based on the conditional. Otherwise, only macro code can be used in the conditional.

%ELSE

Just like %if, can be used as %else or %else %if.

%RETURN

This function is very useful to stop a macro. Some of my macros can take 20-30 minutes to run, based on data size. If there is an incorrect parameter or something that does not make sense, usually start date>end date, then %return will end a macro before it goes into an endless loop. Very useful when used with %if.

%DO

Like the do statement in a DATA step, this function will start a do loop. Unlike do, however, is the fact that %do creates a macro variable for EACH iteration. This is extremely helpful in separating data into subsections.

%LET

%let is how a macro variable is created in open code. It cannot work in a DATA step or other function, however.

CALL SYMPUTX

Before SAS 9, we had call symput, no x. This function is to create a macro variable within the DATA step. While they both do the same thing, call symputx accomplishes the same task with less work, so I often use it.

SYMGET

Unlike call symputx, symget pulls macros into the DATA step. While not very useful with single macros, when there are large numbers of dynamic macros, this can be an efficient solution.

:INTO

Like call symputx, this creates a macro out of a data set, but using proc sql instead of a DATA step. A big difference is the ability to use separated by functionality, meaning that it can create lists much more efficiently then call symputx or %let.

%PUT

As mentioned before, macro variables are in memory. %put is one way to print out macro variables into the SAS log in order to make sure the correct variables/parameters are being set. When code gets complex, %put is a quick and dirty way to make sure that the right information is populating. While other ways give more information, %put puts it in only if you want it.

OPTION STATEMENTS: MLOGIC MPRINT SYMBOLGEN

Whenever testing macros, these options should always be on. Mlogic shows how conditional statements resolve, such as those in %if. Mprint shows when %macro code runs, seeing if there is a problem in progression. Symbolgen shows how macro variables are changing in memory and how the code is resolving. When debugging, these options are lifesavers. However, in production, they quickly eat up memory in the log and can cause problems there, which is why I use %put in production.

ODS TRACE

This statement is used with ODS OUTPUT, enabling the user to know how to call output information.

ODS OUTPUT

This extremely useful statement allows SAS data sets to be created based on SAS output results. For example, if I wanted to see the regression line created by proc reg, I would use ODS OUTPUT to make that a SAS data set that could then be read.

%EVAL

%eval creates the evaluation of macros. If I had a macro that equaled 4, but put in macro+1, instead of saying 5, it would say 4+1. %eval allows the mathematical function to happen so that I get a text value of 5, instead of 4+1.

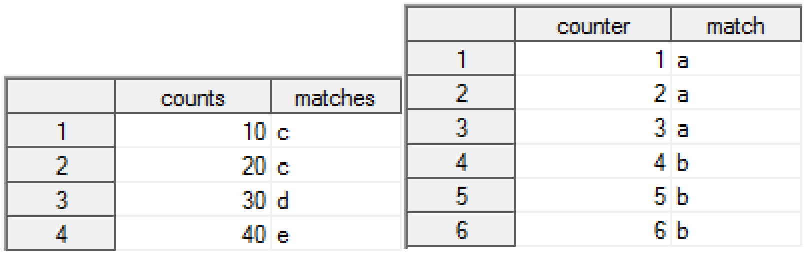
%SYSFUNC

%sysfunc allows for the rendering of multiple macros, which opens up many possibilities. I usually use it to format dates, since SAS reads dates in as SAS dates, not formatted dates.

EXAMPLE OF MACRO COMPLEXITY SAVING TIME

Now that we have a process of determining when to use macros and the tools to use them, let us try an example of macros in action. For demonstration purposes, we will have two data sets, a and b, which each have two variables. For purposes of illustration, these variables are alike, but not exactly the same.

Output 1 & 2. Data Set B and A



	counts	matches
1	10	c
2	20	c
3	30	d
4	40	e

	counter	match
1	1	a
2	2	a
3	3	a
4	4	b
5	5	b
6	6	b

Now, let us say that we wanted to separate the variable match or matches into distinct data sets, so that all the a's are in their own data set, all the b's in theirs, and so on. In order to write this simple code, we will have the following code.

Code 1. Hard-code Example

```
data count_a;
set a;
where match='a';
run;
```

Now, in order to hard-code this in every case just in this example, it would take at least five times to get each match. That is assuming we do not make any mistakes in coding, which would increase that potentially exponentially. So, assuming our process says that we need to make this more complex to save later time, we need to identify commonality in the code that a macro could use as a substitution. Think of a macro like a pronoun, it just needs to be clear what it is referencing.

In the hard-code example, text that is green means a data set name. Text that is purple means that it is a variable name. Text that is yellow means that it is a variable value.

I usually prefer to start with automating variable values, but that is personal preference. However, I find that keeping the data set constant means that I can find errors more easily. This is also a great tip for debugging-Change ONE THING AT A TIME!!! When many changes happen, it can be very difficult to find where an error might be occurring, and errors in macro code can cause endless loops that would necessitate force closing SAS.

So, focusing on variable value, there are some considerations immediately to consider. Sometimes there are two values for match, or three values in matches. So, I need to create a lookup table that will determine how many values there are. Since this is a descriptive summary, proc freq is a great tool to use, outputting the output into a new data set that will be a_count.

Code 2. Creating a Lookup Table

```
proc freq data=a noprint;
tables match/out=a_count ;
run;
```

Now that I have a lookup table, I need to know how many times a variable name is found. Using call symputx, I will count the number of times a variable name shows up in the proc freq output.

Code 3. Finding Number of Values in Lookup Table

```
data _null_;
set a_count end=last;
if last then call symputx('lastnum',_n_);
run;
```

The thing to remember about call symputx, the macro variable is not created until the run statement. So, do not try to use a macro variable in the data set that you are creating. Now, SAS knows how many times the variable match shows up, meaning SAS knows how many times the code needs to be run.

Since I want to run the same code multiple times, that means a do loop. Since macro do loops allow me to specify the iteration, I can use that to look in the lookup table to go through each value there individually, put it in a macro, and then use it later. Since the macro names will be changing with each iteration, I will need to use && to resolve macro names that have my macro iteration variable, in this case i, in them. Otherwise, I will stay with just one &.

Code 4. Using iterations in Lookup Table to parse each Value

```
%macro autocode;
%do i=1 %to &lastnum;
data _null_;
set a_count;
if _n_=&i then call symputx("num_&i",match);
run;
data count_&&num_&i;
set a;
where match="&&num_&i";
run;
%end;
%mend autocode;
%autocode;
```

Here we have a %do loop, going from 1 to the macro variable lastnum, which it got from code 3. Then, when going through the first iteration, the macro variable i resolves to 1, where I use my lookup table to find that the first line (_n_) is a. Call symputx is used to create a macro variable num_1, using the iteration macro variable i. Finally, a new data set is created called count_a, as num_1=a, pulling from data set a only those cases where match=a.

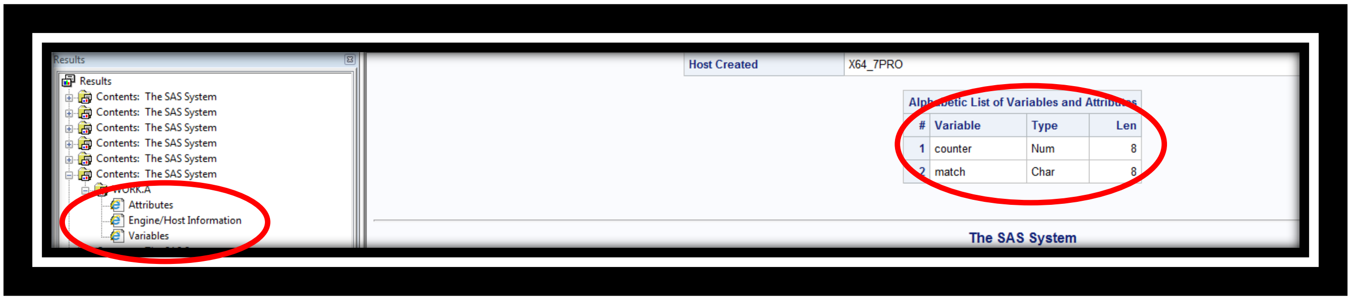
So, to do the same thing hardcoding, it took 4 lines for each data set to be created. Here, it took 20 lines of code and some interesting mental gymnastics to create the same thing. The difference, if I wanted to do match=b, I would need another 4 lines of code. However, the autocode macro would do b immediately, no additional coding required.

So, in this case, the autocode macro would not be worthwhile, since I could accomplish this very quickly by hardcoding. But, what if I had 100 values? What if I had to change this often? I usually always hard-code the first attempt, because then I can see what commonality there is and how to modify the code into a macro. Or, I might see that it is not worth changing any more, since I will not be required to use the code again.

Now it is time to get really crazy. This code would work for data set a, but would not work for data set b. While it would find the values, the variable names would have to be the same. In this case, we cannot change just one thing, but have to change both variables and data set names since they are intrinsically linked.

So, everywhere we have a, we are now going to change that to &dataset. But, how do we find out what the data set structure looks like? Proc contents tells us data set structure, but the out= option gives us details about the structure, not the variables that are included. However, when we run proc contents, we can see the following info:

Output 3. Proc Contents



Host Created	X64_7PRO

Alphabetic List of Variables and Attributes

#	Variable	Type	Len
1	counter	Num	8
2	match	Char	8

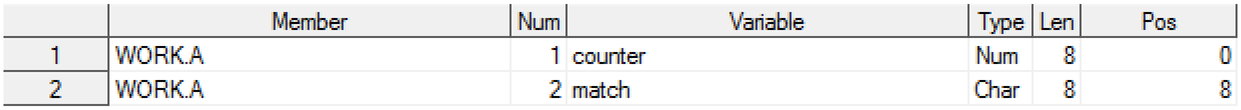
The SAS System

So, SAS is producing the information that we need in the output window, not as a data set. Using our tools from above, ODS OUTPUT can be used to create data sets from output windows. Since it is creating this from an output window, do not use the noprint option. Nothing will be created anywhere! So, when we use ODS OUTPUT, we get the following code and result:

Code 5. ODS OUTPUT

```
ods output "Variables"=vari;
proc contents data=&dataset;
run;
ods output close;
```

Output 4. Variables Data set Created by ODS OUTPUT

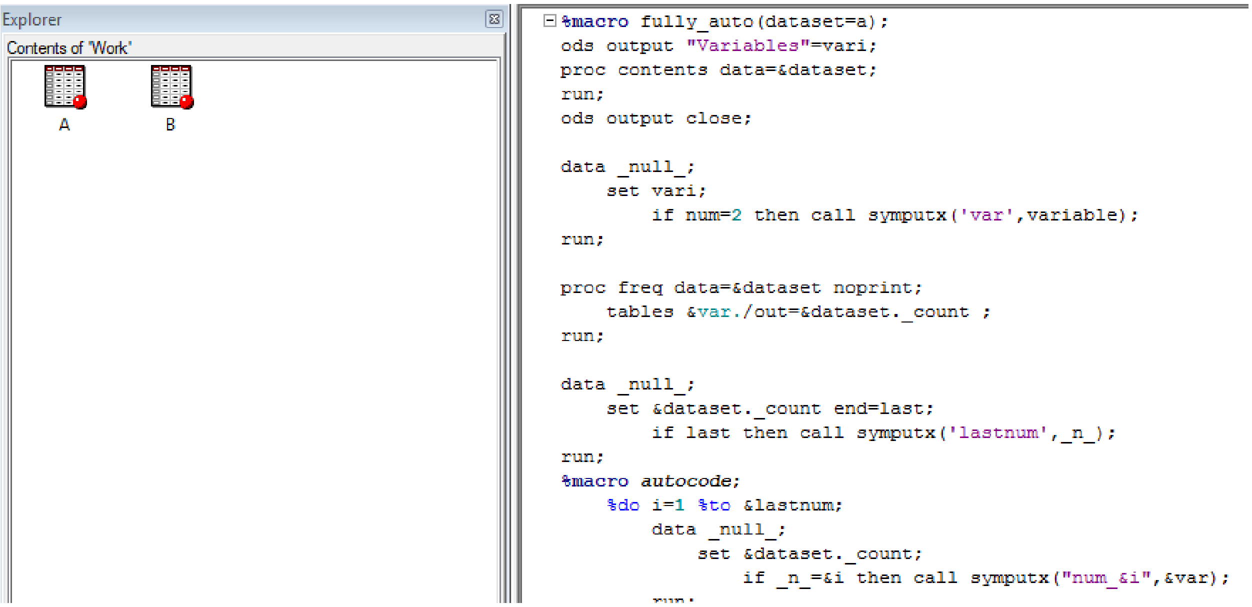


	Member	Num	Variable	Type	Len	Pos
1	WORK.A	1	counter	Num	8	0
2	WORK.A	2	match	Char	8	8

Always look at ODS OUTPUT data sets before trying to use them, because they can sometimes label variables a little unusually. Here, we see that the variable match is the second variable. Since data set b is structured like a, with matches as num=2, we can use that. However, since data is usually unstructured, when creating macros, it is a best practice when using this step to use a format statement to put it into the first position.

Now that I have the code, I enclose everything in another macro, called fully_auto, and can set the data set name to the one I need to test. I will start with data set a. When I start, I only have data sets a and b:

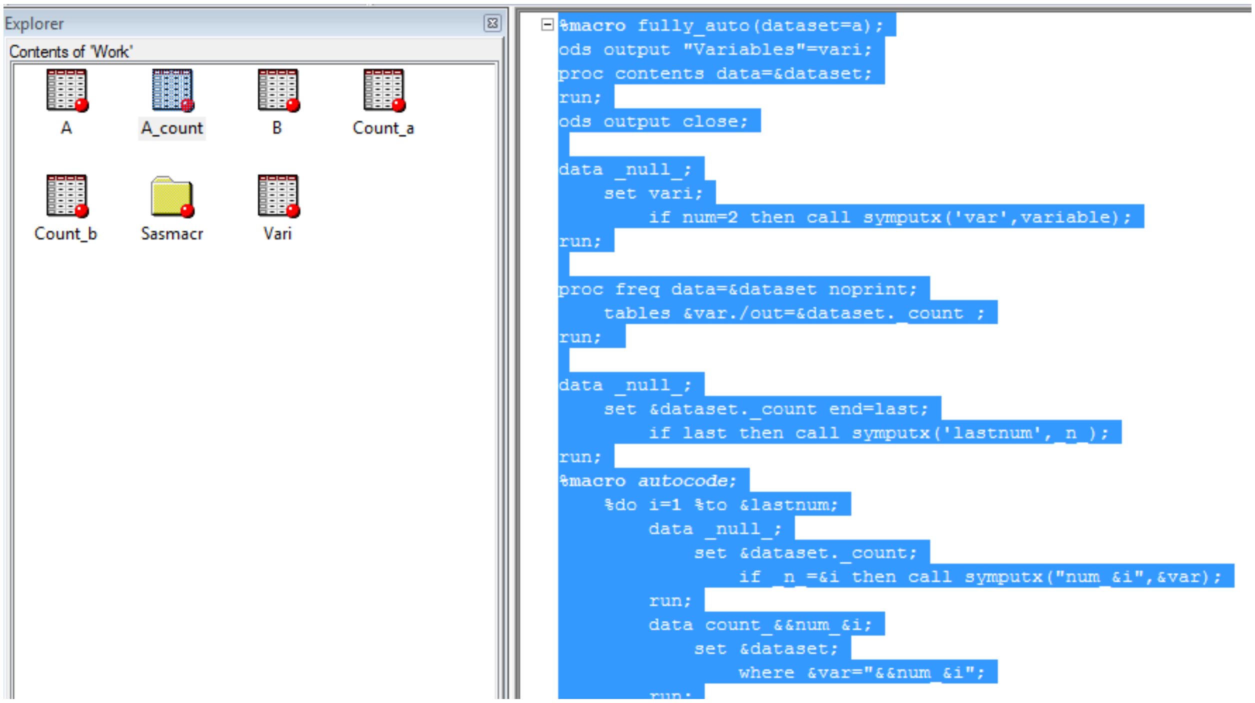
Output 5. Before the run



```
%macro fully_auto(dataset=a);
ods output "Variables"=vari;
proc contents data=&dataset;
run;
ods output close;

data _null_;
    set vari;
        if num=2 then call symputx('var',variable);
run;

proc freq data=&dataset noprint;
    tables &var./out=&dataset._count ;
run;

data _null_;
    set &dataset._count end=last;
        if last then call symputx('lastnum',_n_);
run;
%macro autocode;
    %do i=1 %to &lastnum;
        data _null_;
            set &dataset._count;
                if _n_=&i then call symputx("num_&i",&var);
        run;
```

Now, when after a run, always check the log for errors. After my run, however, I find the following information:

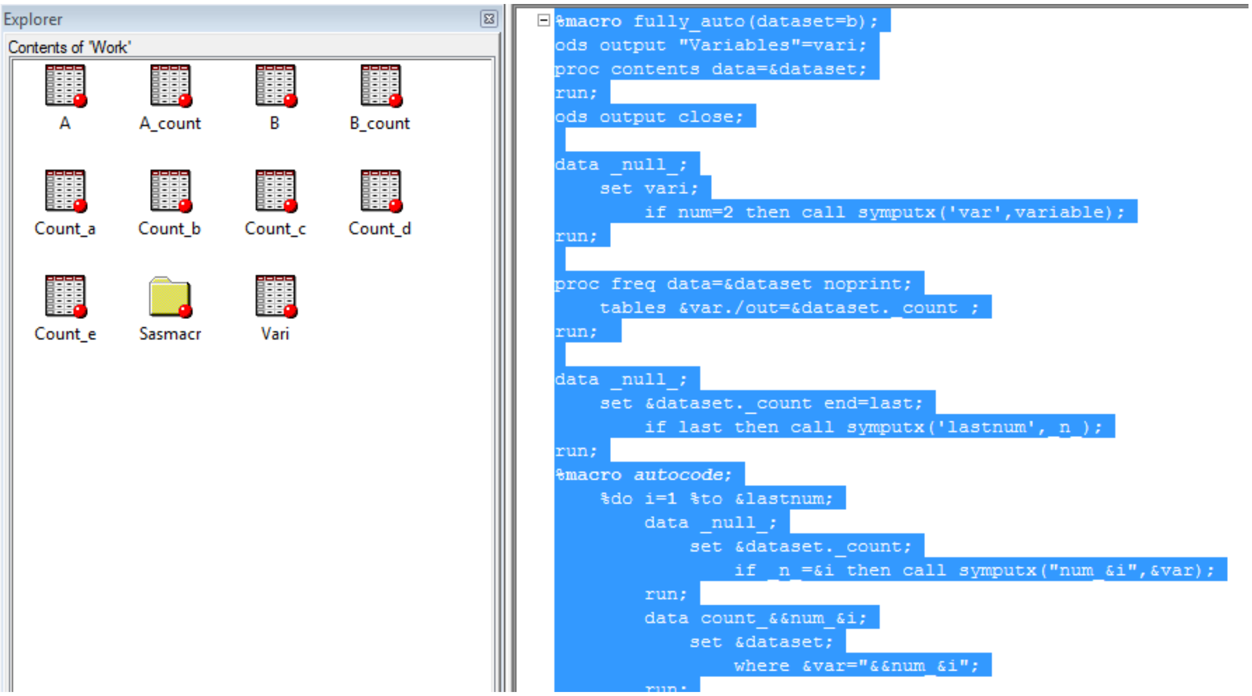
Output 6. After Data Set A



```
%macro fully_auto(dataset=a);
ods output "Variables"=vari;
proc contents data=&dataset;
run;
ods output close;

data _null_;
    set vari;
        if num=2 then call symputx('var',variable);
run;

proc freq data=&dataset noprint;
    tables &var./out=&dataset._count ;
run;

data _null_;
    set &dataset._count end=last;
        if last then call symputx('lastnum',_n_);
run;
%macro autocode;
    %do i=1 %to &lastnum;
        data _null_;
            set &dataset._count;
                if _n_=&i then call symputx("num_&i",&var);
        run;
        data count_&&num_&i;
            set &dataset;
                where &var="&&num_&i";
        run;
```

I see that the count_a and count_b data sets are created, no errors are in the log, and the data, when checked, comes back correctly. Now, it is time to check this with data set b.

Output 7. After Data Set B



```
%macro fully_auto(dataset=b);
ods output "Variables"=vari;
proc contents data=&dataset;
run;
ods output close;

data _null_;
    set vari;
        if num=2 then call symputx('var',variable);
run;

proc freq data=&dataset noprint;
    tables &var./out=&dataset._count ;
run;

data _null_;
    set &dataset._count end=last;
        if last then call symputx('lastnum',_n_);
run;
%macro autocode;
    %do i=1 %to &lastnum;
        data _null_;
            set &dataset._count;
                if _n_=&i then call symputx("num_&i",&var);
        run;
        data count_&&num_&i;
            set &dataset;
                where &var="&&num_&i";
        run;
```

So, we can see that the new data sets count_c, count_d, and count_e are created, data checks correctly, all hurrahs!

This is a living process, because what may be useful one day is not the next, while a piece of SAS code that was created to explore becomes a necessity the next. SAS code needs to be examined and categorized, but the SAS programmer can save a lot of time on the back end by doing the up-front code. Even better, it automates processes and removes human error from the equation. If you are currently wondering if you could save time, you probably can!

CONCLUSION

Like all computer code, SAS code can have a life of its own. With the use of macros, code can be automated in such a way that significant time savings can be had. There are a great number of useful macro tools to automate data, such as pulling data out of data sets into macro variables, formatting and streamlining data or reporting processes, and creating parameters for errors or reporting purposes. SAS programmers are always constrained by time and effort, but in many cases, adding a little effort and time up front can reap huge time benefits in the future.

ACKNOWLEDGMENTS

I would like to thank FLETC and SAS for the opportunity to write this paper.

This paper is released to inform interested parties of (ongoing) research and to encourage discussion of work in progress. Any views expressed on statistical, methodological, technical, or operational issues are those of the author and not necessarily those of FLETC.

CONTACT INFORMATION

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

William Zupko II
U.S. Department of Homeland Security, Federal Law Enforcement Training Centers
William.ZupkoII@fletc.dhs.gov

APPENDIX 1: FULLY AUTO SAS MACRO

```
%macro fully_auto(dataset=b);
ods output "Variables"=vari;
proc contents data=&dataset;
run;
ods output close;

data _null_;
      set vari;
            if num=2 then call symputx('var',variable);
run;

proc freq data=&dataset noprint;
      tables &var./out=&dataset._count ;
run;

data _null_;
      set &dataset._count end=last;
            if last then call symputx('lastnum',_n_);
run;
%macro autocode;
      %do i=1 %to &lastnum;
            data _null_;
                  set &dataset._count;
                        if _n_=&i then call symputx("num_&i",&var);
            run;
            data count_&&num_&i;
                  set &dataset;
                        where &var="&&num_&i";
            run;
      %end;
%mend autocode;
%autocode;
%mend fully_auto;
%fully_auto;
```

APPENDIX 2: DATA SET CREATION

```
data a;
      input counter match $;
      datalines;
            1     a
            2     a
            3     a
            4     b
            5     b
            6     b
      ;
run;
data b;
      input counts matches $;
      datalines;
            10    c
            20    c
            30    d
            40    e
      ;
run;
```