Paper 125-2019

Tips for the lazy typist (and other strategies for smart coding)

Melissa R. Pfeiffer, Children's Hospital of Philadelphia, Philadelphia, Pennsylvania;
Kristina B. Metzger, Children's Hospital of Philadelphia, Philadelphia, Pennsylvania

ABSTRACT

Time and experience are your best tools for developing your SAS coding prowess; experience includes both working on a variety of tasks and working with different programmers. Through our combined 40 years working with SAS, we have worked on a wide assortment of projects with different data management needs and collaborators. Through these experiences we have learned to employ techniques that reduce the likelihood of introducing errors into our coding, help identify errors when they do appear, and generally make our programming lives easier. These tips include strategies for dealing with a growing number of user-defined formats, employing keyboard macros and %INCLUDE statements for commonly used code, establishing standardized naming conventions, using arrays and DO loops to cut down on coding, and cleaning up our "WORK" space. We all become better programmers when we share tips and strategies; our hope is that the suggestions we share expand your programming repertoire.

INTRODUCTION

Through many years of experience in SAS programming, we have learned several tricks that have made our programming more efficient and expanded our analytic and data management abilities. Few tricks are as exciting as those that allow us to type less AND reduce the likelihood of typing errors. Here are some of our favorites.

MSGLEVEL = I [WHAT'S HAPPENING?]

Standard error and warning messages in the SAS log are wonderful for alerting you to critical issues in your program, and standard notes can provide useful details that can help confirm what you thought (like "the new data set has the expected number of observations") or plan activity (like "last time this took a while to execute – time to go pour more coffee.") But did you know that there are information messages too? These can be very helpful. For instance, perhaps you are merging two data sets in a data step, but unbeknownst to you both contain a common variable. If this variable is not one of your BY variables – that is, you are not merging your data sets using this variable – then the values from the data set named second in your MERGE statement will overwrite the values from the data set named first (Display 1).

```
INFO: The variable dob on data set FORFAKE.FAKEMERGEPART1 will be overwritten by data set
      FORFAKE.FAKEMERGEPART2.
```

Display 1. Example of Information Message – Variable Overwritten

Knowing this can save you and your programming colleagues from confusion down the line. To instruct SAS to show you these messages, simply submit the following global option:

```
OPTIONS MSGLEVEL = i ;
```

One helpful practice, since you don't necessarily know when a useful informational message might appear, is to submit that option at the start of each session. If too many messages are being produced, you can turn off the information messages by changing the option back to the default of N:

```
OPTIONS MSGLEVEL = N ;
```

WHERE ALSO [NOT ONLY BUT ALSO]

You may remember from English class the "not only…but also" construction that lets you describe two things – "She's not only beautiful but also very smart." Well, SAS has construction that lets you add more than one limiting condition. You've probably used a WHERE statement to limit observations in a data step or procedure, such as:

```
WHERE gender = 'M';
```

If you want to add another condition, you could use the AND conjunction to add the other condition to your WHERE statement:

```
WHERE gender = 'M' AND age > 25;
```

But you have another option: the WHERE ALSO statement. This allows you to have a separate statement that augments the WHERE statement rather than replacing it or modifying it:

```
WHERE gender = 'M';
WHERE ALSO age > 25;
```

A benefit of this is that the WHERE ALSO statement can be easily and quickly commented out in situations in which you are experimenting with the restrictions you are implementing. Similarly if you are iterating through different restrictions in a series of procedures or other processes, using WHERE ALSO can simplify your macro code. You can further take advantage of coding flexibility by having multiple WHERE ALSO statements:

```
WHERE gender = 'M';
WHERE ALSO age > 25;
WHERE ALSO race = 1;
```

You now have a separate statement for each of the conditions you want as restrictions; any or all of the statements can be commented out—independently or together—including the WHERE statement.

WHERE SAME AND is equivalent language, but WHERE ALSO may be more intuitive.

STANDARDIZED NAMING CONVENTIONS [UNITED WE STAND]

Establishing a naming convention in which similar variables and similar data sets have a particular structure can help remove naming guesswork. That removes or reduces time spent thinking about what a variable should be called, and it also makes it easier to remember the names of variables when you need to reference them. If other programmers are working with the same data sets, it also increases understandability and transparency in meanings for others. Some naming conventions have obvious meanings, such as using the prefix "date_" for date variables, if your data set has several. We have extended this principle to other series of similar variables. For example, if you have variables for age on the 15th of the month for several years, a standard naming convention could be "Age_yyyymm," as in Age_201701 and Age_201702. A series of variables that indicate the existence (or not) of particular co-occurring conditions, such as ADHD, anxiety, or seizures, could be named cooccur_adhd, cooccur_anxiety, and cooccur_seizures, respectively.

You may also want to begin the name of temporary variables – like indexing variables used in DO loops – with an underscore to help identify them as temporary variables that are not ultimately needed and to facilitate dropping them. Other advantages to using such standard naming conventions include being able to easily view the characteristics of similar variables in a table of contents and be able to take advantage of name prefix lists.

NAME PREFIX LISTS [KEEP IT SIMPLE]

One of the benefits to standard naming conventions that use common prefixes – such as "Crashes_" or "_" – is that programmers can then take advantage of name prefix lists, which allow you to refer to all variables that begin with a specified character string. A colon following a name prefix tells SAS that you want to refer to any variable that begins with that prefix, regardless of what follows the prefix. This can greatly simplify your code with functions (summarizing the number of crashes as sum(of crashes_:)) and statements (drop _:;). In the first example, the sum of all variables beginning with "crashes_" is generated and in the second, all variables beginning with an underscore are dropped, regardless of what follows the underscore. Use this powerful capability advisedly, though. Since any variable that begins with the prefix

is included, it is easy to mistakenly include a variable you did not intend to. For instance, perhaps the data set has the number of crashes each month over a two year period but it also has a variable – crashes_total – that already summarizes the number of crashes. That variable would be included in a name prefix list crashes_:, and your summary would be incorrect.

USING ARRAYS TO INITIALIZE AND RETAIN [TWO-FOR-ONE DEAL]

SAS arrays – temporary groups of variables – are versatile tools that can help you reduce your coding when you want to conduct similar processing for several variables. There are one dimensional arrays, multi-dimensional arrays, and temporary arrays. There are many resources (including proceedings papers from SAS Users Group conferences) that discuss the capabilities of arrays and how to use them. What we would like to address briefly is the often overlooked opportunity to use arrays to initialize and retain variables. You can use arrays to initialize either numeric or character variables (or numeric or character temporary data elements) – to do so you simply assign initial values in parentheses after the array elements, remembering to put character values inside of quotes. You can do this as a list, with one value for each of the elements in your array, or you can use an iteration factor with sublists. For example, perhaps your array elements (variables crashes_total, fatal_total, and injury_total) should have initial values of 0. This can be accomplished with the code:

```
ARRAY forsum{3} crashes_total fatal_total injury_total (0, 0, 0) ;
```

or with the code:

```
ARRAY forsum{3} crashes_total fatal_total injury_total (3*0) ;
```

where “3” is the number of iterations and 0 is the value.

These methods of initializing can be mixed. Perhaps you have 10 elements in your array that should have the values 100, 90, 80, 60, 60, 60, 50, 40, 0, 0. You can assign those values with the code:

```
ARRAY newvars{10} x1-x10 (100, 90, 80, 3*60, 50, 40, 2*0) ;
```

There are a few things to keep in mind when using arrays to initialize variables. First, values are assigned to elements according to their position – the first value is assigned to the first element, the second value is assigned to the second element, etc. Take care to match the number of values listed with the number of elements in your array. Second, you can use either blank spaces or commas to separate the values in your initial value list. But here’s the bonus: when elements are assigned initial values, they behave as if they were named in a RETAIN statement! This means that if you’re creating new variables and finding yourself naming them in an ARRAY statement and naming them in a RETAIN statement, you do not need to do both. By simply setting your initial values in the ARRAY statement, you can omit the RETAIN statement (or comment it out until you’ve tested this for yourself).

An example of when this could be helpful is summarizing information across records for individuals. Perhaps you have a data set with multiple records per person, each with a diagnostic code, and you want to search through those diagnostic codes to create flags for whether the person has particular co-occurring conditions. Since you will ultimately want to keep one record per person with the indicators for the co-occurring conditions, you want to retain information across observations. If you’re willing to set initial values for your variables such as by starting off with the values coded as no/0, then you can retain your indicator variables, initialize them to 0, and group them in an array with one statement:

```
ARRAY cond{3} cooccur_adhd cooccur_anxiety cooccur_seizures (3*0) ;
```

Remember that you need to re-set values, to 0 in this example, when you reach the first record for a new person so that you don’t inadvertently retain information from one person to the next. This is true regardless of the method – a retain statement or an array statement – used to retain values.

DO OVER TO RESET VALUES [JOY IN REPETITION]

Speaking of arrays, you’re probably familiar with iterative DO loops that allow you to execute a statement (or statements) based on the value of an index variable. That index variable is often used to reference an

element in your array. For instance, the code below allows you to change the value for an indicator variable to 1 if a corresponding diagnosis code is found:

```
ARRAY cond{3} cooccur_adhd cooccur_anxiety cooccur_seizures ;
ARRAY codes{3} $ _holdvar314 _holdvar300 _holdvar345 ('314', '300', '345');
 DO _a = 1 TO 3;
    IF diagnosiscode = codes{_a} THEN cond{_a} = 1;
 END;
```

Well, you can also use a DO OVER loop with non-indexed arrays. Non-indexed arrays are like typical arrays except that they do not (in fact, cannot) specify the number of elements in the array. If you are going to take the same action on all of the elements in your array, regardless of element position, then you can use a DO OVER loop instead of an iterative DO loop. For example, perhaps you want to change the values in all of your character variables to be entirely upper case. This can be accomplished easily with a non-indexed array and a DO OVER loop. Begin with the ARRAY keyword, name your array (for example "allchar"), and use the _CHARACTER_ variable list to include all character variables in your data set in the array. Then you can use a DO OVER loop to execute the same action, in this case upcasing your variables, on all of the elements of the array, regardless of their position in the array:

```
ARRAY allchar _character_ ;
DO OVER allchar ;
   allchar = upcase(allchar) ;
END ;
```

Remember that, like all DO loops, your DO OVER loop needs to have a corresponding END statement.

FORMATS IN EXCEL [USE ALL THE TOOLS IN YOUR TOOLBOX]

Some teams may choose to use format catalogs to store and access user-defined formats. That capability did not work for multiple programmers on our remote sharing network. However, correctly assigning user-defined formats to variables is critical for understanding and working with our data. We needed a system to ensure that all of our programmers could access appropriate and up-to-date user-defined formats, view formats to understand the labels being applied to each value, and could add to the repository of user-defined formats as needed. Our solution was to create a user-defined format repository that we maintain in Excel and read into each SAS session. The Excel file is easy to maintain; search and filter features make it easy to find particular formats as well as to determine if a particular format has already been created. The Excel file, at minimum, needs to have the following fields: fmtname, for the format name (repeated for each value); start; end; HLO, for High, Low, Other; label, for the text associated with each value or range; and type, with values of either "C" for character formats or "N" for numeric formats (Display 2).

Fmtname	Start	end	HLO	Label	Type
Racefmt	W	W		White	C
Racefmt	B	B		Black	C
Racefmt	A	A		Asian	C
Racefmt	NA	NA		Native American	C
Racefmt	O	O		Other/multi-race	C
Racefmt	U	U		Unknown	C
agegroupfmt	0	17		Child (0 - 17)	N
agegroupfmt	18	64		Adult (18 - 64)	N
agegroupfmt	65		H	Senior (65+)	N
yesnofmt	0	0		No	N
yesnofmt	1	1		Yes	N

Display 2. Example of Excel File With Information Needed to Establish User-Defined Formats

The start and end fields let you indicate a range of values to be associated with the same text; if you do not need a range, you can have the same value in both the start and end fields. The HLO field lets you use HIGH, LOW, or OTHER keywords instead of start or end values. Sort your file with the character formats before the numeric formats. We choose to sort our file by format name within character/numeric format type, although that is not required.

This file can then be accessed by SAS to create your user-defined formats (Display 3). First, establish a FORMATS libref using the Excel engine and pointing to your Excel file. Then, use the CNTLIN option on the PROC FORMAT statement to point to that libref and sheet name (in our example, our sheet name is creatively called “formats”). Executing that code creates your user-defined formats. We then execute an additional statement that uses the CLEAR option on the LIBNAME statement to release the Excel file (that is, end SAS’s hold on the file) so that others can access the file.

```
1  libname formats excel "K:\Projects\Young Drvr\2. Documentation\Formats 2016.06.01.xlsx" ;
2
3  proc format cntlin = formats.'formats$'n ;
4  run;
5
6  libname formats clear;
```

Display 3. Example of Code to Create User-Defined Formats From an Excel File

%INCLUDE [DOUBLE DIPPING]

Rather than having the above code to read in an Excel file to create user-defined formats in each program – and potentially having to change the path to the Excel file in multiple programs if a new version is created – that code could be a separate program that’s run before other programs that use these formats. You could include an annotation in your program to document and remind users of this process, such as:

```
/**** Execute program “1. Create user defined formats” before running the
next steps ****/
```

However, that note could be overlooked. If it’s not, the process still requires the user to open another program and execute that before proceeding. The process of running another program from within your current program is simple with a %INCLUDE statement. The %INCLUDE statement allows you to read an entire file into your current SAS session and submits that code immediately. It is almost as though you have pasted that code into your program and executed it. The structure of the statement is simply %INCLUDE followed by the source that describes the location of the information you want to access; when the source is a file specification such as a SAS program, enclose it in quotes. We use %INCLUDE

statements routinely to read in a common program that establishes librefs and user-defined formats. The program (partially shown in Display 3) is easily referenced with the %INCLUDE statement:

```
%INCLUDE "K:\Projects\Young Drvr\5. Analysis\1. Create latest user defined formats.sas";
```

That program can be updated as needed – in only one place – without needing to change the %INCLUDE statement.

%INCLUDE statements can be used for longer programs that perform data management or analytic tasks that you commonly use. For instance, you may routinely create a standard demographics table—often referred to as "Table 1" by epidemiologists—that provides the proportions of an analytic population by age, sex, race/ethnicity, and other demographic characteristics, as well as comparative statistics of these variables between two groups. If you develop standard code that uses macro language to produce this demographics table, you can execute the code from within your main analytic program using a %INCLUDE statement:

```
%INCLUDE "K:\Projects\Standard Code\macro for Table 1.sas";
```

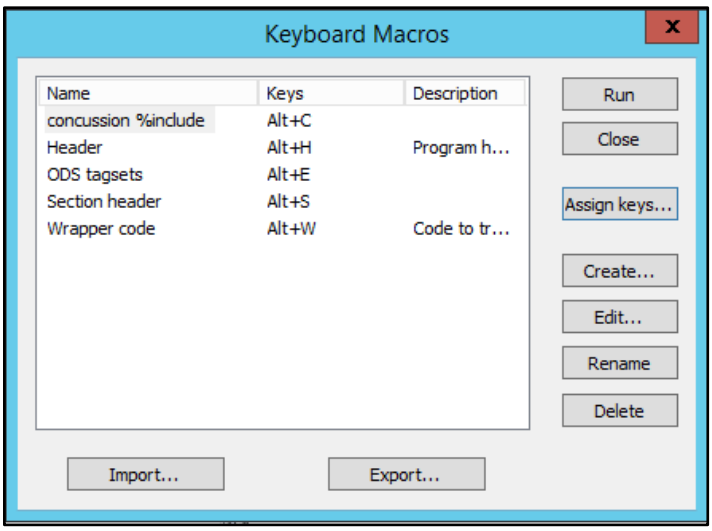
This strategy is a flexible alternative to using macro catalogs, which in some circumstances (like ours!) are not available to multiple programmers on a remote access network. Additionally, pointing to standard code using a %INCLUDE statement reduces potential error introduced by copying and pasting these coding steps into each program that creates a Table 1.

KEYBOARD MACROS [QUICK DRAW]

The more often your fingers type code that you use frequently, the easier it is to remember the syntax. But why overload your memory – and risk forgetting your best friend's birthday – when you can reduce that code to just a few keystrokes?

That's right, you can bring in long and complicated (as well as short and simple) code very easily with the use of keyboard macros. Instead of retyping code or finding, copying, and pasting code from another program, you can use a specific key stroke combination to paste desired code into your program. We use keyboard macros for things like our program header template, commonly used %INCLUDE statements, and shells for ODS TAGSETS code.

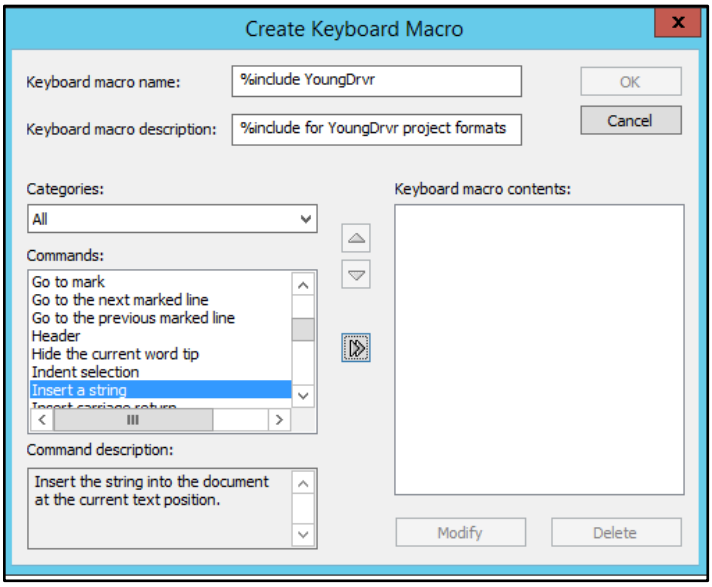
Creating a new keyboard macro is easy. First, copy the code or annotations that you want included in your keyboard macro. Then, from the Enhanced Editor window, open the Tools menu, select Keyboard macros, then Macros. A window showing your current keyboard macros and providing the tools to run, alter, or create macros appears (Display 4).



Display 4. Primary Keyboard Macros Window, Used to Create Macros and Assign Keys As Well As

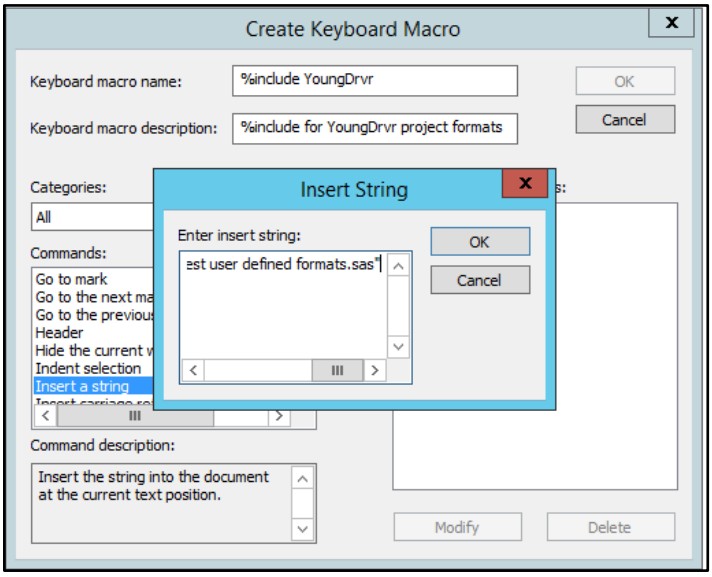
to Edit, Rename, and Delete Macros

Click Create. Enter a keyboard macro name and a description (if you wish). Then, with “Insert a string” highlighted in the Commands box, click on the double arrows pointing to the keyboard macro contents box (Display 5).



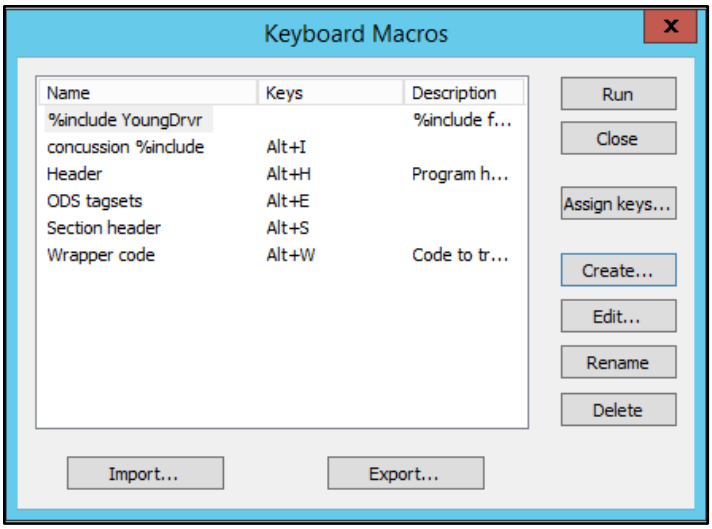
Display 5. Window for Creating Keyboard Macros

An “Insert String” box opens, in which you can paste the code you copied—the code you want to appear when you use the keyboard macro (Display 6).



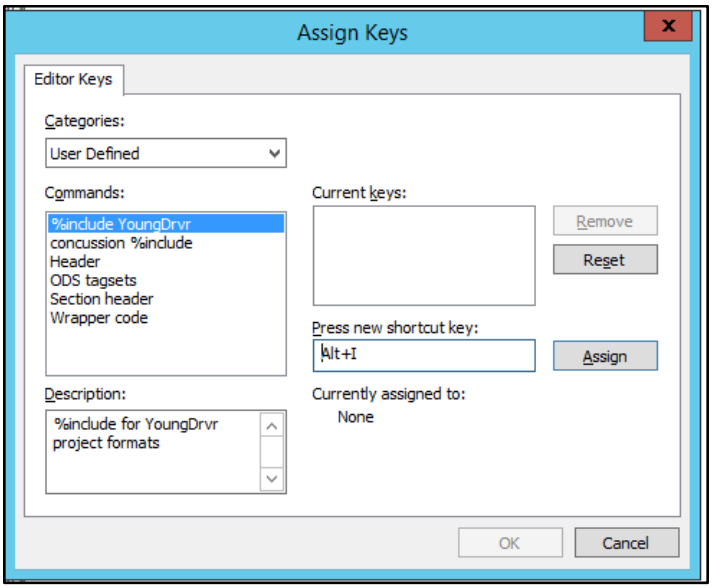
Display 6. Sub-Window to Assign Text to a Keyboard Macro

Click OK. Click OK again. Now to complete the process, from the main Keyboard Macros window, select “Assign keys” (Display 7).



Display 7. Primary Keyboard Macros Window, Now Showing Our New Keyboard Macro

With your new macro highlighted, and your cursor in the “Press new shortcut key:” box, press the keystrokes you want associated with this keyboard macro, such as ALT+I (Display 8). A combination of ALT plus a letter is often a good choice; those keystrokes are less likely to already be associated with a capability than CTRL plus a letter. A message lets you know if that shortcut is already in use. Note that you do not need to remove the word “None” first—it will change as soon as you pick your new keystrokes.



Display 8. Window to Assign Keys to Macro, Showing That the Shortcut Key Alt+I Is Not Currently Assigned

Then click on Assign, OK, and then Close.

Although this process seems lengthy, it will save you loads of time compared with typing out commonly used code repeatedly. There is another process to create a keyboard macro in which you record a new macro. However, we find this step to ultimately be more streamlined. When you record a new macro, all of your keystrokes are recorded, including the ones that correct (seemingly inevitable) typos.

Using keyboard macros is even easier. When you want to insert code contained in a macro, simply type the relevant keystroke combination. Now, instead of copying and pasting the path for an external program

you want to execute or remembering the template you use for your program headers, you just have to remember ALT+I (for the %include statement) or ALT+H (for the header).

CONCLUSION

We hope that you found these tips helpful. Remember that there are many other tips and tricks to help your programming life. Experienced programmers have a wealth of information about strategies and different ways of accomplishing the same goals; don't forget to turn to your colleagues as resources. And share what you know, too!

REFERENCES

SAS Support Samples and SAS Notes. "Usage Note 33602: Displaying Additional SAS Log Messages with MSGLEVEL =." Accessed July 1, 2019. http://support.sas.com/kb/33/602.html [MSGLEVEL = i]

Base SAS 9.4 Procedures Guide, Seventh Edition. "WHERE ALSO." Accessed July 1, 2019. http://documentation.sas.com/?docsetId=proc&docsetTarget=p14b00ukl7f0qxn1ph7w2697j0wq.htm&docsetVersion=9.4&locale=en#p04cskv86hsr80n1hat386kh4fya [WHERE ALSO]

SAScommunity.org. "Where also." Accessed July 1, 2019. http://www.sascommunity.org/wiki/Where_also [WHERE ALSO]

Metzger, Kristina B and Pfeiffer, Melissa R. 2018. "Collaborations in SAS Programming; or: Playing Nicely with Others." *Proceedings of South Central SAS Users Group Conference 2018*. Austin, TX: SCSUG 2018. http://www.scsug.org/proceedings/2018-papers/ [Standardized Naming Conventions]

SAS 9.4 Language Reference: Concepts, Sixth Edition. "SAS Variable Lists." Accessed July 1, 2019. https://documentation.sas.com/?docsetId=lrcon&docsetTarget=p0wphcpsfgx6o7n1sjtqzizp1n39.htm&docsetVersion=9.4&locale=en
[Prefix Lists]

Pollack, Stuart. 2005. "Techniques for Effectively Selecting Groups of Variables." *Proceedings of SAS Users Group International 30*. Philadelphia, PA: SUGI 30. http://www2.sas.com/proceedings/sugi30/057-30.pdf [Prefix Lists]

SAS 9.2 Language Reference: Dictionary, Fourth Edition. "ARRAY Statement." Accessed July 1, 2019. https://documentation.sas.com/?docsetId=lestmtsref&docsetTarget=p08do6szetrxe2n136ush727sbuo.htm&docsetVersion=9.4&locale=en [Arrays]

SAS Technical Paper. "Using Arrays in SAS Programming." Accessed July 1, 2019. http://support.sas.com/resources/papers/97529_Using_Arrays_in_SAS_Programming.pdf [Arrays]

First, Steve and Schudrowitz, Teresa. 2005. "Arrays Made Easy: An Introduction to Arrays and Array Processing." *Proceedings of SAS Users Group International 30*. Philadelphia, PA: SUGI 30. http://www2.sas.com/proceedings/sugi30/242-30.pdf [Arrays]

Waller, Jennifer L. 2010. "How to use ARRAYs and DO Loops: Do I DO OVER or Do I DO i?" *Proceedings of SAS Global Forum 2010*. Seattle, WA: SASGF 2010. http://support.sas.com/resources/papers/proceedings10/158-2010.pdf [Do over]

SAS Support Samples and SAS Notes, Usage Note 22194. "How to use the CNTLOUT= and CNTLIN= options in PROC FORMAT to move formats from one platform to another." Accessed July 1, 2019. http://support.sas.com/kb/22/194.html [Formats in Excel]

Wright, Wendi L. 2007. "Creating a Format from Raw Data or a SAS Dataset." *Proceedings of SAS Global Forum 2007*. Orlando, FL: SASGF 2007. http://www2.sas.com/proceedings/forum2007/068-2007.pdf

[Formats in Excel]

SAS 9.4 Companion for Windows, Fifth Edition: "%INCLUDE Statement: Windows." Accessed July 1, 2019. http://support.sas.com/documentation/cdl/en/hostwin/69955/HTML/default/viewer.htm#chincludefstart.htm [%INCLUDE]

SAS 9.4 and SAS Viya 3.4 "Programming Documentation/Global Statements: %INCLUDE Statement." Accessed July 1, 2019. http://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=lestmtsglobal&docsetTarget=p1s3uhhqtscz2sn1otiatbovfn1t.htm&locale=en [%INCLUDE]

SAS Support Samples and SAS Notes, Usage Note 32774. "Creating keyboard macros and keyboard abbreviations in the SAS Enhanced Editor to reduce typing for repetitive code." Accessed July 1, 2019. http://support.sas.com/kb/32/774.html [Keyboard Macros]

Black, Steven. 2015. "Keyboard Macros – The most magical tool you may have never heard of – You will never program the same again (It's that amazing!)." *Proceedings of Pharmaceutical SAS Users Group 2015*. Orlando, FL: PharmaSUG 2015. https://www.pharmasug.org/proceedings/2015/QT/PharmaSUG-2015-QT26.pdf [Keyboard Macros]

ACKNOWLEDGMENTS

The authors wish to thank Meghan Carey and Fairuz Mohammed for their thoughtful review of this paper. Any mistakes herein are the responsibility of the authors.

CONTACT INFORMATION

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

Melissa R. Pfeiffer
PfeifferM@email.chop.edu

Kristina B. Metzger
metzgerk1@email.chop.edu