PharmaSUG 2021 - Paper AP-195

How To Ingratiate Yourself With The "Old-timers"

Kirsty Lauderdale, CSG an IQVIA business;
Kjersten Offenbecker, GSK

ABSTRACT

Ever doubted that your program code would stand up to the review of some of the great* programmers, ever wondered if your logic process makes sense and that your programming style would pass the "white-glove" test. This paper will help guide you through some of the steps that long-time programmers use to ensure their program codes are readable, transferrable, and robust.

Start your journey to becoming a well-known name in the industry by following some of these handy tips and tricks, and instantly elevate your programming code to previous unattained levels.

* Definition of great may be self-proclaimed.

INTRODUCTION

We've all been there – 5pm on the day of a deliverable picking up another programmer's code for an "easy" tweak, and as you open the program code and start to scroll, your heart absolutely sinks……. *x* years of experience kicks in and you know this is not a fast fix – this is going to take hours of deciphering or a re-write.

It happens to us all at some point in our career, we have picked up "bad code", but what do we do; we grumble, we sigh and we either re-write or clean it up and try to make it better for the next programmer.

The thing is – that "bad code", it is probably not all that bad and had the programmer followed a few rules, it very well may have been that quick easy modification that we originally thought was needed.

This paper will offer some things that should be used daily in your programming, and by implementing these you will see more clarity in your program flow.

K.I.S.S (KEEP IT SIMPLE…)

When we first started programming, there was this wonderful notion going around to have programming be as complicated as possible to ensure job security…. that has never been our style. We believe that programming should be transferable, easy to read and easy to understand, and a sign of great coding is that anyone can pick up and know exactly what your code is doing every step of the way.

STYLISTIC / LAYOUT OF PROGRAMMING

Certain people have a very specific way of programming – an instantly recognizable style – as easy to assign to a person as if you were looking at their handwritten notes. Sometimes your style is developed from years of following a specific company standard, other times its from programming in a different language and having those habits carry forward, sometimes it is a throw-back to an older version of SAS. Regardless of your programmatic habits, you can make some changes that will enhance your code read-ability, and in-turn elevate other programmers' perception of your programs.

The layout of program code can tell you a lot about a programmer; a well laid out logical process flow, with line breaks, indentation and good solid code sectioning instantly makes us feel that a seasoned programmer has generated this. It also gives us confidence that we can debug the program quickly and efficiently – rather than making an ugly hack at the end of the program.

Capitals (and everything in-between)!

A hard and fast rule with respect to capitalization would ultimately delight and upset people at the same time; and is something we would look to avoid! Some programmers program all in capitals, some all in lower case, some initials and some mixed – just wait until you see our examples!

In this paper, we are not saying everyone's code needs to be identical and we understand that specific coding habits can be hard to overcome. Think about some code you have reviewed recently – what was the easiest to read – was it all uppercase, all lower, keywords highlighted?

The point is this is a minor item – SAS has some helpful commands that can instantly transform a full program to different capitalization, if that is where you want to draw your line!

Original

```
pRoc SORT data=test;
   BY usUBJid;
Run;
```

Ctrl Shift + u

```
PROC SORT DATA=TEST;
   BY USUBJID;
RUN;
```

Ctrl Shift + l

```
proc sort data=test;
   by usubjid;
run;
```

Blank space is your friend!

Given our experience reviewing and debugging other people's programs, the use of return lines and indentations are seriously under-utilized!

Every single data step or PROC SQL code should be separated by a blank line above and one underneath; this is simply following Good Programming Practices** (this is also not just relevant to coding, this should be the norm in all your writing practices, but we digress).

Every line of executable code should have its own line.

Consistent indentation is also nice and makes code much easier to read. In general, everything in between your DATA and RUN, PROC and RUN, PROC and QUIT, or MACRO and MEND should be indented. If you are using nested code, related lines that wrap or nested macros, indent those so they are very quickly discerned as related chunks.

Which of the below is faster to decipher?

```
DATA one (KEEP=var var2 var3 var4);
SET data_a (RENAME=(var1=var1a var2=var2a var3=var3a)) data_b (RENAME=(var1=var1a var2=var2a
var3=var3a)) data_ba (WHERE=(age=30));
BY usubjid;
RUN;
```

```
DATA one (KEEP=var var2 var3 var4);
   SET data_a (RENAME=(var1=var1a var2=var2a var3=var3a))
       data_b (RENAME=(var1=var1a var2=var2a var3=var3a))
       data_ba (WHERE=(age=30));
   BY usubjid;
RUN;
```

SAS has some very handy keyboard shortcuts that can be set to automatically indent an exact number of spaces, and should you pick up code that is extremely difficult to read SAS can even reformat it for you if you use the CTRL + I option.

**If you are not familiar with Good Programming Practices (G.P.P), we would recommend giving it a google.

Finish your SAS Statements!

Every single data step or SQL step should complete with either a RUN or a QUIT statement, followed by a semi-colon. The semi-colon should ideally be right next to the keyword.

When it comes to debugging the code, it is much easier to follow a step to completion if the individual step has an end. It is tempting to leave off that last RUN statement, add a PROC SORT without having to explicitly call a dataset name – but for the sake of the programmers picking up your code, please complete the step fully. Future releases of SAS are not guaranteed to treat items the same way, and if your code ends up in a legacy study, it is potentially around for 10+ years – will it hold up to SAS implementation/structural changes?

Independently executable

```
DATA one;
   SET zero;
   BY usubjid;
RUN;

PROC SORT DATA=one;
   BY usubjid age;
RUN;
```

Not so much…

```
DATA one;
  SET zero;
  BY usubjid;
  PROC SORT;
  BY usubjid age;
RUN;
```

What's it all about!

When it comes to producing datasets and outputs, understanding what you are programming and why is key. Programmers need the ability to document what they are doing, and why. Sometimes we program end to end and the project goes out the door, other times we work on several projects before coming back to one for data refresh re-runs. Think how many times you have picked up your own code and went "hmmm – why did I put that in….", it is far worse picking up someone else's code and you cannot fathom the logic behind a derivation that does not seem to match the specifications, however the code had previously passed validation.

There are two great ways within the program code that we can pass on pertinent information; using the program header (or a similar info header) and making programmatic notes in comments throughout the code. Good use of both will have the eternal gratitude from other programmers and save you headaches down the line.

Program Headers/Details

Every company has its own standards when it comes to program headers and you need to follow those guidelines. Although you should follow the company standard guidelines, some companies collect the bare minimum, and that can mean that the documented information is not really useful from a programmatic perspective. Luckily, there are always ways to add information to your program while still following your SOPs.

If you are using a versioning system that checks in and out your program code, archives deliverables, and maintains a separate audit trail history, or you have an amazing project tracker that keeps notes of every programmer who has touched that program – then a minimal project header of program name and creation date might be fine.

Most companies, however, do not have versioning systems. This means that program header details are completed by the programming team and depending on workload and attention to detail, the content can vary dramatically. All too often headers are spotty at best, details are not completed or accurate, and modification history is not kept up to date.

At the end of a study, before archiving or programs are to be sent to client, the lead programmer or designee should review all program code for accuracy, compliance to specifications and programming guidelines, and those reviews include header checks.

Anyone who has ever undertaken this task will confirm - these are very time-consuming reviews, and typically multiple programs are flagged for updates. Sometimes it is a simple quick change, the header program name needs modified, or study project code needs updated, other times you find there has been an accidental cross-contamination between your production and validation team….

Proper documentation at the time of program creation and modification can minimize the amount of rework, and hopefully mean no Note to File document creation.

```
************************************************************************
* Client      :                                                        *
* Project     :                                                        *
* Program Name:                                                        *
* Purpose     :                                                        *
* Author      :                                                        *
* Date        :                                                        *
************************************************************************
* Modification History - include author, date and change notes         *
************************************************************************
* Modification:                                                        *
*                                                                      *
*                                                                      *
************************************************************************;
```

```
*************************************************************************************************************;
*                                    Overview (2021)                                                        *;
*************************************************************************************************************;
* DSUR                                                                                                       *;
* Studies Included     : 5                                                                                   *;
* Completed studies    : AAA-BBB, XYZ-AAA                                                                    *;
* Completed during cut : AAA-AAA                                                                             *;
* Ongoing Studies      : AAA-AAB, xAAAAAC                                                                    *;
* Input Datasets       : xAAAAAA.adsl                                                                        *;
*                      : xAAAAAB.dm, xAAAAAB.ex, xAAAAAB.visdt                                               *;
*                      : xAAAAAC.dm, xAAAAAC.ex, xAAAAAC.visdt, xAAAAAC.studcomp                             *;
*************************************************************************************************************;
*************************************************************************************************************;
* Program Modification History                                                                               *;
*************************************************************************************************************;
*  DATE9.  KL   Copied and modified from xxxxxxxxxxxxxxxxxxxxxx                                               *;
*************************************************************************************************************;
```

Commenting code

There are several different ways in SAS to add comments to program code – a review of our programming codes tends to show a switch between two main styles of code, one is used to add descriptors and one to comment out chunks of code for testing, for future use, or for deletion post updates.

Adding comments to your code adds value to your overall program and individual data steps – you do not need massive in-depth comments on every step or derivation, but here is a very nice list of why you should comment code.

1. Show logical process flow
2. Explain derivations
3. Give a SAP page reference for additional detail

4. Make a note of a specific statistician or client request and note reference emails (for filing)
5. Explain data checks on unclean data
6. Document temporary code until data is fixed with a note for removal
7. Modification notes to link back to program header

```
* document a derivation here *;
* create new var ABLFL - see SAP px for additional info *;

/* Perhaps use cntrl + / to automatically comment wrap text blocks */
/* or use cntrl shift + / to uncomment block of text */
```

As always if you comment out chunks of code to test programs, and that code will not be used later, please remove those commented out sections from your code.

EFFICIENT CODING PRACTICES

Programming efficiently is more than simply using specific SAS statements, it is understanding that you can create efficiencies through programmatic changes, thoughtful code placement, understanding the data analysis, re-using previously developed code, making use of company macros, and recognizing how all this comes together.

There are many simple tips, tricks and techniques that can help you create robust and efficient programs, and we will discuss some of these below.

SUBSET EARLY AND KEEP ONLY NEEDED VARIABLES

In our industry, we read in a variety of datasets, in various formats, some large, some small. One of the biggest time savers when reading in data comes from sub-setting the data early and only keeping the variables that will be needed in or used to derive your analysis.

One of the first steps we do when reviewing code is to look at the data sources – if you are creating a dataset based on a specific subset of data or if you only need 20 variables versus the 80 variables in the source data, you should immediately apply those restrictions.

There is an immediate time saving solely from reading in less data, and the streamlining of the variables is a huge factor for helping quickly understand your program code.

The first time any dataset is called in the code - use the KEEP option for variables, and WHERE for sub-setting - this is the most efficient use of the PDV, and more informative to anyone who subsequently picks up your code. Make sure you use the statements on the main dataset options rather than within the datastep.

If you read in data, and immediately sort it in the next step, perhaps amend your first step to be sorting and reading in the data at the same time!

```
PROC SORT DATA=derived.adsl (WHERE=(saffl="Y") KEEP=usubjid age saffl)
          OUT=adsl;
RUN;
```

```
DATA adsl;
   SET derived.adsl;
   IF saffl='Y';
   KEEP usubjid age saffl;
   PROC SORT;
   BY usubjid;
RUN;
```

The above code is far more efficient than the code to the left.

Use a proc sort whenever you read in any data source, it is also very useful to sort all datasets before you merge them, even if your data is in the correct order now, it does not mean it will remain so for your next transfer. Also remember if you are using PROC SQL you do not need a separate SORT step.

NAMING CONVENTIONS FOR DATASETS

Debugging program code can be a long and arduous task – it is made a million times harder when the original programmer has re-used the same dataset name multiple times. Datasets are overwritten time and time again and finding which code section needs modification is a slow process of tracing data. It is especially painful if those datasets are in a macro, or randomly populated throughout the program code.

Good programming practices suggest using a distinct dataset name that is also somewhat descriptive. A good rule of thumb is to create sequential meaningful dataset names, for example a subset of ADSL may be named “adsl_1”, “adsl_2”, etc. Remember your code is public, so use sensible judgement when picking names!

Also, if naming datasets within macros that use macro variable input, try to follow a standard convention, or use a comment to explain, deciphering “&&P&i_&site&loc._3” into a variable name takes some time!

Always check with your department SOPs and working guidelines to ensure you are following company standards.

```
DATA advs (KEEP=usubjid saffl adt weight height trtsdt ady);
   SET derived.advs;
   ady=(trtsdt-adt)+1;
   PROC SORT;
   BY usubjid;
RUN;

DATA advs;
   MERGE advs
         derived.adsl (KEEP=usubjid age);
   BY usubjid;
RUN;

DATA advs;
   SET advs;
   .....;
RUN;
```

```
PROC SORT DATA=derived.advs (KEEP=usubjid saffl adt weight height trtsdt)
     OUT=advs;
  BY usubjid;
RUN;

DATA advs_1;
   MERGE advs
         derived.adsl (KEEP=usubjid age);
   BY usubjid;
   ady=(trtsdt-adt)+1;
RUN;

DATA advs_2;
   SET advs_1;
   .....;
RUN;
```

More efficient and better traceability

CREATING AND NAMING VARIABLES

Similar to naming datasets, some companies have specific guidelines for naming variables, you also need to take into account SAS and CDISC guidelines.

When you create a new variable in your code, please make a note of it in your comments – unless 100% self-explanatory, define it and explain the derivation. Depending on the complexity level, you may want to note an SAP reference, specific stat or lead guidance, or document anything that might help another programmer understand the specificities. Within the step where the variable is created, use an ATTRIB statement and define all the related items to that variable.

```
DATA adsl_ae;
   MERGE adsl (IN=A) ae (IN=B KEEP=usubjid);
   BY usubjid;
   IF A;

   * Create experienced AE flag for use in later derivations *;
   IF B THEN ae_flg=1;

   ATTRIB ae_flg LABEL="Pt AE Yes Flag" LENGTH=8. FORMAT=yn.;
RUN;
```

Any variables that are no longer required after the derivation is completed, DROP from the code.

USE DEFENSIVE CODING PRACTICES

Typically, when programming starts on a study, we have limited data and usually an upcoming initial draft deliverable. All too often a programmer will program exactly to the specifications, however data has a nasty habit of not doing what we expect it to…. The CRF and specifications can capture the normative expected data response, but a typo in data entry, different capitalization in the data, or an “interesting” subject can cause untold issues in our program codes if we do not account for the unexpected.

Say your CRF shows a list of 2 options for a particular data point – do you code for those 2 options or anticipate that perhaps something else may arise.

```
DATA adsl_1;
   SET adsl;
   BY usubjid;

   IF sex="Female" THEN sexn=1;
    ELSE IF sex="Male" THEN sexn=2;
RUN;
```

```
DATA adsl_1;
   SET adsl;
   BY usubjid;

   SELECT(sex);
      WHEN ("Female") sexn=1;
      WHEN ("Male") sexn=1;
      OTHERWISE PUT "WAR" "NING" Unexpected value of sex;
   END;
RUN;
```

Using SELECT over IF is more efficient in terms of resource processing, as when a condition hits true in a SELECT statement, it does not continue looping through other SELECT options. For PROC SQL coders, look into using CASE.

Many programmers add in log notes, using “ERR” “OR” or “WAR” “NING” for unexpected values, this is very useful during the draft versions of your deliverables as it helps monitor for data issues that should be raised and cleaned. Due to the space used in “ERR” “OR” or “WAR” “NING” , these code lines will not appear in your log checks.

There are several amazing SAS papers on defensive coding, and we would recommend spending some time reading up on these.

MACRO VARIABLES/CALLS

When it comes to macros, ultimately, they need to serve a specific function. There are two types of macros defined in SAS – GLOBAL and LOCAL – both serve different purposes within a project's need.

Two of the main points with macro use, that we would like to highlight.

1. Only use macros when needed – do not overly complicate your code; no-one needs a macro for 3 lines of stand-alone code.
2. Use keyword parameters whenever you create a macro and ensure these are well defined and documented.

This paper is not focused specifically on macros, and as such if you are looking for more information on global or local macros, we would recommend looking at some past conference papers on that subject.

```
*******************************************;
* Macro to create age from usubjid D.O.B   *;
*******************************************;
* Params: dsnin, dsnout, date              *;
*   dsnin : source dataset                 *;
*   dsnout: source dataset                 *;
*   date  : numeric date used for age calc *;
*******************************************;

%MACRO age (dsnin=, dsnout=, date=);
   DATA &dsnout.;
      SET &dsnin.;

      IF birthdt NE . AND &date NE . THEN age=INT((birthdt - &date.)/365.25));
   RUN;
%MEND;
```

Global

A good sign of any biometric centric organization is a well-defined and organized SAS® global macro library, with documentation on what is available and how to use it.

SAS defines global macros variables as "variables that are available during the entire execution of the SAS session or job". Global macros are used to help create consistent standard outputs, and gain efficiencies with regards to reusability and program set-up.

When calling these macros in your code – please ensure that you define the variables used in a comment above the macro call and add a note to the macro intent.

Local

Local macros are those created within a project – these may be stored within the individual project directory, or inside programming code.

If you opt to have macros inside programming code, please define the macros at the top of your code, and ensure you use keyword parameters in your set-up, and an explanation on what is used for. If you created nested macros, please ensure you follow proper indentation rules, to ensure code readability.

TIDY UP AFTER YOURSELF!

We all comment out chunks of code when debugging programs or make client comment modifications. That is a completely normal part of the process as we iron out unseen issues in the code and data. After the item passes validation however there is some hesitancy in removing the old redundant code.

This appears to come from that innate fear that you may at some unknown future timepoint be required to revert to that coding, but chunks of commented out code can cause problems, especially if you tend to put multiple executable statements on one line, and from a readability perspective it can make a program appear quite disjointed.

Please consider keeping an archived copy of the code, this way you can removed the commented-out code from your program, however you do have a back-up should you need to revert to a previous version. If your company uses a versioning system, there is no reason commented out code should be kept.

CONCLUSION

Inheriting code or modifying other programmers code is becoming the norm across our industry, if you have not experienced it yet, you most likely will at some point in your career. Given that this is becoming more and more common; why is it something that most programmers instinctively cringe over?

The problem is that all too often, its time-consuming and painful to comb through programs line by line, deciphering un-commented code with inexplicable variable names, and disjointed logic flow.

It doesn't have to be though, by following a few simple guidelines you can raise your programmatic game – create programs that flow beautifully, are readable, descriptive, transferable, and structurally sound. Code that produces clean logs, won't collapse upon a new data refresh, and when transferred can be picked up and modified by any level of programmer.

For some, the things we have noted are the basic constructs of their code, but if we all look at our code critically there is always room for improvement, some simple updates will change how your code is received and reviewed by your peers.

When it boils down to it; we know we can do better, we are all in this together! Be kind to your fellow programmer, and bear in mind that the steps you take today in your program code, can help out your entire team in the future.

RECOMMENDED READING

- SAS® 9.4 and SAS® Viya® 3.5 Programming Documentation

  (*Available at:* SAS Help Center: Programming Documentation for SAS 9.4 and SAS Viya 3.5)

- Good Programming Practices

  (Available at: Good Programming Practice Guidance - WELCOME - PHUSE Advance Hub)

- "Defensive Coding by Example: Kick the Tires, Pump the Breaks, Check Your Blind Spots, and Merge Ahead!" (Available at: Defensive Coding by Example: Kick the Tires, Pump the Breaks, Check Your (lexjansen.com))

CONTACT INFORMATION

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

Kirsty Lauderdale
CSG an IQVIA business
klauderdale@csg-inc.com

Kjersten Offenbecker
GSK
kjersten.x.offenbecker@gsk.com