SESUG Paper 265-2019

You’re Doing It Wrong! Volume 002

Shane Rosanbalm, Rho, Inc.

ABSTRACT

You might think that you're a good programmer. But you're not. It's not just that you're doing it differently than I would do it. It's that you're actually doing it in a way that is unquestionably, incontrovertibly wrong!

But, take heart. I am here to set you on the righteous path. Listen to me, and you will be adored by your coworkers, accepted by SUG section chairs, and solicited by recruiters.

The focus of volume 002 will be the virtuosity of vertical code.

INTRODUCTION

Written communication is about more than just the words you use. The way you choose to organize those words on the page can have a significant impact on the reader’s ability to comprehend the material. For instance, consider the following two variations of the same string of sentences.

This is a sentence. String several of them together and you have a paragraph. Keep each individual sentence reasonably short. Longer sentences are more difficult to process, so don’t write overly long run-on sentences, else your reader will lose focus and not absorb whatever gasbaggery it is that you’re trying to pass off as wisdom. Keep paragraphs short as well. Overly-long paragraphs (ie, those with too many sentences in them) will cause your reader to become distracted. Indention of paragraphs and spacing between paragraphs also matters. You should indent or space your paragraphs to make the transitions easier to see visually.

This is a sentence. String several of them together and you have a paragraph. Keep each individual sentence reasonably short.

Longer sentences are more difficult to process, so don’t write overly long run-on sentences, else your reader will lose focus and not absorb whatever gasbaggery it is that you’re trying to pass off as wisdom.

Keep paragraphs short as well. Overly-long paragraphs (ie, those with too many sentences in them) will cause your reader to become distracted.

Indention of paragraphs and spacing between paragraphs also matters. You should indent or space your paragraphs to make the transitions easier to see visually.

The presentation on the left is too crowded. The lack of paragraphs makes it harder to maintain focus and absorb all that is being written. Whereas the presentation on the right, with its utilization of paragraphs, breaks the material down into several digestible nuggets. None of the words where changed, and yet one’s ability to comprehend the passage has increased significantly.

In the same way that we can make prose more readable through simple layout changes, so too can SAS® code be made more readable. In this paper we will explore various tips and tricks for achieving improved readability, not the least of which will be to verticalize our code. To paraphrase an Olivia Newton John song title from the 80s: “Let’s Get Vertical (Vertical)”.

ONE STATEMENT PER LINE

Consider the following PROC SORT example, written all on one line.

```
proc sort data=sashelp.cars out=mycars; by make model; where origin = "Asia…
```

This reads like a run-on sentence. One of the simplest readability techniques you can use in a program is to put each statement on a separate line.

```
proc sort data=sashelp.cars out=mycars;
by make model;
where origin = "Asia";
run;
```

INDENT NESTED CODE

Another very simple readability technique is to indent nested code. One of the more common examples is to indent statements within a DO block.

```
do i = 1 to 10;
   new = i*old;
   output;
end;
```

Applying this technique to statements within DATA and PROC steps is another very common practice.

```
proc sort data=sashelp.cars out=mycars;
   by make model;
   where origin = "Asia";
run;
```

DOUBLE INDENTION

A far less common practice is to add a secondary indentation to portions of statements that are related to content on an adjacent line. For instance, in this SQL block, the arguments after each keyword are indented so that they left-align.

```
proc sql;
   select   distinct make
   into     :asia1-
   from     sashelp.cars
   where    origin = "Asia"
   ;
quit;
```

The benefit of the double-indent becomes even more apparent when the queries get more complex.

```
proc sql;
   select   distinct make
   into     :asia1-
   from     sashelp.cars
   where    origin = "Asia"
            and drivetrain = "All"
            and mpg_city > 20
   ;
quit;
```

THE DANGLING SEMICOLON

Note the dangling semicolon in the example above. This will increase readability when there are multiple queries within a block.

```
proc sql;
   select   distinct make
   into     :asia1-
   from     sashelp.cars
   where    origin = "Asia"
   ;
   select   distinct make
   into     :europe1-
   from     sashelp.cars
   where    origin = "Europe"
   ;
quit;
```

BREAKING STATEMENTS ACROSS MULTIPLE LINES

Some of the statements in PROC SGPLOT have a lot of options that need to be applied to get the desired cosmetics in the output. Having all of these options on one line can cause readability to suffer.

```
proc sgplot data=forplot;
   series x=avisitn y=mean / group=trt01pn groupdisplay=cluster markers marke…
   scatter x=avisitn y=mean / group=trt01pn groupdisplay=cluster yerrorupper=…
   yaxis label="Weeks";
run;
```

Splitting the statement over multiple lines can increase readability. In this example the options related to grouping are on one line, the marker options on another, and the error bar options on yet another.

```
proc sgplot data=forplot;
   series x=avisitn y=mean /
      group=trt01pn groupdisplay=cluster
      markers markerattrs=(size=10px)
      ;
   scatter x=avisitn y=mean /
      group=trt01pn groupdisplay=cluster
      markerattrs=(size=0px)
      yerrorupper=upper yerrorlower=lower
      ;
   yaxis label="Weeks";
run;
```

In addition to splitting the options up based on relatedness, indention is used to show which options are associated with a given statement and dangling semicolons are used to give breathing room between statements. This verticalization significantly increases the readability and maintainability of the code.

This technique is not only applicable to plotting procedures. Consider a MERGE statement in which a multitude of datasets are being pulled together to form a subject-level analysis dataset.

```
data penultimate;
   merge initial bheight bweight dcsreas btsd10 bmprotyp bmpro bupro bflcr bri…
   by usubjid;
   …
run;
```

The simplest thing to do here is to add a break when the dataset names begin to roll off of the screen to the right.

```
data penultimate;
   merge initial bheight bweight dcsreas btsd10 bmprotyp bmpro bupro bflcr brisk
      bref2mmt bprevtrp bpriorth cyccompg cycinitg dosercvg dosecumg doseintg
      cyccompb cycinitb dosercvb dosecumb cyccompc cycinitc dosercvc dosecumc;
   by usubjid;
   …
run;
```

Yet another technique is to break the dataset names up even further based on relatedness. Eg, put the items that go into the demographics display on one line, disease characteristics on another line, etc.

```
data penultimate;
   merge initial
      bheight bweight dcsreas
      btsd10 bmprotyp bmpro bupro bflcr brisk bref2mmt bprevtrp bpriorth
      cyccompg cycinitg dosercvg dosecumg doseintg
      cyccompb cycinitb dosercvb dosecumb
      cyccompc cycinitc dosercvc dosecumc
      ;
   by usubjid;
   …
run;
```

COMPLEX MERGES

Some merges require extensive use of parenthetical dataset modifiers. Leaving all of these on a single line typically causes the code to run off the screen to the right. This is an obvious opportunity to verticalize the MERGE statement.

```
data labs;
   merge narrow (in=narrow) ranges (keep=testcode lower upper rename=(testcode=…
   by lbtestcd;
   if narrow;
run;
```

As you can see, the readability has been increased by splitting each dataset name, and the corresponding parentheticals, onto separate lines. Once again, the dangling semicolon adds some breathing room.

```
data labs;
   merge
      narrow (in=narrow)
      ranges (keep=testcode lower upper rename=(testcode=lbtestcd))
      ;
   by lbtestcd;
   if narrow;
run;
```

Going a step further with this line splitting technique, we can further increase readability by spitting the parenthetical RANGES modifiers up onto separate lines.

```
data labs;
   merge
      narrow (in=narrow)
      ranges (
         keep=testcode lower upper
         rename=(testcode=lbtestcd)
         )
      ;
   by lbtestcd;
   if narrow;
run;
```

The KEEP and RENAME modifiers are indented to show that they go with RANGES. The opening and closing parenthesis are put on separate lines to clearly show where the modifications start and end. The semicolon is left dangling to show where the MERGE statement finally ends. Have you ever seen such readability?!

MACRO CALLS

Macro calls are often written on a single line. While a macro call is technically only one statement, readability often suffers.

```
%parallelplot(data=sashelp.iris,var=sepallength sepalwidth,group=species);
```

One common strategy is to put each parameter on a separate line.

```
%parallelplot(data=sashelp.iris,
              var=sepallength sepalwidth,
              group=species);
```

While this is a good start, because the length of the macro name supplies the indention length, it suffers from not being robust to changes of indention (ie, when you move a macro call inside a DO block, the indention can get thrown off).

A more flexible option would be to indent all parameters (even the first one), and to indent them not based on the length of the macro name, but using the standard indent size (in this case 3 characters).

```
%parallelplot
   (data=sashelp.iris
   ,var=sepallength sepalwidth
   ,group=species
   );
```

You will note that in this last example the commas and parentheses were moved to the start of each line. This is more robust to the adding/subtracting of parameters during development. For instance, suppose you wanted to run the above macro call without the GROUP= parameter. If the macro's closing parenthesis is on the same line as the GROUP= parameter, you have more editing to do than if the macro's closing parenthesis is on its own line.

IF-THEN STATEMENTS

Most folks write their IF-THEN code as follows:

```
if gender = 1 then sex = "Male";
```

A quick perusal of the online documentation reveals that IF-THEN has the following structure:

```
IF expression THEN statement;
```

Because we are advocating for each statement to be on a separate line, this therefore logically leads us to rewrite our IF-THEN code as:

```
if gender = 1 then
   sex = "Male";
```

Justification: that which follows the THEN, being a statement, belongs on a separate line; it being a continuation of the previous line, we also indent it. Perhaps this is good practice, or perhaps it is good intentions run amok. But the auto-formatting Ctrl+I option, available within both Enterprise Guide and Studio, follows this rule, so it can't be too crazy.

%IF BLOCKS

It is all too easy to break your code when you introduce %IF blocks. The need for the double-semicolon in particular is easy to overlook in horizontal code.

```
data string;
   %if &type = LESS %then string = "short and sweet";
   %else %if &type = MORE %then string = "something more verbose";;
run;
```

But if we verticalize this same code, the required double-semicolon tends to jump out a little more.

```
data string;
   %if &type = LESS %then
      string = "short and sweet";
   %else %if &type = MORE %then
      string = "something more verbose";
   ;
run;
```

COMMENTS

Adding comments is not only good practice for documentation purposes, but this practice also has the potential to increase readability. The simplest form of comment is made with just a single asterisk.

```
*all possible subject/visit combinations;
data visitshell;
   set adam.adsl;
   do avisitn = 1 to 7;
      output;
   end;
run;
```

While this comment explains why the data step exists, readability is not the best. Many possible enhancements can be made to this basic comment. First, add some white space.

```
*all possible subject/visit combinations;

data visitshell;
   set adam.adsl;
   do avisitn = 1 to 7;
      output;
   end;
run;
```

Readability can be further increased by adding breathing space around the comment text. My preferred technique is to use a few dashes.

```
*--- all possible subject/visit combinations ---;

data visitshell;
   set adam.adsl;
   do avisitn = 1 to 7;
      output;
   end;
run;
```

Some comments are more important than others. A technique for drawing attention to these comments is to frame them.

```
*-----------------------------------------------;
*--- all possible subject/visit combinations ---;
*-----------------------------------------------;

data visitshell;
   set adam.adsl;
   do avisitn = 1 to 7;
      output;
   end;
run;
```

CONCLUSION

Convert to verticalism; the resulting code will be easier to read and modify. After getting over the initial shock of it, you'll come to love it. Let's get vertical!

CONTACT INFORMATION

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

Shane Rosanbalm
srosanba@gmail.com