SESUG Paper 135-2023

Coding for the Future: Smart Commenting

Brooke Ellen Delgoffe, Marshfield Clinic Health System, Marshfield, WI

ABSTRACT

We have no idea what is coming to SAS programming in the future, but most of us have worked with multi-author, multi- time-period, conglomerate code written over multiple versions of SAS. How do we bridge the gap between versions, time, and programmers? Properly commented code makes renovating, adding to, or reviewing code much easier. It is one of the only ways to talk to the future and make it easier for the next person, next version of SAS, or next version of you to pick up where you left off.

There's an art to commenting in a way that is meaningful. While keeping commented out code may provide insight into what was already tried, it may also clutter the program and leave the next programmer wondering why it was commented out. Whether it was a future area of development, an attempt that didn't work, or something else, the next viewer won't know unless you tell them. In the same way, commenting what was done could be helpful to novice programmers who may pick up complex code later, but true code renovation requires a full view of why something was done in addition to what. Single option methods used today could be replaced by a more efficient method or accompanied by many new solutions in later years.

INTRODUCTION

In a scene from Pixar's Ratatouille, a rat is asked where he is going and cordially responds, "..with luck, forward", and so too is the trajectory of the programming world. The SAS language changes quickly and its uses and applications grow exponentially as the language evolves. While there may be no way to do something today, you can almost promise there will be a way in the future. In a similar way, each programmer learns SAS in their own way based on what's available at the time, the specific uses they have for it, and with the help of others that have their own unique backgrounds. Since we don't know what will be available in the future, or even what our future selves will know, it's imperative that we communicate effectively through commenting.

Commenting allows us to speak to future readers and communicate problems, workarounds, functions, and goals of the code. By providing these insights we take the guess work out of the equation and allow for efficient updating and additions. By standardizing these comments, we allow for programmatic updating, reporting, and indexing. You can communicate hierarchy and metadata in addition to the text by using comment styling. Utilizing HTML-like tags allows for automatic application of metadata (like author).

This paper explores how, what, and why to comment. The topics covered in this paper are meant to be informative, but do take on opinion-based guidance at times.

COMMENTING DOS AND DON'TS

WHO IS HARRY? A CONTINGENCY PLAN

When you're writing code keep in mind your code's next viewer. Code that isn't intended to be carried into the future or run on a reoccurring basis could still be viewed again. Even if the next viewer is you, the addition of helpful comments is crucial.

Here's a few tips that will make your code more helpful for the next viewer:

- **Stick to the Author Constants:** Focus on programmer attributes that will not change for a programmer, like name, or can be inherited by the next programmer like department and title. Attributes like phone extension or email may change or follow the person into different roles, so these should be avoided.
- **Location, Location!** Embedding the location of documentation for the code, version control, change history, or previous/new versions is extremely helpful. Be careful about putting network

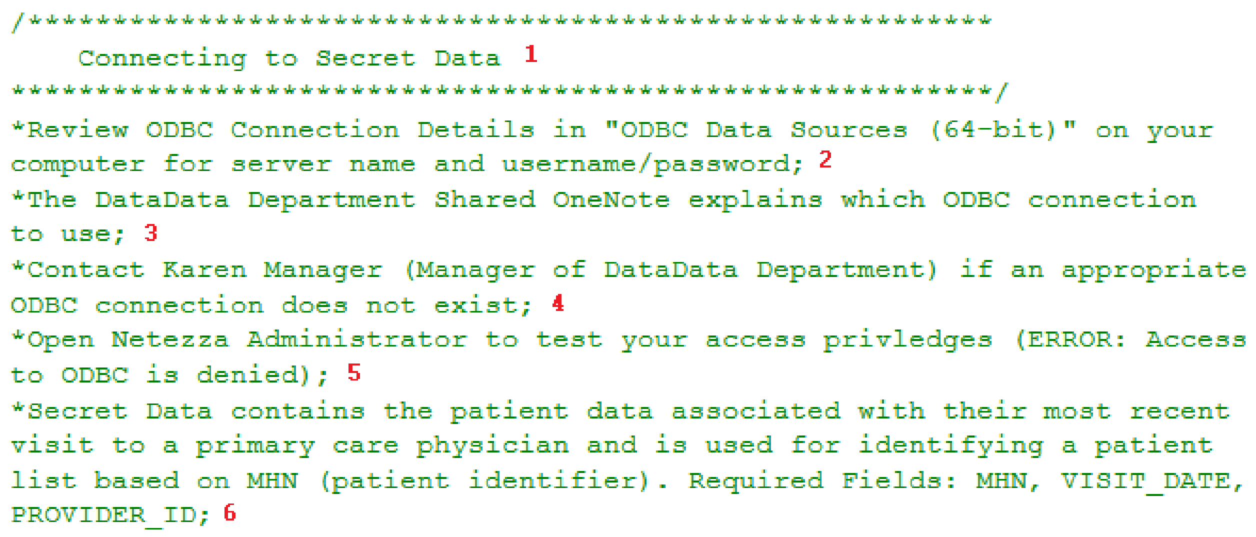
location paths that use specific drive names or have other specifically named components. For the same reason we avoid emails, the network/web location may also change. Using descriptive words like “hosted on the department SharePoint site”, “Biostatistics Department Shared Drive”, or “Project ___ Home location” may be just as helpful, if not more helpful. When new programmers take over, knowing where all associated items exist is high priority.

- **Major Change Authors:** Even if the code is in version control or a change log exists, having a major change history with important dates and authors may be helpful. While there’s a balance to what is helpful or not helpful in terms of specificity, a log of who performed the major changes may be helpful in tracking down correspondence, additional documentation, or other related materials. If your department does not already have a system of unique identifiers for its programmers that is unlikely to change, or if initials are your best method, consider having one place where the full name (and minimalist details) are laid out.

KEEP YOUR SECRETS, KAREN

Working in healthcare, privacy and protection are always top of mind. In our code, it may be required to have an access point (such as ODBC libname) established with credentials. The use of macro variables and user-specific credentials can avoid the overt listing of credentials in written code, but communication about what is needed should not be avoided. At first this may seem counter-intuitive, but eloquently explaining what is needed does not need to include usernames, passwords, secure location paths, or other “secrets”. Through commenting, we should be able to convey context, direction, and troubleshooting without spilling the protected information.

Figure 1. Covert Commenting



```
/**********************************************************
    Connecting to Secret Data 1
**********************************************************/
*Review ODBC Connection Details in "ODBC Data Sources (64-bit)" on your
computer for server name and username/password; 2
*The DataData Department Shared OneNote explains which ODBC connection
to use; 3
*Contact Karen Manager (Manager of DataData Department) if an appropriate
ODBC connection does not exist; 4
*Open Netezza Administrator to test your access privledges (ERROR: Access
to ODBC is denied); 5
*Secret Data contains the patient data associated with their most recent
visit to a primary care physician and is used for identifying a patient
list based on MHN (patient identifier). Required Fields: MHN, VISIT_DATE,
PROVIDER_ID; 6
```

Covert Language for Detailed Context

In Figure 1 detailed context is given by markers 1 and 6. The context that should be given at each point of the code is a general section header that gives the desired outcome or purpose of the following code. These headers should follow a standardized style and purpose (covered in Standardizing Styling to Provide Meaning) and appear throughout any code. Most likely, this is the commenting that already exists in most code.

The section header’s placement as the 1st comment is as important as its styling. When we’re skimming code, we read what comes first. By putting descriptive information first, you improve the efficiency of code review and updates by putting the most important information first. The length of the text should also be optimized to be succinct (less words; more meaning), basic (easy to understand—not jargon), and descript. This is the place for the most revealing information to be given. While we should avoid being too descript, we also need to be direct. By sharing things like the databases specific name or most succinct description of the data, we quickly identify the target and avoid pauses for the reader.

If the section header draws the attention of the reader, you might think the next thing would be the descriptive text (marker 6), but it comes last because of the second skimming tendency: to read the text just above the code to determine what is in the dataset/target or what the code is doing. Errors are given in the same way; where the problem code is identified just before the error.

The arrangement of the descriptive text is also intentional. Putting dependencies, logic, or non-secret metadata attributes last, allows for "error corrector" readers to find what they're looking for quickly when reviewing the log. The first part of the descriptive text does what would naturally come next: what is the code doing and what expectations do we have for its contents. Specific data should not be detailed but rather a focus on the data's attributes or qualities. By focusing on attributes and qualities, we allow the reviewer a way to fact check what they have, without revealing anything. For example, specifying that there should be one row per patient.

Direction not Details

Next in Figure 1 we aim to give direction to reader about what the targets of the code are. Markers 2 and 3 give direction without giving information. If the code has a target (especially one external to a SAS session), then specifying where the target is physically is tempting. Not only is this potentially secret information, there are additional downsides.

A big downside of giving a physical path or other specific location details is that locations and targets can change. By giving the reader the tools to find the answers instead of the answers themselves, you keep the information updated, private, and free from the "update needed" list (covered in The Friendship of Current and Future You: A Teamwork Story). Providing direction makes your code a timeless "this way" sign instead of a hindrance to your future readers; who should be questioning how up to date this information is. One of the biggest disasters you also avoid is hitting a "dead source" that is no longer updated and whose original source cannot be determined.

ERROR: What Does it Mean?

Next is the point of contact reference (Marker 4 in Figure 1) that the future coder (you or someone else) can fall back on. Note that I've applied the rule about author constants from Who is HARRY? a contingency PLAN, by adding the role of the contact and sticking to names instead of email addresses or phone numbers. Contact guidance it also provides a means to troubleshoot and resolve known issues. If you've encountered a familiar error, let your reader also become familiar with the resolution. A good way to find these is to run code that works well for you on a different computer or have someone else run it. Any errors that pop-up, further settings that need to be changed, or extra "to-dos" that need to be done before getting the desired result are all great candidates for commentary.

Continuing with the idea of helping your reader troubleshoot, Marker 5 represents a circumstance where the problem isn't the code. Denied privileges are one of the most common reasons for unnecessary edits that cause more harm than good. While the tendency is telling the reader they must have access to a certain location, database, or package, the better way is to tell them how to check. If, in this case, they can access the data using the other software, then the problem might really be the code/SAS. Otherwise, you've just saved a lot of potentially useless changes that may even need to be undone.

THE PATCHWORK CODE

Unlike Elmer, the patchwork elephant in the popular children's story, inheriting patchwork code isn't celebrated for the uniqueness it offers. While it can show you a multitude of different programming styles and preferences, patchwork code is almost never the most efficient solution. However, if a re-write isn't an option, the next best thing you can do is annotate appropriately.

The minutes it takes you to comment your code can save the next programmer hours. If you already know what your code does, why you chose the methods you did, and what can be improved, then you can likely answer most of the questions the next programmer will be asking.

The Friendship of Current and Future You: A Teamwork Story

The world of programming is changing and improving at rapid speed. New methods are being created by new and seasoned programmers alike. One of the few things we can always depend on is the continuation of time with little ability to see what's coming next. The saying goes that "hindsight is always 20/20"…so let's talk to our future and make it count.

Here are a few examples of times when you can expect your future self to be capable of fixing your current problems and where best to write it down. Note, this does not always mean that availability will allow for these fixes.

- **Downtime Activities List**: this should be a list of programs/updates that are both known to you and are currently achievable. These are functional programs/code that could use updating because they were inefficiently written originally, were patched instead of fixed, or because a new development in your programming ability has a known use case. These should be tracked in a shared document/spreadsheet in addition to within the code itself. By having these in a list, you avoid the need to depend on memory. By putting them in the code, you cordially invite future readers/authors to help and save their time in return. A reference to your list in the code, may be helpful in the header of the program as a reminder to yourself, but will be mostly of no interest to others. When adding these to either location, it is important to denote why this is needed/why the update wasn't already performed.
    - **Examples**:
        1. Data which can be condensed to one DATA step after results have been validated. Separated for validation of data transformations.
        2. Code that uses a DATA step to stay consistent with previous methods which can be converted to PROC SQL for thread processing.
        3. Something to check file presence in a directory would be helpful, but you're not aware of a way to do this and have no time to investigate prior to the current deadline
- **Wish List Items**: These are especially helpful comments about work-arounds, clumsy cluttered code, and given in moments when you wish/hope there will be a better way. If you are currently unaware of a better way to do it or a better way is not achievable due to outside restraints then consider adding to your wish list. These are good subjects for future coding developments, shared macros, and "downtime list" wannabes. Since it's unknown how to fix them at the time, I would suggest inline only commentary. The only caveat would be towards programmers whom do not have enough resources (time, mentors, or access) to fully investigate existing means to fix the problems.
    - **Examples**:
        1. Noting a step is done in multiple steps due to hardware constraints of the computer.
        2. Explaining SAS/Access Netezza Host is not licensed and they should use ODBC connection instead.
        3. Instruction to place code in macro to avoid errors of macro syntax in open code. Future versions of SAS allow this, so it could be implemented after upgrade.
- **Errors Explained:** A listing of current issues with the code is usually helpful at the top of the program. If you get in the habit of listing issues in the program header, you will avoid the need to read the entire program to determine known issues. Copying them into the body of the code or using a %PUT statement to print them in the log may also be good options. Issues with the outcome of the code (data/output errors) and reoccurring/expected errors in the log are great candidates. Put header references next to the known issues/errors if you already know which part of the code is causing the error. Error listing is most important if the errors should not be

fixed/suppressed (example 1 below), have a known reason (example 2 below), or a known solution to the error exists (covered in ERROR: What Does it Mean?).

- **Examples:**
  1. PROC SURVEYSELECT will error if not enough rows are available for the desired sample size. Do not suppress this error or fix sample size. It indicates the exact number needed for adjustment and will report this in the output as desired.
  2. %PUT statement purposely generates an error in the log if the most recent refresh date in the data is not today. If not expecting this, refresh source data and re-run code.
  3. Transpose creates duplicate rows for a single identifier. Use following PROC SORT NODUPKEY code below until root cause is identified/resolved.

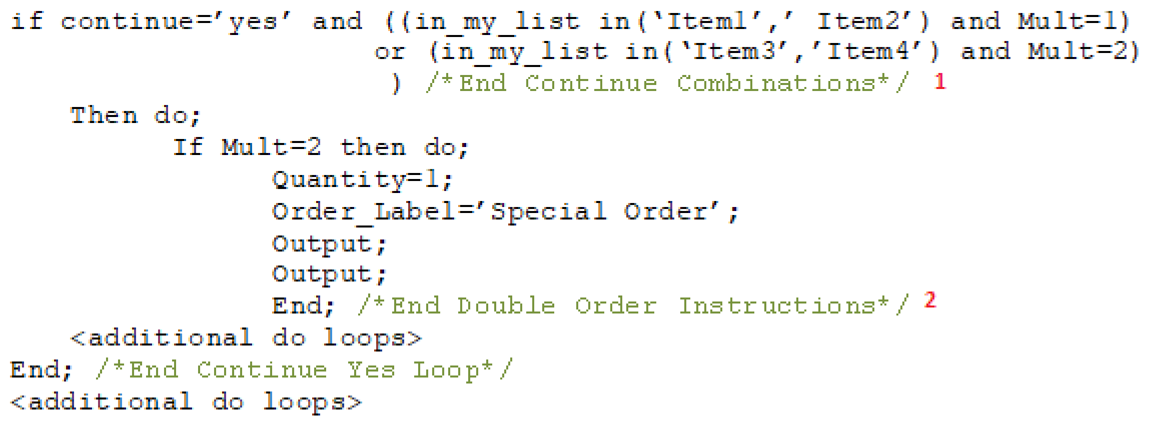
END THE DO-LOOP & MACRO MESSY NEST

Patchwork code is renowned for lack of organization inside and potentially outside of the code, but ubiquitous organization and documentation can improve any code base and even unrelated programs. Shared macros and code that is commonly used amongst programming teams are the best candidates for improvement by these tips. Patchwork, multi-authored, and lengthy code are also good targets.

Label Your Pairs

One of the easiest ways to improve the readability of your code is to label code that "needs a pair" and for which a pair is not easily identified or one of many. For example, in the nested do loop you may have multiple END statements that pair up with one of many IF/IF-ELSE/ELSE statements (See Marker 2 Figure 2). By labeling which logic block the END completes, the place for additional code or editing is more obvious. The same goes for paired punctuation, such as parentheses (See Marker 1 in Figure 2). By identifying the pair, you easily identify a place for additional logic or logic editing.

Figure 2. Labeling Pairs



```
if continue='yes' and ((in_my_list in('Item1',' Item2') and Mult=1)
                        or (in_my_list in('Item3','Item4') and Mult=2)
                         ) /*End Continue Combinations*/  1
    Then do;
           If Mult=2 then do;
                  Quantity=1;
                  Order_Label='Special Order';
                  Output;
                  Output;
                  End; /*End Double Order Instructions*/ 2
    <additional do loops>
End; /*End Continue Yes Loop*/
<additional do loops>
```

The Problematic Wormhole and Its Friends

The use of external programs (via %INCLUDE) can be a great way to shorten the length of code and re-utilize existing code, but can be a nightmare when it comes to tracking down problematic code. The same is true of macro definitions utilized through item stores, options defined in autoexec programs, and items/settings defined in the user profile, SASUSER library, or SAS User Interface (ex. SAS Enterprise Guide ®). All are examples of situations where the problem may be located outside the program being viewed.

You may be a victim if you have experienced the following scenarios:

- You did not change the code and suddenly it no longer works
- The code works for you, but not for a coworker or on a different computer
- The output changes if you run the code in a new session or following the run of a different program

To avoid being a perpetrator, use one or more of the following:

1. Nest no more than two programs deep and label your nest in the original program. If you use an %INCLUDE statement in a program, the included program should only %INCLUDE one or less programs. If the included program includes another program, then you have gone too far.

```
%INCLUDE “C://myexternalprogram.sas”; *This program calls
C://my2ndexternalprogram.sas;
```

2. Co-locate your external programs (or copies of them) and specify the common location.

```
*This program %includes programs that %include other programs. A copy of
all called programs can be found in C://Programs/Nested Programs;
```

3. Specify anything required that you know is external to the current program and include location.

```
*If using SAS Enterprise Guide, make sure option for only ‘Basic Variable
Names (V7)’ is set under Options>Data>General;
```

PROVIDING THE WHY TO THE WHAT

As you attempt to update a program, you may encounter code that you’re unfamiliar with. If the function of the code or the syntax used isn’t obvious, the task of updating the code can become quite difficult. This is especially true when the code cannot be ran and/or no longer creates the desired outcome. If you aren’t familiar with the code and data, then you may not even know that it no longer creates the intended results. Conveying the function of code is imperative and must be understood, so most programmers already provide what the code is doing in commenting, but it isn’t always the solution.

To be sure that the function of the code is understood it is also crucial to explain why the code is doing what it’s doing. In this section we cover additional information to include in your code for the purpose of communicating why code is necessary and what other steps are required a priori.

DID YOU READ THE INSTRUCTIONS?

The best way to avoid confusion on the part of the future programmer when it comes to “functioning code that errors” is to define all steps that need to take place prior to running the code in order to obtain the desired result. This prevents the new programmer from spending time “fixing an error” that is truly useful. The SAS error won’t say “ERROR: You forgot to run program ReformatTheData.sas before running this section”. Instead, it may simply state “Import Unsuccessful”.

Below are a few instructions that should be mentioned:

- Other programs that need to be ran prior
- Data transformations required
- Circumstances that require different pre-processing (ex. If no appointment data available, run table definition prior to keep numeric/character definitions as desired.)
- Conditions that need to be met or cannot be (ex. Do not re-run without clearing data from previous runs)

HE SAID SHE SAID: DECISIONS THAT MATTER

Sometimes we choose to do things a certain way for a reason, but the reason isn't obvious to the next reader. This is especially true when a work around is required or when the reason is external to the program.

Situations when the reason for a decision may need to be listed:

- Hardware constraints (ex. Local drive not large enough to process locally)
- Security considerations (ex. Two libnames are used (one read-only and other full privileges) to avoid accidental overwrite of data in export only areas of the code)
- Available options/licensing (ex. We will connect using DBMS instead of NETEZZA due to company licensing)
- Business decisions (ex. Use server DB1 instead of DB2 per company policy)

STANDARDIZING STYLING TO PROVIDE MEANING

COMMENT STYLES

SAS has graciously provided a number of ways to comment in your code. In the most robust interpretation, that means having the option of `/*Your Comment*/` and `*Your Comment;`. Beyond the basics, there are endless opportunities for what you put between the `/**/`and `*;`. Below are a few examples to keep in rotation.

```
/***************************************************
*                                                  *
*               My Big Section Title               *
*                                                  *
***************************************************/
```

```
              /*************************

                         Headings

              *************************/
```

```
           /*****************************/
           /*     Sub-Heading           */
           /*****************************/
```

```
*This paragraph is usually put underneath a
Section title/heading/sub-heading entry and
describes the process;
```

```
      /*------------Output-------------*/
```

```
      [line of code]; *function/reasoning;
```

```
           /*Describe DATA step Below*/
```

Table 1. Examples of Coding Styles

You can see in each of the examples I've added how they can be used, but the important thing is to have a consistent system. This system should be easy to read and help to organize your code. When others are reading your code they should be able to tell what is going on without needing an explanation.

KEEPING STYLES CONSISTENT

Knowing what comment styles are available is helpful. Being able to type them in line as you go is an option. Beyond these basic concepts, the option to implement a standard style structure can be more successful if you place templates into keyboard shortcuts.

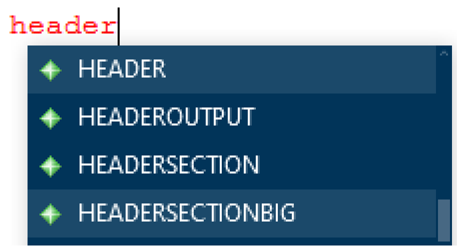
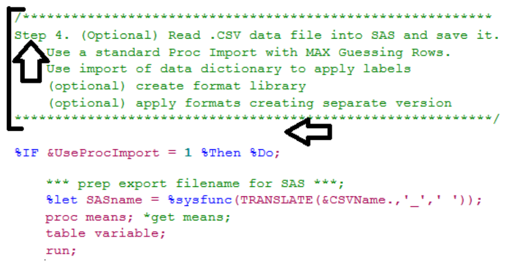


Figure 3. Keyboard Shortcuts for Labels

By having these templates available, you only have to type “header” to see all the options you’ve defined and appropriate naming should help you select a consistent option.

USING STANDARD COMMENTS

Since comments are a great way to denote important information in your code, the first thing they should be doing is increasing the readability of your code. Readability also can mean skimming for important information. By using styles that drawn attention as needed, you can save your readers time. A few things you can do to make the important comments stand out are: 1) Cover more than 3 rows, 2) Do not indent or start the comment -1 tab from code start, 3) use multiple lines of separation (create white space), and 4) place them on their own line. Even in the small picture of Figure 4, you can see the top comment stands out more than the two below it.

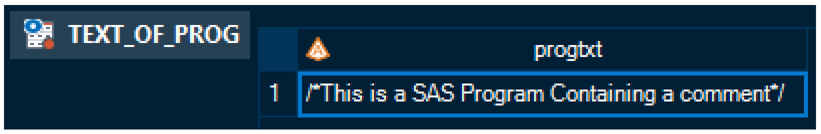


```
/**********************************************************
Step 4. (Optional) Read .CSV data file into SAS and save it.
    Use a standard Proc Import with MAX Guessing Rows.
    Use import of data dictionary to apply labels
    (optional) create format library
    (optional) apply formats creating separate version
**********************************************************/

%IF &UseProcImport = 1 %Then %Do;

    *** prep export filename for SAS ***;
    %let SASname = %sysfunc(TRANSLATE(&CSVName.,'_',' '));
    proc means; *get means;
    table variable;
    run;
```

Figure 4. Emphasized Comment Properties

If you keep comment styles standard, there are lots of ways to parse your code to use that information. Yes, SAS can read your code too! Since .sas files are considered text files, they can easily be “read” by other SAS programs. By denoting your program as a file, you can easily read the text of the program into a data set for processing (Figure 4):

```
filename sasprog "C:\Users\[username]\Downloads\Program 1.sas";

data TEXT_OF_PROG;
infile sasprog truncover end=eof;
length progtxt $1000 ;
input progtxt $1000.;
run;
```



	progtxt
1	/*This is a SAS Program Containing a comment*/

Figure 5. Dataset containing text of .sas file

Once you have made the text of your program available to a DATA step for processing the possibilities are endless for parsing of its contents; including the comments. By standardizing the way you comment, you can now process your program for important information programmatically.

HTML TAGGING: READING YOUR CODE LIKE A BOOK

Now that you know how to make your program available to a DATA step for processing, another point to consider is the end location of that information. One option is to have your SAS program help you create documentation for the programs in a given directory like Hughes did in his presentation at Midwest SAS Users Group 2018 (Hughes, 2018).

CONCLUSION

There are many ways to comment your code that keep the future in mind. Whether the next reader is you or someone else, conveying the background in a succinct, secure, and consistent manner is imperative. By mastering the art of commenting you will greatly improve your documentation skills and retain your future time for completing the task at hand.

REFERENCES

Hughes, Troy. 2018. “From Readability to Responsible Risk Management: Facilitating the Automatic Identification and Aggregation of Software Technical Debt within an Organization Through Standardized Commenting in SAS® Program Files and SAS Enterprise Guide Project Files” *Proceedings of the Midwest SAS Users Group 2018 Conference*, Indianapolis, IN: MWSUG. Available at http://www.mwsug.org/proceedings/2018/BL/MWSUG-2018-BL-127.pdf

CONTACT INFORMATION

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

Brooke Ellen Delgoffe, M.S.
Marshfield Clinic Health System
brooke_delgoffe@hotmail.com
https://www.linkedin.com/in/brookedelgoffe/