MWSUG 2016 - Paper TT12

Defensive Coding by Example: Kick the Tires, Pump the Brakes, Check Your Blind Spots, and Merge Ahead!

Nancy Brucken, inVentiv Health, Ann Arbor, MI
Donna E. Levy, inVentiv Health, Cary, NC

ABSTRACT

As SAS® programmers and statisticians, we rarely write programs that are run only once and then set aside. Instead, we are often asked to develop programs very early in a project, on immature data, following specifications that may be little more than a guess as to what the data is supposed to look like. These programs will then be run repeatedly on periodically updated data through the duration of the project. This paper offers strategies for not only making those programs more flexible, so they can handle some of the more commonly encountered variations in that data, but also for setting traps to identify unexpected data points that require further investigation. We will also touch upon some good programming practices that can benefit both the original programmer and others who might have to touch the code. In this paper, we will provide explicit examples of defensive coding that will aid in kicking the tires, pumping the brakes, checking your blind spots, and merging ahead for quality programming from the beginning.

INTRODUCTION

While there may be many ways to program, review, and check your work within the SAS environment, some of those approaches may be better, more efficient and robust to get the job done. Expecting the unexpected (defensive driving), defensive coding convention (vehicle maintenance), coding efficiency (good gas mileage), good programming concepts (checking your blind spot) and programming no-no's (running a red light) are concepts we will address. This paper will go through several examples, comparing various alternative routes leading to accurate and inaccurate destinations. Along the way, we will share several tips regarding techniques that we have found helpful over the years. Buckle up and enjoy the ride.

DEFENDING YOUR CODE AGAINST DATA (EXPECT THE UNEXPECTED)

Back when we were learning to drive, we were taught collision-avoidance strategies, such as waiting an additional second as well as looking both ways after the traffic light turns green before proceeding. This strategy is taught so as to avoid being hit by someone running a red light in the cross direction. Programming defensively is a similar concept. We want to write our programs in such a way as to avoid being flattened by oncoming data. The challenge in both cases is that we do not know exactly what will happen in the future. This is especially true when developing programs early in a project, when there is very little data to work with, or when what data is available, may be immature, incomplete, and/or incorrect.

In the heavily regulated pharmaceutical industry, programming work can begin with a detailed set of requirements, which describe not only what the final product is supposed to look like, but also the algorithms required to produce the data set. These requirements, the map to our final destination, include variable attributes and derivations, as well as analytical procedures and options, and serve both as instructions for the programmer and a description of what was actually done for the regulatory reviewers. Unfortunately, the requirements are written by humans and as such, are not always accurate. In addition, the requirements can evolve over time as unexpected data is encountered or clarifications are made. As such, programmers need to be critical thinkers as the program is being created, by asking questions, and actually looking at the raw and derived data. Merely serving as a typist, by copying and pasting code from the requirements without checking against the data pushes the errors further downstream. Unfortunately, the results are errors that are more difficult to catch, as well as more expensive to fix ***IF*** they are caught. Do not just follow the map. Instead, contribute to decisions being made as you travel toward your final destination.

Here is a simple example of defensive programming, relative to requirements. Suppose the instructions for programming a given variable state the following:

Set to 1 if RACE='WHITE'; set to 2 if RACE=' BLACK OR AFRICAN AMERICAN'; set to 3 if RACE='ASIAN'; set to 4 if RACE='AMERICAN INDIAN OR ALASKA NATIVE'; set to 5 if RACE='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER'.

However, suppose the data looks like this:

USUBJID	RACE
01-003	white
01-004	BLACK OR AFRICAN AMERICAN
01-005	OTHER

A program written only to the specifications would probably end up looking something like this:

```
data example1_0;
  set old;
  if race='WHITE' then racen = 1;
    else if race='BLACK OR AFRICAN AMERICAN' then racen = 2;
    else if race='ASIAN' then racen = 3;
    else if race='AMERICAN INDIAN OR ALASKAN NATIVE' then racen = 4;
    else if race='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER' then
      racen = 5;
run;
```

However, this code yields the following result, which is probably neither what is expected nor what is needed:

USUBJID	RACE	RACEN
01-003	white	
01-004	BLACK OR AFRICAN AMERICAN	2
01-005	OTHER	

One improvement is to make the comparison of the RACE value case-insensitive. Converting the text value to uppercase is more efficient when done once for each observation, such as what is done below, instead of repeatedly for each comparison. Once again, looking at the raw and derived data would help with finding these types of data issues and inconsistencies.

```
data example 1_1;
  set old;
  length racet $ 50;
  racet = upcase(race);
  if racet='WHITE' then racen = 1;
    else if racet='BLACK OR AFRICAN AMERICAN' then racen = 2;
    else if racet='ASIAN' then racen = 3;
    else if racet='AMERICAN INDIAN OR ALASKAN NATIVE' then racen = 4;
    else if racet='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER'
      then racen = 5;
run;
```

The above code was a good step in the right direction however, improvements can still be made. Specifically, we still have the problem of what to do with unexpected values that might appear in the future. One solution could be to merely add another ELSE condition. However, that will not alert us to the presence of unanticipated values in the data. Knowing if such values exist, and if these values are legitimate or need to be sent back for querying, is imperative for quality data and analysis. Here are some additional improvements:

```
data example 1_3;
  set old;
  length racet $ 50;
  racet = upcase(race);
  if racet='WHITE' then racen = 1;
    else if racet='BLACK OR AFRICAN AMERICAN' then racen = 2;
    else if racet='ASIAN' then racen = 3;
    else if racet='AMERICAN INDIAN OR ALASKAN NATIVE' then racen = 4;
    else if racet='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER' then
      racen = 5;
    else put 'WAR' 'NING: Unexpected RACE value- Patient ' usubjid ', race'
      race;
run;
```

Note that in the above code, 'WARNING' has been split into two segments. This strategy is utilized to avoid having the text string flagged as a log warning by automated log checkers or manual text searches.

In addition, on occasion variables may contain leading or trailing blank spaces. SAS has a simple function, TRIM that you can utilize to remove trailing blank spaces, and when combined with the LEFT function, will also left-justify a text string and remove any leading blanks. As such, here is another improvement on the code above:

```
data example 1_3;
  set old;
  length racet $ 50;
  racet = upcase(trim(left(race)));
  if racet='WHITE' then racen = 1;
    else if racet='BLACK OR AFRICAN AMERICAN' then racen = 2;
    else if racet='ASIAN' then racen = 3;
    else if racet='AMERICAN INDIAN OR ALASKAN NATIVE' then racen = 4;
    else if racet='NATIVE HAWAIIAN OR OTHER PACIFIC ISLANDER' then
      racen = 5;
    else put 'WAR' 'NING: Unexpected RACE value- Patient ' usubjid ', race'
      race;
run;
```

None of the updates took an excess amount of time to create. As such, small changes in your code can dramatically improve the quality of the output.

Here is a slightly more complicated example. The objective of the program is to set a baseline metabolic syndrome flag to 'Y' if three or more of the five component symptoms were present. Here is the way a programmer implemented this logic:

```
if (bmetsyn1="Y" and bmetsyn2="Y" and bmetsyn3="Y") then bmetflg= "Y";
if (bmetsyn1="Y" and bmetsyn2="Y" and bmetsyn4="Y") then bmetflg= "Y";
if (bmetsyn1="Y" and bmetsyn2="Y" and bmetsyn5="Y") then bmetflg= "Y";
if (bmetsyn1="Y" and bmetsyn3="Y" and bmetsyn4="Y") then bmetflg= "Y";
```

and so on….

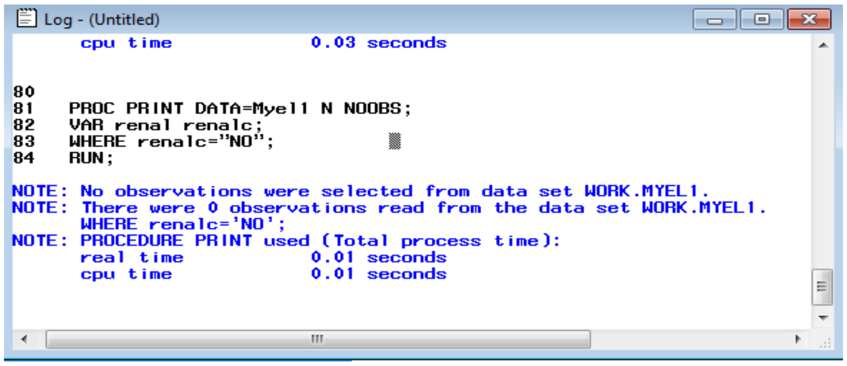
Coding in this manner forces SAS to evaluate each IF statement for every observation, rather than stopping as soon as a condition is true. Furthermore, the programmer only enumerated the actual combinations found in the data at the time the program was written. The next time the program was run against a newer version of the data, many cases that should have been flagged were unfortunately missed. In addition, there are 16 possible combinations, which is far too many for all to be listed in the code. A better way to write this section is the following:

```
if (bmetsyn1='Y') + (bmetsyn2='Y') + (bmetsyn3='Y') + (bmetsyn4='Y') +
  (bmetsyn5='Y') >= 3 then bmetflg = 'Y';
```

OUT OF RANGE VALUES

If you have mature data (or not) and you are filtering and get an empty data set, do not slam on the brakes. Perhaps there are no records that meet the criteria, but you should check the data and the code. Similar checks should be done for empty outputs. Be sure to proceed with caution. In the example below, the programmer got an empty data set. Can you identify the cause?

```
data example_2_1;
  length renalc $ 3.;
  set myel;
  if renal=0 then renalc="No";
    else if renal=1 then renalc="Yes";
run;

proc print data=example_2_1 n noobs;
  var renal renalc;
  where renalc="NO";
run;
```



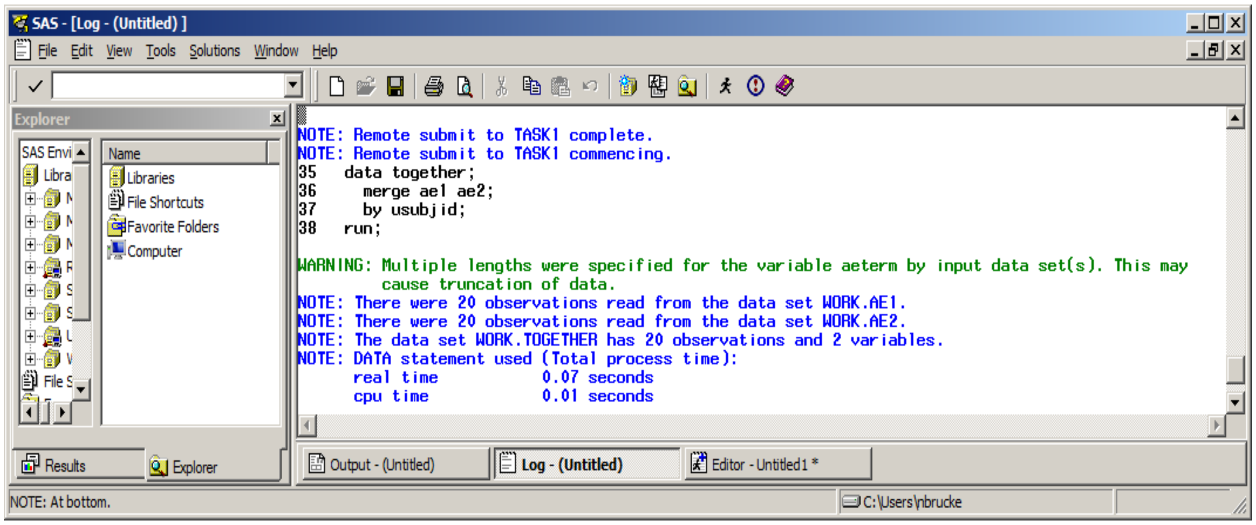
```
     cpu time            0.03 seconds


    PROC PRINT DATA=Myel1 N NOOBS;
    VAR renal renalc;
    WHERE renalc="NO";
    RUN;

NOTE: No observations were selected from data set WORK.MYEL1.
NOTE: There were 0 observations read from the data set WORK.MYEL1.
      WHERE renalc='NO';
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Display 1. Results from PROC PRINT step above

Here is a possible code improvement using the UPCASE option:

```
proc print data=example_2_1  n noobs;
  var renal renalc;
  where upcase(renalc)="NO";
run;
```

As you are coding your data set, check your log and record counts closely. Be sure to investigate any unexpected changes in counts --- both too many records and too few records. Be especially aware of record counts when you merge or join data sets where there are multiple records per patient. This situation could result in an unexpected many-to-many merge.



```
NOTE: Remote submit to TASK1 complete.
NOTE: Remote submit to TASK1 commencing.
    data together;
      merge ae1 ae2;
      by usubjid;
    run;

WARNING: Multiple lengths were specified for the variable aeterm by input data set(s). This may
         cause truncation of data.
NOTE: There were 20 observations read from the data set WORK.AE1.
NOTE: There were 20 observations read from the data set WORK.AE2.
NOTE: The data set WORK.TOGETHER has 20 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.07 seconds
      cpu time            0.01 seconds
```

Display 2. Multiple length specification warning message

Assuming that your data set is coded, knowing your data could simply decrease the chance of issues arising. We recommend always looking at the data to build an understanding. For example, you can run PROC FREQ on the entire raw data set before you start to get a feel for what types of values you will encounter. Nothing is more dangerous than developing a data set without understanding either the input data or the objective of what you are doing.

Adding a little defensive coding to your program will improve the quality of your code as well as the quality of your data set. Remember that even if a data set has been validated, that does not mean it is correct. For example, suppose there is a small issue like we see in the data set requirements above in example 2_1, where the requirements say to look for “NO” instead of “No”. The primary and validation programmer may program the exact same way and miss the issue in the requirements. You should perform a review of your output data set to ensure variables are not truncated or are populated as expected, and eliminate any obvious and not so obvious issues with the data set prior to the first review. In addition, for numeric variables with a known range, this step provides an easy check of whether the maximum and minimum make sense. For example, suppose you have a variable that you know cannot be negative, such as time-to-event values. Issues like this can quickly be identified with PROC FREQ or PROC UNIVARIATE, or a view of the data. Such a review can decrease the chances of downstream errors being found, or worse, not being found.

Aside from identifying values that are out of range for a variable, you should also look for missing values, and assess whether or not the missingness of specific variables make sense. In addition, if missing values are acceptable, does it make sense, given your knowledge of the source data, that a variable could be missing on every record in the data set? For example, does it make sense that 80% of the values for gender in a clinical study would be missing? Having that much missing data for a basic variable like gender is not likely at any stage of the study, even early on. If there is that much missing data for gender, there is probably an issue with the code. Further, if there is truly that much missing data, this information should be relayed to statistics and data management, as there appears to be an issue with the timeliness of the data being collected. A quick check of the data can improve the quality before a formal review, so that the focus can be made on more significant issues. Here is an example of some quick code that can be utilized (and turned into a utility macro) for ease of use throughout your program:

```
%macro checkdata (inds=);
  proc freq data=&inds;
    tables _all_;
  run;
%mend checkdata;
```

See Appendix I for a sample macro that will allow you to simply look at the range values for numeric variables in a data set (Appendix I).

HANDLING MISSING VALUES

No data set is perfect, regardless of project stage. As such, once again, you need to prepare for the worst in your initial coding. Key variables may not be populated and your code should take that into account. For example, in the code below, if DATE1 is not populated, a patient may be miscategorized by the following code:

```
if (date2 – date1 + 1) < 10 then catvar=”<10 days”;
  else if (date2 – date1 + 1) < 30 then catvar=”<30 days”;
```

When DATE1 is missing, (date2 – “missing” + 1) is also “missing. SAS considers missing values to be smaller than the smallest number that can be represented, and hence, less than 10. A small code improvement using the NMISS function will eliminate the miscategorized patients:

```
if nmiss(date1, date2)=1 then catvar=”Missing/Unknown”;
  else if nmiss(date1, date2)=0 then do;
    if (date2 – date1 + 1) < 10 then catvar=”<10 days”;
      else if (date2 – date1 + 1) < 30 then catvar=”<30 days”;
  end;
```

Once again, writing defensive code when data circumstances are unknown will make your code more robust. The more defensively you are driving/coding, the better the quality of your code and final results. Here is another example below:

```
data mydata;
  set olddata;
  if datvar < trt01date then phase= “Pre”;
    else if datvar >= trt01date then phase= “Post”;
run;
```

However, after running the code, here is what the data looks like:

Patient ID	TRT01DATE	DatVar	Phase
1	12DEC2000	01JAN2000	Pre
1	12DEC2000	01MAR2001	Post
1	12DEC2000	.	Pre
1	12DEC2000	17JUL2001	Post
2	.	21MAR2002	Post
2	.	01MAY2002	Post

As you can see above, there is missing data in the data set, which the code does not accommodate. Records are being put into “Pre” and “Post” categories when there is missing data. As such, as we have previously seen, more defensive code should look like this, in this case, using the MISSING function:

```
data mydata;
  set olddata;
  if not missing(trt01date) and not missing(datvar) then do;
    if datvar < trt01date then phase = ”Pre”;
      else if datvar >= trt01date then phase = ”Post”;
  end;
run;
```

Here is the output from the defensive coding above:

Patient ID	TRT01DATE	DatVar	Phase
1	12DEC2000	01JAN2000	Pre
1	12DEC2000	01MAR2001	Post
1	12DEC2000	.	
1	12DEC2000	17JUL2001	Post
2	.	21MAR2002	
2	.	01MAY2002	

The above output more accurately categorizes the data. When either DATVAR or TRT01DATE is missing, we cannot assess whether the date occurs before or after the first treatment date. Any time you are working with dates, you should take into account that data may be missing and incorporate that into your code. Furthermore, for any analysis, you should try and anticipate what the data and variables might look like. Particularly early on in the study, you cannot assume that the data is going to be of high quality. Even at the end of a study, there may also be issues with the data. You cannot anticipate everything, but you can defend against some of the more frequently encountered data issues.

DEFENDING YOUR CODE AGAINST CODE (VEHICLE MAINTENANCE)

As we all know, developing code can take many hours, days, or even weeks, and the development is not always linear. There are many curves, stop signs, hills, potholes and unfortunately U-turns. With so many stops and starts, we should develop code that is as robust and defensive as possible in anticipation for a possible rough, road ahead. In addition, the next person who modifies your code will probably not be as familiar with the study or the code in comparison to you. Thus, you should make the transition as easy as possible, and avoid creating potential potholes that could flatten their tires.

For example, when calling procedures, we recommend explicitly specifying the input data set even though SAS does not require the option. This way, if more code is added, or you change the order of your code, the procedure always will apply to the correct data set. An example of code is provided below:

```
proc means sd mean min max;
  var var1;
  output out=SummOut;
run;
```

versus:

```
proc means data=mydata sd mean min max;
  var var1;
  output out=SummOut;
run;
```

Also, within code comments are very helpful, both for someone else looking at your code, as well as when you have to make changes to a program that you may not have opened in months, or even years. Complicated sections of code should always contain comments explaining what is being done. All major sections of code should also be documented, so another programmer needing to make a change can easily hone in on the relevant steps via the comments, instead of having to carefully review every step.

CHECKPOINTS

As you progress through your program, another quality programming technique is to include code that you can use to check your program over and over. For example, suppose you are filtering patients with

VAR1=1 that have values of 1 and 2 only. By simply adding the code below, you can verify the data each time you run your program. Permanent program checks after critical steps are very important through the duration of the study. Checking code is important in the beginning of the study when not all the possible values have been reported. Checking code is also important later on in the study, to ensure that your early code is still accurate based on what the data now looks like. As we have mentioned previously, check code can also be used to ensure your underlying assumptions are correct.

Similarly, creating intermediate checkpoint data sets that you know are correct can help you track down problem cases in the final data set. Using the raw data checks will hopefully help minimize errors with filtering or typographical errors leading to the wrong subset. Here is an example of printing specific variables in raw tumor measurement data to check that the computed best overall response is correct:

```
proc print data=RawTumor;
  var patid visit dttum visresp diff comp_resp;
run;
```

Note that the variables visit, date of tumor assessment (DTTUM) and visit response (VISRESP) come from the raw data. The raw data is compared to the computed response (COMP_RESP) in the analysis data set.

Patient ID	**Visit (raw)**	**Date (raw)**	**Visit Response (raw)**	**Diff From Prev**	**Computed Overall Response (derived)**
1	1	01JAN2001	---	---	
1	2	01MAR2001	SD	59	
1	3	04MAY2001	PR	64	PR
1	4	17JUL2001	PR	74	
2	1	21MAR2002	--	---	
2	2	01MAY2002	SD	41	**PR**
2	3	26JUN2002	PD	56	
CR=Complete Response (1); PR=Partial Response (2); SD=Stable Disease (3); PD=Progressive Disease (4)					

Looking at the above raw data, the best response for patient 1 can be seen as PR and the best response for patient 2 as PD (assuming confirmation is needed). However, looking at patient 2, there may be an issue with the computed response of PR since PR was never reported in the visit response. As such, running the above code on raw data for a quick comparison to the derived response (last column) can be used to check the computed response data. Similar code can be created for other complex endpoints such as time to event and worst grade for an adverse event.

Finally, before you start programming, be sure you understand the endpoints of the study as well as the data. In addition, you should understand the variables that you are going to create and how they relate to each other. For example, suppose you have to create numeric and character variables from the same original variable. As the variables are related, programming these variables within the same section of seems to make sense. This will make debugging your code easier, and make those derivations easier to find for someone else who may eventually inherit your code.

EFFICIENT CODING (GOOD GAS MILEAGE)

Here is another case where writing independent IF-THEN statements can trip you up. Suppose you have the following requirements:

Set Y to 1 if x < 10; set Y to 2 if 10 <= x < 20.

And the code is written as follows:

```
if x < 10 then y = 1;
if x < 20 then y = 2;
```

What do you think will happen if X=5? Probably not what was intended based on the specifications provided above. Y will be assigned a value of 2, since the result of the second IF condition will overwrite the result of the first one. However, you will obtain the correct result with the following code:

```
if x < 10 then y = 1;
  else if x < 20 then y = 2;
```

Sure, there are many ways you can program, however some programming code will just get you there faster and more efficiently. The code may not necessarily be more complicated, just faster.

Suppose you have the following code:

```
if var1=”a” then var2 = “Alpha”;
if var1=”b” then var2 = “Beta”;
```

In this example, every line of code is executed for every record, regardless of whether the IF condition has already been met.

Now suppose our data looks like this

Obs	Var1
1	a
2	b

For the first observation, though the value of VAR1 falls into the first category (VAR1=”a”), SAS will still evaluate the second IF statement. A more efficient coding technique would be:

```
if var1=”a” then var2 = “Alpha”;
  else if var1=”b” then var2 = “Beta”;
```

In this example, the code will stop executing once an IF condition is met, and a category is found.

SELECT statements will also achieve the same efficiency, and are often easier to read, and thus maintain. The code above can also be written:

```
select (var1);
  when (”a”) var2 = “Alpha”;
  when (”b”) var2 = “Beta”;
  otherwise put “WAR””NING: Unexpected value of VAR1: “ var1;
end;
```

Again, the SELECT statement will stop executing once a condition is met. The OTHERWISE statement provides a convenient way to vacuum up any stray values that have fallen all the way through to the seat cushions. This example is yet another way to identify potentially incorrect data values. For those who prefer SQL, CASE statements accomplish the same purpose.

AUTOMATION (CRUISE CONTROL)

If you had the choice of automatic transmission or a stick shift, what would you choose? Automatic transmission might cost more up front, but may result in less effort and greater gas efficiency down the road. The same can apply to programmatic validation versus manual validation. Programmatic validation of a table, listing or analysis data set may take additional time up front in a project. However, if a program is going to be run for multiple deliverables over the course of a project, programmatic validation may save you time, energy and effort. Manual review of a lengthy table may take more time than the process is worth. In addition, the manual reviewer may unintentionally miss errors or issues. Let's be honest, a full manual review of a lengthy table can be difficult, particularly when multiple tables are being reviewed at the same time. Saving the data set going into your final report-generation step, or using the OUT= option on PROC REPORT, will allow an independent validator to validate the contents of the table with PROC COMPARE. This may require more time up front, as programmer and validator need to agree on the structure of the data set to be compared, but time may be saved in the long run particularly if a program is going to be run repeatedly over the duration of a study. Note that the automatic validation is not a replacement for a manual check of the table format or content. The programmatic validation just simplifies the process and hopefully decreases the overall validation time.

Often in a project, you utilize the same code to perform a similar task over and over. While the code could be written over and over again, ideally you should write the code once and utilize the same code over and over. As such, using a macro is one way to improve code like the following:

```
title1 ”Output 1: Var 1 by Var2”;
proc freq data=mydata;
  tables var1*var2 / fisher;
  exact;
run;

title1 ”Output 2: Var 2 by Var3”;
proc freq data=mydata;
  tables var2*var3 / fisher;
  exact;
run;

title1 ”Output 3: Var 3 by Var4”;
proc freq data=mydata;
  tables var3*var4 / fisher;
  exact;
run;
```

The above code has been best described as “wallpaper code”. The programmer is repeating the same code over and over; the repeating code forms nice patterns when printed, but takes a lot of effort to maintain. While programming, you should consider if a BY statement will suffice, if a macro is needed, or both. For the above code, a macro was utilized and the code was improved to the following:

```
%macro freqnum(inds=, num=, v1=, v2=);

  title1 ”Output &num: &v1 by &v2”;
  proc freq data=&inds;
    tables &v1*&v2 / fisher;   *** line A;
    exact;
  run;

%mend freqnum;

%freqnum(inds=mydata, num=1, v1=var1, v2=var2);
```

```
%freqnum(inds=mydata, num=2, v1=var2, v2=var3);
%freqnum(inds=mydata, num=3, v1=var3, v2=var4);
```

Using a macro decreases the amount of code you need to maintain. For example, suppose a different test statistic method is required. Using the macro, you only need to change the code in one place (line A above). Additional macro variables can be added, such as the data set name and the test statistic method, so that the macro is even more flexible.

In addition, by using macros to generate variations of unique tables, the same validation code can be used to check multiple displays. Similarly, the primary code can utilize macros to produce that output as well. Once again, time is saved for the delivery as well as for the project as a whole.

As you are developing your code, do not just go into autopilot. Think about what you are doing, and how to improve and develop your code as you go. Once again, a little care in the beginning will save you time in maintaining your code and avoiding careless errors in the long run.

KEEP ONLY WHAT YOU NEED (TOWING CAPACITY)

Sometimes data sets can be quite large relative to both the number of records and the number of variables. Manipulating data sets this large can slow your processing time down substantially. Once again, while you are developing your code, you should consider the path you want to travel, including the variables you want to keep and drop in order to improve the efficiency of your code. Limit the number of columns you read in and keep in your program to only the variables that you need. This is just like staying within the towing capacity of your car. Carrying more than the car can comfortably handle will really slow you down and use up a lot of extra fuel. Also, pay attention to whether your KEEP and DROP statements should apply to the input or output data sets, and make sure they are placed in the proper location, as placement can impact the efficiency of your code. Code is also easier to read and maintain if a consistent method of specifying KEEP and DROP statements is followed. For example, always placing them at the top of the step, or as an input/output data set option, makes the variables included in the data set easy to locate, track and debug.

GOOD PROGRAMMING CONCEPTS (CHECKING YOUR BLIND SPOTS)

Good programming concepts are the basic lessons learned in Driver's Education. At a minimum, you should be writing well-structured, well-documented programs that are easy to follow, and that someone else can maintain while you are on vacation. The last email or phone call you want to receive while you are relaxing on a tropical beach is your colleague asking how to fix one of your programs which just broke when run against the latest version of the database. This section provides additional guidelines for improving the quality, readability and maintainability of your SAS code.

Before you head out on a road trip, particularly a long one, you open a map and plan your trip (okay, think back before GPS). When you are programming, think about what you are doing before you start developing. Before you dive into code, think about how you are going to attack the problem before you shift into high gear, and make sure you understand what should be produced at each step. Think about which variables you are going to code first before you create them. This will help you plan which variables should be programmed together, as well as identify relationships between variables. Planning ahead also helps with program structure and readability.

The example code below relates to time-to-event data. As you are computing the time to event, deriving the censoring at the same time is more efficient, since they are related to each other. Note that, as every record will need the overall time computation, we will compute the value once, which is the very last step of the code below (line B):

```
*************** PFS primary *************;
if base=0 then do;
  adt = startdt;
  cnsr = 1;
  evntdesc = "No baseline assessment";
end;
```

```
   else if not missing(pddt) then do;
     adt = pddt;
     cnsr = 0;
     evntdesc = "Progression Disease.";
   end; ** pd, no sys;

   else if not missing(dtdead) then do;
     adt = dtdead;
     cnsr = 0;
     evntdesc = "PFS 1: Death.";
   end; ** death, no sys;

   else if missing(pddt) and missing(dtdead) and not missing(progfreedt) and
     progfreedt >= startdt then do;
       adt = progfreedt;
       cnsr = 1;
       evntdesc = "Last Objective Progression-Free Assessment.";
   end; ** no event, no sys;

   else if missing(progfreedt) or progfreedt < startdt then do;
       adt = dtrand;
       cnsr = 1;
       evntdesc = "No evaluable basln/post-basln disease assessment.";
   end;  ** no prog free assessment after randomization;

aval = adt – startdt + 1;  **** line B;
```

As you are voyaging on a road trip, everyone likes a little elbow room--- it just makes for a more pleasant trip. The same principle applies when you are programming. Spaces are free. Indenting is free. Both spaces and indenting make your code easier to read, for you, and for whoever may review or inherit your code. As you can see in the above code, there is good indentation to help with readability. In addition, as previously mentioned, related variables such as the end date, censoring variable, and description of the censoring reason, are derived together. Finally, there are general comments within each section of the code to help with understanding the code. Comments are imperative for you and for whoever inherits your code. Trust us. You will not remember why you created a variable, or even why you computed the variable that way. Comment. Comment. Comment. You will appreciate it later. So will the next person that may have to review or utilize your code.

While your program is important, you cannot forget about your log. Your log is like checking your blind spot when you are driving. While you can drive and change lanes without checking your blind spot, at some point you may have a close call or an accident because there is another car in the lane that you cannot see. Similarly, while there may be no issues or no significant problems in your log, a review of the log should be part of your regular routine. Searching for “MERGE WITH MULTIPLE”, “UNINITIALIZED”, “LENGTH VARIABLE”, “ZERO RECORDS”, “WARNING”, “NOTE”, “CONVERT”, “CONVERGE” and “ERROR” is a good starting point to catching any possible programming issues. If you take care of the log as you go, then if your program needs to be handed off to another programmer quickly, they should not have to deal with log issues produced by an unfamiliar program. If you think you have a good first draft of the program, then check the log. Do not wait until the end. You may think you are busy now; however you will be a lot busier right before a delivery. If you have an automatic log checker, use the output and review the file every time. Even a small change in the code can create a large problem in the log. The automatic log checker does not replace looking at the log and output, but once again, will save a lot of time.

Just as your license and registration should be located in one place in your vehicle, so should the directory paths that are related to the project. Since data for a project may be received multiple times and possibly in multiple locations, directory paths should be stored in one place, and defined via a macro or program that should be called by every other program in the project. This way, for every delivery, only the paths within the macro in one location need to be updated, instead of having to modify every program. This will ensure that every program (and every programmer) is pointing to the correct directory and the correct data for every single run.

Batch jobs also save time and are an efficient way to run all the programs with literally the press of a button. A batch job is like cruise control. If multiple deliveries of multiple programs are expected, and especially if the order in which they are run is important, then you can create a batch job to save time. The other advantage to running a batch job, as opposed to a driver program full of %INCLUDEs, is that the batch job clears out everything- all temporary data sets, macro variables, etc. The batch job also provides a brand-new SAS session for each program. As such, you do not have to worry about macro variables that have not been deleted or reinitialized, among other issues that may affect subsequent programs.

One would expect a newly permitted driver would become a better driver with experience. Similarly, as in any position that you hold, you want to continue learning and continue improving your skill set. As a SAS programmer or statistician, SAS workshops through your organization, sponsored by your local or regional SAS users group or through SAS itself (in person or online) are a good place to keep improving your programming skills. There are always better, different or improved methods to attack a problem. New features in SAS can make coding easier. In addition, SAS has many certification programs that will allow you to improve your general programming skills as well as specific skills related to clinical trials, data management, Business Intelligence, platform administration, advanced analytics among others. Keeping your skills up-to-date is like taking a defensive driving course to improve your driving skills. In addition, while taking defensive driving also decreases your insurance cost, improving your programming skills will enable you to do more SAS programming more efficiently and effectively. Revving up your resume never hurts either.

PROGRAMMING NO-NO'S (RUNNING A RED LIGHT)

In every position, there are always aspects of the job that are considered unacceptable. The same can be true for SAS programming. As we have mentioned previously, programs devoid of comments, white space, and indentation are nearly impossible to read, and often difficult to understand. Some coding techniques can also make it difficult to trace a path from your input data to your output data sets, or make debugging harder than it should be.

Programming should be based on the data that is stored in the database. As such, hardcoding for a specific patient to override what is in the database (or what is not in the database) is a no-no. This technique is a violation of good programming practice as described by many regulatory, governing and collaborative bodies (PhUSE, 2014). If there is a value that is obviously incorrect, data management should be informed and the database should be updated. For those in the pharmaceutical industry, this ties in with the CDISC (Clinical Data Interchange Standards Consortium) concept of traceability from analysis results back to source data. Be sure to communicate database issues with programmers, statisticians as well as data management. You might see the issue, but the other team members may only find out about the problem through your communication.

Using someone else's identification for any reason is a big no-no. Similarly, using the same name for intermediate data sets throughout your program may appear to be efficient on the surface, however there will likely be downstream consequences. Unfortunately, when you are trying to debug or find errors in your code, identifying the issue becomes very challenging when all of the temporary data sets carry the same name. As such, having a consistent data set naming convention for all your programs would be beneficial. One way to implement this approach is the following:

```
data aevents2;
  set aevents;
  ae = substr(var1);
  num + 1;
run;
```

```
data aevents3;
  merge aevents2 demog;
  by subjid;
run;
```

Uniquely naming the data sets as we have above would help with debugging and improve your overall programming practice. However, what happens if you subsequently need to insert additional programming steps? Choosing meaningful names for your work data sets is an even better solution. Simple, well thought out changes to your code can significantly improve the readability:

```
data aeventct;
  set aevents;
  ae = substr(var1);
  num + 1;
run;

data aedemog;
  merge aeventct demog;
  by subjid;
run;
```

Be careful when merging data sets, especially when they share variables in common. If those variables are not being used as merge keys, the hitch-hiking values coming in from the second data set specified on the MERGE statement will overwrite those coming in from the first data set. If a merge is ahead, once again be aware of the data as well as the variables within each data set. As we indicated above, the KEEP and DROP statements come in handy.

Logical development of your program is also of benefit to you as well as others that may utilize your code. For example, if you keep the sections reading from your input data sets at the beginning of the program, followed by all of your derivations, and finish with your output step, debugging or finding a raw or derived variable can be done with greater ease. Additionally, try to create variables only once in a program. Re-using utility variables, such as loop counters, probably will not cause difficulties when a program is later modified, If however, one of your output variables is programmed and later overwritten or re-derived, one of those code sections may be missed or overlooked the next time an update is made. While starting with a plan as you begin your program is important, continuing with the plan through the duration of the program so your code remains readable and organized is also imperative, no matter how busy you are.

CONCLUSIONS

We hope this paper has given you some ideas of better programming techniques and tools that you can do to make your programming journeys more efficient, and minimize the impact of bumpy database updates. Here are some of the main considerations for the next time you write a program:

- Spaces and tabs are free, and make your code easier to read.
- Think about what you are going to do before you do it. Think about which variables should be coded together before you do it. Before you program a step, know exactly what to expect coming out.
- Trust us. You will not remember why you created a variable or even why you computed it the way you did. Comment. Comment. Comment. You will appreciate it later. So will the next person who may have to review your code.
- Your code is a representation of you. Your code is also a reflection of who you work for as well as all your colleagues. It is a representation of ALL of us. Take pride in your code.

Keep in mind when you are programming that you are not just programming for yourself. You are also programming so that others can understand your logic within your program. In addition, you will likely appreciate your quality code six months later when you cannot recall details of the code or the project. You will thank yourself for quality programming, as will others who utilize and review your code. So go ahead. Give your SAS code a nice shiny coat of wax and be sure to check under the hood before you head off on your next road trip. Your passengers and drivers will thank you.

REFERENCES

Pharmaceutical User Software Exchange [PhUSE] (2014). Good Programming Practice. Retrieved from http://www.phusewiki.org/wiki/index.php?title=Good_Programming_Practice.

ACKNOWLEDGMENTS

We would like to thank all of the programmers who unintentionally contributed the examples for this paper, and our colleagues for reviewing and providing feedback. Also thanks to Paul Slagle, Jeannette Day and Daniel Quinn for their careful review and comments.

RECOMMENDED READING

Carpenter, A.L. (2011). *Job Security: Using the SAS® Macro Language to Full Advantage.* Retrieved from http://www.lexjansen.com/pharmasug/2005/technicaltechniques/tt05.pdf.

Cody, R. (2011). *SAS® Statistics by Example*. Cary, NC: SAS Publishing.

Delwiche, L.D. and Slaughter, S.J. (2003). *The Little SAS® Book: A Primer* (3rd ed.). Cary, NC: SAS Publishing.

Hunt, A. and Thomas, D. (1999). The Pragmatic Programmer: From Journeyman to Master. Reading, MA: Addison-Wesley.

SAS® Certification Prep Guide: Advanced Programming for SAS9 (2007). Cary, NC: SAS Publishing. Cary, NC: SAS Publishing.

SAS® Certification Prep Guide: Base Programming for SAS9® (2nd ed.) (2009). Cary, NC: SAS Publishing.

CONTACT INFORMATION

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

Nancy Brucken
inVentiv Health
Ann Arbor, MI 48108
734-887-0255
Nancy.Brucken@inventivhealth.com

Donna Levy
inVentiv Health
Cary, NC 27513
Donna.Levy@inventivhealth.com

APPENDIX I – MINIMUM AND MAXIMUM MACRO FOR NUMERIC VARIABLES

```
%macro dset_min_max(indata=, outdata=);

  %* “indata” is the name of the input data set.                  ;
  %* “outdata” is the name of the output data set which includes the ;
  %* variable name, minimum, maximum, and number of missing values   ;
  %* for each numeric variable in the input data set.               ;

  %* Create an empty output data set to hold the results;
  proc sql;
    create table &outdata (name char(30), vmin num, vmax num, vmiss num);
  quit;

  %* Open the input data set and iterate through the variables;
  %local dsid rc j varname;
  %let dsid=%sysfunc(open(&indata));
  %do j=1 %to %sysfunc(attrn(&dsid, nvars));
   %* If the variable is numeric then insert the appropriate values ;
   %* into the output data set.                                     ;
    %if "%sysfunc(vartype(&dsid, &j))"="N" %then %do;
      %let varname=%sysfunc(varname(&dsid, &j));
      proc sql;
        insert into &outdata(name, vmin, vmax, vmiss)
          select "&varname", min(&varname), max(&varname),
                 sum(case when &varname is missing then 1 else 0 end)
          from &indata;
      quit;
    %end;
  %end;
  %* Close the input data set;
  %let rc=%sysfunc(close(&dsid));

%mend dset_min_max;
```

/* Demonstration of the macro */

```
data work.test;
  myvar1='hello';
  myvar2=1;
  myvar3=1000;
  output;
  myvar1='there';
  myvar2=2;
  myvar3=.;
  output;
run;

%dset_min_max(indata=work.test, outdata=work.out);

proc print data=work.out;
run;
```