Paper 128-2013

30 IN 20 THINGS YOU MAY NOT KNOW ABOUT SAS®

Timothy Berryhill, Wells Fargo

ABSTRACT

30 things you may not know SAS® can do. In 20 minutes, I hope to widen your eyes and improve your programming. I have used SAS on many platforms and operating systems, with many databases. Most of these ideas will run anywhere SAS runs.

INTRODUCTION

In this paper I present two theories of the best code. Then I discuss 30 underutilized features of SAS which support one or occasionally both of my approaches.

WHAT IS THE BEST CODE?

The best code is correct and clear. Assuming you can achieve both of those, then you can decide whether to save human time or machine time. The best code is correct, because if the answer is wrong, nothing else matters. I am going to mention several features you may adopt to avoid silly errors or to understand where a program is going wrong.

The best code is clear. A program you write to use once may be expanded into a monthly process, and then passed along from analyst to analyst for years. You may also be asked to "refresh" a report months after you originally write and run it. In either case, someone will be fortunate if you avoid confusing or convoluted code.

My final measure of best code is ease. This can be measured several ways. The right measure depends on the situation. If a program runs every day and it runs for minutes or even hours, then make it easy for the machine. Look for wasted sorts, look for unneeded variables, and combine steps. Look for any chance to remove reads and writes (input and output or I/O). On the other hand, if your boss expects the results before lunch, then the best code is code you are comfortable using, even if some Computer Science major disrespects it. With the ideas, correct, clear, and easy in mind, consider using these features if you do not already.

USE A %LET ANYWHERE

The %LET statement is valid in "open code," (outside a macro definition). References to a macro variable defined in a %let statement will resolve in open code. While I generally consider the number of macros inversely proportional to the quality of the code, I will use a series of %LET statements at the top of a program the same way another programmer may use a parameter file: to pull everything which changes regularly to a single place in the code. This is valuable to me if, for example, the same date constant is required in nine places in a long program. I use this in some SQL code which references many tables, each with a year and month value embedded in the name.

REDUCE I/O: MERGE WITH A VIEW

I may want to merge two datasets by state and city, and then to merge the result to a third dataset by state. I can use a data step merge to merge the first two datasets, and then a second data step merge to add the third dataset. This reads the records in the first two datasets, writes them, then reads the combined records again and the third dataset, and writes it all again. I cannot code it as a three way merge because the BY statements are not the same. Only if I am willing to strain myself and future programmers, I will change the first merge to a view. If you are not familiar with data step views, the only change to the code is to add the VIEW= option on the data statement. The effect during execution is that SAS will read observations from the first two datasets, join them in memory, and then pass them directly to the second merge. This eliminates almost half the I/O. As always, the savings comes with a cost. Adding the view takes additional memory, and you cannot use a view as flexibly as you can use a dataset. I wrote some code once with a series of nested views which tried to use the same tape two ways at the same time. It eliminated 100% of the I/O and eliminated any useful results.

CONTROL A MANY TO MANY MERGE

I write most of my code in SAS and I believe it is an excellent language. With that said, it is easy to get awful results from a many to many merge in SAS. Consider the code below. This is intended to apply reversals to a transaction file, setting the final amount to 0 if the transaction was reversed. The two statements on the second line manage the IN= variables. Without this management, if the customer made three transactions for the same amount and then reversed one, all three would be reversed.

```
DATA FLAGGED;
  TRAN=0; REV=0;
  MERGE TRANSACT(IN=TRAN)
                 REVERSAL(IN=REV);
  BY ACCT_NUM EFF_DT TRAN_AMT;
  IF NOT TRAN THEN ABORT ABEND 67;
  IF REV THEN FINAL_AMT=0;
  ELSE FINAL_AMT=TRAN_AMT;
RUN;
```

TRAP UNLIKELY ERRORS

Did you notice: `IF NOT TRAN THEN ABORT ABEND 67;` in the above code? The abort statement stops the code. With the ABEND option, it can pass a chosen value out to the calling environment. If a problem seems unlikely, but it is possible, it will ruin the results, and it is hard to fix in advance in the code, this is a way to force the user to call a programmer. Particularly when I am coding ad hoc code with unfamiliar data under time pressure, I may sprinkle these through the code, somewhat as you can code an assert in other languages. I wrote some code which has run monthly for five years, generating this error message exactly once. When it happened, I fixed the offending data in an hour instead of spending four days up front solving it in code. Of course, I was fortunate I was not on vacation when it abended.

NOBS NAMES THE OBS COUNT

On a set statement, the NOBS option allows me to name a variable which SAS sets to the number of observations in the input dataset. SAS looks this value up from an internal table and plugs it into the variable before the step starts. This means I can test the observation count BEFORE executing the set statement the first time. If you have ever tried to code a report to say there are no observations in the error dataset, the NOBS option is one way to do it. This only works for disk files, not for views or tapes.

TOLERATE BAD DATA DURING READ

Perhaps you have worked in an environment where the data did not perfectly match the layout you were given. For instance, a field described as PD8.2 may have text blanks in it for missing data. When SAS reads such a record, it generates many lines in the log documenting the exception. If you believe your data is perfect, then these lines are wonderful as they alert you to the problem. More often in my experience, thousands of known errors obscure an unexpected and important error. Assuming you read your logs—yes, a big assumption—you can increase the value of the log by suppressing the expected errors. The simplest method is the double question mark informat modifier. Inserting the string “??” (without quotes) between the variable and the informat tells SAS to silently tolerate bad data. If the program defaults the ERRORS option to some reasonable value, then expected errors do not fill the log and unexpected errors still generate log notes.

TOLERATE BAD DATA LATER

When I am not seriously stressed, I will occasionally write code which might work, just to see if it does work. Recently I was pleasantly surprised to learn SAS supports the double question mark informat in the input function, as well as in an input statement.

BUILD A FORMAT FROM A DATASET

PROC FORMAT can be controlled with a dataset instead of statements. The trivial example below creates a format named tim which displays 1 as "blue" and 2 as "red"—yes, for this example it would be MUCH clearer to code the normal PROC FORMAT with a value statement. This method is most valuable if you have many values which change often enough to make maintaining the value statement difficult. It also helps if someone provides the values and labels in a form SAS can read. Typing them into a dataset just to avoid typing them into a value statement is not efficient by any measure.

```
data cntlin;
  fmtname='tim';
  start=1;
  label='blue';
  output;
  start=2;
  label='red';
  output;
run;

proc format fmtlib cntlin=cntlin;
run;
```

BUILD A DATASET FROM A FORMAT

PROC FORMAT can also generate a dataset from a format. The CNTLOUT option introduces the name of a dataset PROC FORMAT will create and populate with the details of a selected format. That format can be a format supplied with SAS or a user-written format. Generating a CNTLOUT dataset from a known format and printing it is an easy way to start understanding the many variables available in either a CNTLOUT or a CNTLIN dataset.

GET THE DSN FROM THE JOB FILE CONTROL BLOCK (MAINFRAME TRICK)

This feature was more useful in the dim past, before we got the dictionary views. On a mainframe, the JFCB contains information about the OS dataset an INFILE statement references. The JFCB option accepts the name of a variable and delivers a copy of the JFCB into that variable. If the INFILE references several concatenated datasets, the JFCB value changes as a data step moves through the input data. A program can capture and store the exact source dataset for each observation. The absolute generation for a GDG referenced by relative generation is also available in the JFCB.

USE A HASH TO AVOID SORTING

I might avoid hashing as a contract programmer because I do not expect most SAS programmers to understand the feature well enough to maintain the program. On the other hand, I/O is the enemy, sorting requires lots of I/O, and a hash can avoid sorting. The following example loads an existing dataset into a hash. It identifies the keys—the variables I would have coded in BY statements to use a sort and merge. Loading the hash is an executable statement, so it is important to execute it only once, at the start of the first iteration of the data step (while _N_ is one). With the hash defined, the SET statement brings in observations from a large and unsorted dataset. This code invokes the FIND method of the hash object using object oriented syntax. In my example, if the return code is 0—this indicates a match—the subsetting IF keeps the observation. I saw some code recently which used four hashes this way in a single data step, rather than sorting the large file several ways.

```
DATA DASD.IWANT;
IF _N_ EQ 1
THEN DO;
DECLARE HASH TARGET(DATASET: 'JULY2011');
TARGET.DEFINEKEY('ID', 'ACCOUNT');
TARGET.DEFINEDONE();
END;
SET DISK.BUNCHASTUFF;
RC=TARGET.FIND();
IF RC EQ 0;
RUN;
```

CONTROL THE LENGTH OF COMPARISON

In some languages, strings or character variables with different lengths are never equal. When you compare two character values in SAS using the EQ or = operators, SAS will pad the shorter value with blanks to make the lengths the same. If your code compares “CAT” to “CATTLE”, the actual comparison is “CAT   “ to “CATTLE”. These differ and SAS returns false or unequal. If you add a colon to the operator (EQ: or =:), SAS will truncate the longer value so that “CAT” EQ: “CATTLE” is comparing “CAT” to “CAT”.

ARE YOU ASSIGNING OR COMPARING?

This is subtle and you are free to disagree. The symbol “=” can be an assignment operator, or a comparison operator. It is ambiguous. For example, consider the statement:

```
MATCH = CAT=DOG;
```

Compare it with

```
MATCH=CAT EQ DOG;
```

The symbol “EQ” is clearly a comparison operator. I try to use EQ for comparisons and = for assignments.

TWO TESTS IN ONE

This one is also style; I do not think there is significant difference in machine efficiency. I like the syntax

```
WHERE 2007 LE YEAR LE 2011;
```

You could get the same effect with a WHERE and a WHERE ALSO, or with a WHERE with an AND, and sometimes you can use BETWEEN (although offhand I am not sure how to code BETWEEN to be inclusive or exclusive of the endpoints). This works with IF, too!

SAS CAN PASS A COMMAND TO TERADATA

In the code below, when PROC SQL identifies the execute statement, it will pass along the text inside the parentheses much the way the macro language would. It does not look for any sort of syntax error. It will expand macro variable references. This lets you take advantage of database-specific features SAS does not directly support. I have used this to create a database table with the exact column attributes I want for a join before using PROC APPEND to insert data.

```
proc sql;
connect to teradata (user=MYID pass=&passtdpid=stuff database=default);
execute( drop table EXPL.mine) by teradata;
execute(commit work) by teradata;
quit;
```

CHANGE PART OF A CHARACTER VARIABLE

I frequently use the SUBSTR function on the right side of an assignment statement, or in a comparison, to read just part of a character variable. The SUBSTR function is also valid on the left side of an assignment statement, to let you change just part of a character variable.

```
MY_STRING='A DOG BARKS!';
SUBSTR(MY_STRING,3,3)='CAT';
SUBSTR(MY_STRING,7,6)='SLEEPS';
```

COPY SELECTED RECORDS FROM A NON-SAS FILE

SAS is a powerful and flexible general purpose programming language. The step shown below will copy entire records from the file HUGE references to the file LITTLE_BIT references. There is no need to specify or even know the entire record layout. This example copies records with low key values. If the goal was a random sample or a small subset, the INPUT statement could be coded without any variables—just an INPUT to pull a new record into the buffer.

```
DATA _NULL_;
INFILE HUGE;
FILE LITTLE_BIT;
INPUT @19 SOMEKEY 12.;
IF SOMEKEY LT 12345 THEN PUT _INFILE_;
RUN;
```

QUICK LAZY “COMMENT OUT” CODE

Programmers will change a few lines of code into comments, to avoid their effect without losing them. This can avoid re-typing and even re-thinking if the code turns out to have been valuable. Be careful on mainframes which treat the symbol /* starting in column one as a JCL end of file mark!

```
/* SKIP this next bit for now PROC DELETE DATA=DISK.THEGOODSTUFF;
RUN;
Pay attention again starting here: */
```

QUICK LAZY RUIN YOUR PROGRAM

For reasons which escape me, these comments do not nest. In the example below, the programmer did not intend to ever run the PROC FORMAT code again, and wrapped it in /* */ commenting to leave it as documentation. Later some programmer intended to comment out a long stretch of code for the June run, with June’s execution picking up just after the normal deletion of DISK.THEGOODSTUFF. With the June comment added, the /* before PROC FORMAT is ignored and execution resumes just before the deletion. Pay attention again is taken as normal code and generates an error, but it is too late to save THEGOODSTUFF.

```
/* Skip this in June PROC DELETE…
/* Never again! PROC FORMAT FMTLIB… */
PROC DELETE DATA=DISK.THEGOODSTUFF;
RUN;
Pay attention again */
```

SLIGHTLY HARDER “COMMENT OUT”

For just a bit more work, you can have nesting comments. Define a macro you will not invoke. In this example, the first macro named skip is in the main environment, while the second macro named skip is in the reference environment of the first macro. The first %mend can only match the second %macro skip, and the second %mend can only match the first %macro skip. Macro definitions nest properly. By the way, as long as we have a macro there, I code the optional macro name on mend statements. On the occasions when I fail to nest macro definitions properly, this lets SAS warn me.

```
%macro skip;
Start a macro in main env
%* Not in June…;
PROC DELETE…
%macro skip;
This starts a nested macro
%* Never again!;
PROC FORMAT…
%mend skip;
End the inner macro
%mend skip;
End the outer macro
```

USE KEYWORDS AS NAMES

Some people do not like this, and it can definitely be over-used, but I feel it is tremendously advanced relative to some other approaches. Strings we think of as SAS keywords—like RUN and STOP—are perfectly fine dataset or variable names.

```
DATA RUN;
SET STOP;
ARRAY='My string';
DATE9=5;
LENGTH='Enough';
RUN;
```

I am a big fan of being able to use the keyword LENGTH as a variable name. There is a downside to this feature. If you omit the first semicolon in the code above, SAS will create three datasets. They will be named RUN, SET and STOP. They will each have one observation, with the three variables ARRAY, DATE9 and LENGTH. The original dataset named STOP is gone. The DATASTMTCHK option can reduce the danger of overwriting good data by omitting a semicolon.

GENERATE A SIMPLE RANDOM SAMPLE

The next code uses the system time to initialize the random number generator, so it generates a different series (and a different sample) each time you run it.

```
* Draw a simple 3% sample;
DATA SAMPLE;
SET SAVED.POPULATION;
IF RANUNI() LT 0.03;
RUN;
```

GENERATE “THE SAME RANDOM”

This uses the value you provide to initialize the random number generator. As long as you do not change the input dataset, it will draw the same sample each time you run it.

```
DATA SAMPLE;
SET SAVED.POPULATION;
IF RANUNI(234567) LT 0.03;
RUN;
```

FORCE AN ERROR

For debugging and understanding, I like the information SAS provides when an input record generates an error. I am referring to the block in the log which shows the input record both as text and in hex, with that handy ruler above it. It also shows the values all of the variables have as SAS finishes processing the observation. You can force this by setting _ERROR_ to 1, perhaps with code like this:

```
IF STUFF EQ 'NONSENSE' THEN _ERROR_=1;
```

WHY I TEST WITH OHIO

This is another “I/O is the enemy” tip. Looking at something in memory is fast. Looking at something on disk is slow. Looking at something across the network or on tape (if your OS supports that) is extremely slow. During development I save time by taking advantage of knowing how most of our datasets are sorted. Assume for a moment that our code for Ohio is 1, and our datasets are sorted by the state codes—not the state names. The view below will start reading the whole 63,000,000 observation national file. It will read all of the Ohio observations (and pass them into my next step or proc—I/O is the enemy so this is a view rather than a data step). When it finds the first observation for any other state, it stops. A WHERE clause would have to check every record. Despite the power of the WHERE, you would pay at least part of the I/O cost for each of the 63,000,000 observations. This stops when it gets the first Oregon record.

```
DATA OHIO/VIEW=OHIO;
SET HAS63MILLION;
BY STATE; * Just to remind us it is sorted;
IF STATE GT 1 THEN STOP;
RUN;
```

RENAME VARIABLES DURING READ OR WRITE

I support some code older than several people attending this forum. Some of it uses eight character variable names based on the first few characters or the first few consonants of COBOL variable names. I am not going to change the legacy code but I can at least use names I find more meaningful in my code. This PROC SORT changes the names as it writes the output dataset.

```
PROC SORT DATA=IN.OUR OUT=OUR(RENAME=(RGN_CD=IDACID=ACCOUNT));
BY RGN_CD ACID;
RUN;
```

Because the RENAME option is on the output dataset, the BY list uses the old names.

USE A WHERE CLAUSE IN A SORT

This example will only sort the records with DESC beginning with ‘HIGH’. If you do not want to trust me on that, you could code the WHERE as a dataset option on the input dataset.

```
PROC SORT DATA=IN.OUR
OUT=OUR;WHERE DESC EQ: 'HIGH';BY RGN_CD ACID;
RUN;
```

CONTROL DISPLAY OF MACRO-GENERATED CODE

If you must use macros (I admit I occasionally do), learn OPTION MPRINT. Then learn OPTION NOMPRINT:

```
OPTION MPRINT;
%MONTHUSE(PERIOD=01);
OPTION NOMPRINT;
%MONTHUSE(PERIOD=02);
%MONTHUSE(PERIOD=03);
%MONTHUSE(PERIOD=04);
%MONTHUSE(PERIOD=05);
%MONTHUSE(PERIOD=06);
```

I have cleaned up after programmers who use NOMPRINT. This instructs SAS not to put the macro-generated code in the log. It makes the log much shorter and cleaner, and makes it impossible to figure out what the log notes mean—you can no longer see the code the notes document. In the example above, SAS will give me the generated code with the log notes for the first invocation of the log, then just the notes for the rest of the invocations.

START A LONG UNIX JOB AND GO HOME

```
$ nohup sas humongous.sas &$ exit You have running jobs$ exit
```

Yes, you can always go home. This way, the job finishes while you sleep.

CONCLUSION

LEARN, SHARE, AND GROW

I hope you appreciate some of these tips. I like code which runs quickly--BUT FIRST

- Get it Right!
- Keep it Clear !
- CPU time is less expensive than you are
- Wrong answers no one else can fix are not efficient!

CONTACT INFORMATION

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

Name: Tim Berryhill

City, State: San Francisco, CA

E-mail: tim@aartwolf.com

Web: http://www.aartwolf.com/twb.html