Paper 146-2010

Tales from the Help Desk 4: Still More Solutions for Common SAS® Mistakes

Bruce Gilsen, Federal Reserve Board

INTRODUCTION

In 23 years as a SAS ® consultant at the Federal Reserve Board, I have seen SAS users make the same mistakes year after year. This paper reviews some common mistakes, and shows how to fix them. The following topics are reviewed.

1. Changing the order of variables in a SAS data set.

2. A PROC SQL SELECT clause, when no rows are selected.

3. PROC SQL SELECT clauses select formatted values.

4. Assigning a variable name as the value of another variable.

In the context of reviewing these mistakes, the paper provides details about SAS system processing that can help users employ the SAS system more effectively. This paper is the fourth of its type; see the references for three previous papers that review other common mistakes.

1. Changing the order of variables in a SAS data set.

Data set ONE is created from an external file and contains the following values.

```
Obs   name   email             sales   returns

 1    joe    joe@hotmail.com    100      3
 2    jim    jim@yahoo.com      200      5
 3    sue    sue@aol.com        150      7
```

The code that creates and prints data set ONE is as follows.

```
data one;
  length name $10 email $15;
  infile 'external-file-name';
  input @1 name $ @11 email $ @34 sales @40 returns;
run;
proc print data=one;
run;
```

The order of the variables generated by PROC PRINT is very important to the user. Here is the PROC PRINT output for data set ONE.

```
Obs    name         email         sales    returns

 1     joe     joe@hotmail.com     100        3
 2     jim     jim@yahoo.com       200        5
 3     sue     sue@aol.com         150        7
```

Now, Joe gets a new, longer email address, so the length of character variable EMAIL must be increased from 15 to 25.

In a DATA step, the length of a character variable is determined the first time it is used (“first usage”). Subsequent DATA step statements cannot change the length. Since EMAIL is in data set ONE, a DATA step that begins with the statements DATA TWO; SET ONE; will take the length of EMAIL from its length in data set ONE. To change the length, a LENGTH statement must be placed before the SET statement, as in the following code.

```
data two;
  length email $25;
  set one;
```

```
  if name = "joe" then email = "joelongname@hotmail.com";
run;
proc print data=two;
run;
```

The length of EMAIL has correctly increased to 25 to hold the longer value for Joe. But, the order of the variables generated by PROC PRINT, which was very important, has changed. Here is the PROC PRINT output for data set TWO.

```
Obs    email                      name    sales    returns

 1     joelongname@hotmail.com    joe      100        3
 2     jim@yahoo.com              jim      200        5
 3     sue@aol.com                sue      150        7
```

Why did the variable order change? In a DATA step, SAS orders the variables in a data set in the order that it encounters them. Here is how data sets ONE and TWO are created. The first time each variable is encountered is bolded.

- For data set ONE, variables are encountered in the following order.
  NAME EMAIL in the LENGTH statement
  NAME EMAIL **SALES RETURNS** in the INPUT statement.

  The variable order is as follows: NAME EMAIL SALES RETURNS.

- For data set TWO, variables are encountered in the following order.
  EMAIL in the LENGTH statement.
  NAME EMAIL **SALES RETURNS** in the SET statement.

  The variable order is as follows: EMAIL NAME SALES RETURNS.

More generally, the variable order can change when variables read from a data set with a SET, MERGE, or UPDATE statement are used before the statement that reads them. For example, the following statements reorder variables if they are coded before a SET, MERGE, or UPDATE statement in the DATA step:

ATTRIB, ARRAY, FORMAT, INFORMAT, LENGTH, RETAIN

In this example, the LENGTH statement changed the variable order.

The easiest way to specify the order of variables in a data set is with the RETAIN statement, because no other variable information is needed. For example, lengths must be supplied for all variables included in a LENGTH statement.

This use of the RETAIN statement might at first seem odd. The RETAIN statement is used to specify that a variable created by an INPUT or assignment statement, which is by default set to missing before each iteration of the DATA step, should instead retain its value from iteration to iteration. The reordering of variables described above is a (sometimes unintended) side effect of the RETAIN statement. In this case, RETAIN is used only for its side effect.

In the following code, EMAIL has the desired length, 25, and the variables are printed in the desired order.

```
data two;
  retain name email sales returns;
  length email $25;
  set one;
  if name = "joe" then email = "joelongname@hotmail.com";
run;
proc print data=two;
run;
```

Notes.

1. As noted, SAS orders the variables in a data set in the order that it encounters them. So, in the final version of the code, the RETAIN statement must precede the LENGTH statement. If the LENGTH statement preceded the RETAIN statement, SAS would have encountered EMAIL before NAME, and the variable order would have been as follows: EMAIL NAME SALES RETURNS.

2. Only variables that need to be reordered must be listed in the RETAIN statement or any of the statements that reorder

variables. Variables listed in the statement are reordered to be first, and all other variables follow the listed variables in their previous order.

In this example, we wanted to change the order as follows.

```
Old: EMAIL NAME SALES RETURNS
New: NAME EMAIL SALES RETURNS
```

The following RETAIN statement would have been sufficient.

```
retain name email;
```

3. In this example, another way to print the variables in the desired order would be to add a VAR statement to PROC PRINT. This would not change the variable order in the data set, so there would still be a problem if subsequent use of the data set required the original variable order.

4. In addition to printing results, another common example where the order of the variables matters is when the data are written to an external file for use by another application or language.

2. A PROC SQL SELECT clause, when no rows are selected.

Data set ONE contains the following values.

```
Obs   name   email             sales   returns

 1    joe    joe@hotmail.com    100      3
 2    jim    jim@yahoo.com      200      5
 3    sue    sue@aol.com        150      7
```

First, we send a congratulatory email to anyone with sales greater than 100. This code includes the following steps.

1. In PROC SQL, a SELECT clause creates macro variable SEND_EMAIL, containing a space-separated list of values of the variable EMAIL from observations in which SALES is greater than 100. In this example, SEND_EMAIL has the following value: jim@yahoo.com sue@aol.com.

2. In a macro, check if SEND_EMAIL has a null value. If not, execute a DATA step that selects the email addresses in SEND_EMAIL one at a time and sends an email to everyone with sales greater than 100.

```
  /* Create &SEND_EMAIL, has space-separated email addresses of
     anyone with sales gt 100 */
proc sql noprint;
  select email
  into :send_email separated by ' '
  from work.one
  where sales gt 100;
quit;

  /* If anyone had sales gt 100, run DATA step and send them an email */
%macro goodsales;
  %if &send_email ne %then %do;
    data _null_;
      i=1; /* which email address */
      email_addr = scan("&send_email",1," ");
      do while (email_addr ne "");

        /* Code to send emails to staff with sales greater than 100 (jim and sue
           in this case) goes here.  It is simple but omitted for brevity. */

        i+1;
        email_addr = scan("&send_email",i," ");
      end;
    run;
  %end;
%mend goodsales;
%goodsales;
```

Now, we send a email to anyone with more than 9 returns. As before, PROC SQL creates macro variable SEND_EMAIL with a space-separated list of values of EMAIL that meet the selection criteria (observations in which RETURNS are greater than 9), and a macro conditionally executes a DATA step if SEND_EMAIL is not null.

```
  /* Create &SEND_EMAIL, has space-separated email addresses of
     anyone with returns gt 9 */
proc sql noprint;
  select email
  into :send_email separated by ' '
  from work.one
  where returns gt 9;
quit;

  /* If anyone had returns gt 9, run DATA step and send them an email */
%macro badreturns;
  %if &send_email ne %then %do;
    data _null_;
      i=1; /* which email address */
      email_addr = scan("&send_email",1," ");
      do while (email_addr ne "");

        /* Code to send emails to staff with more than 9 returns goes here.
           It is simple but omitted for brevity. */

        i+1;
        email_addr = scan("&send_email",i," ");
      end;
    run;
  %end;
%mend badreturns;
%badreturns;
```

Since nobody had more than 9 returns, we expect &SEND_EMAIL to have a null value so that no emails are sent. However, emails are sent to Jim and Sue because &SEND_EMAIL has the following value: jim@yahoo.com sue@aol.com

The problem is that when no observations meet the selection criteria, PROC SQL does not set SEND_EMAIL to a null value. In fact, nothing happens to SEND_EMAIL; if it is already defined, the value does not change, and if not, it continues to be undefined. In this example, SEND_EMAIL still has the previous value: jim@yahoo.com sue@aol.com. The same result occurs if both SELECT clauses are in the same invocation of PROC SQL.

Two ways to prevent this problem are as follows.

1. Before each PROC SQL SELECT clause, set the macro variable that stores the result of the PROC SQL SELECT to a null value.

```
%let send_email=;
```

2. The automatic macro variable SQLOBS contains the number of rows selected by the last PROC SQL statement. It is set to 2 in the first PROC SQL step, and 0 in the second PROC SQL step. In macros GOODSALES and BADRETURNS, test if &SQLOBS is not equal to zero instead of if &SEND_EMAIL is not equal to null. Change the &IF statements as follows.

```
Old: %if &send_email ne %then %do;
New: %if &sqlobs ne 0 %then %do;
```

Notes.

1. When no observations meet the selection criteria, PROC SQL writes the following note to the SAS log. The note is easy to ignore.

```
NOTE: No rows were selected.
```

2. Another useful automatic macro variable, SQLRC, contains a status value from the last PROC SQL statement. 0 indicates that the step was successful, with no errors or warnings. The *SQL Procedure* chapter of the *SAS Procedures Guide* has a complete list of status values.

A more comprehensive test of the SELECT statement is as follows.

```
%if &sqlobs ne 0 and sqlrc eq 0 %then %do;
```

3. SQLOBS and SQLRC change after every SQL statement. If there is any chance that another SQL statement will be executed prior to testing SQLOBS or SQLRC, copy SQLOBS and SQLRC to other macro variables and test those other variables instead. Here is an example with the second SQL SELECT statement.

```
  /* Create &SEND_EMAIL, has space-separated email addresses of anyone
     with returns gt 9 */
proc sql noprint;
  select email
  into :send_email separated by ' '
  from work.one
  where returns gt 9;
  %let save_sqlobs=&sqlobs;
  %let save_sqlrc=&sqlrc;
quit;

  /* If anyone had returns gt 9, run DATA step and send them an email */
%macro badreturns;
  %if &save_sqlobs ne 0 and &save_sqlrc eq 0 %then %do;
  ......
```

3. PROC SQL SELECT clauses select formatted values.

We want to subset DATA set ONE, selecting observations where the value of DATE is one of the values in data set GOODDATES. First, let's create data sets ONE and GOODDATES.

```
data one;
  format date yymmddn8.;
  informat date yymmdd8.
  input date v1;
  datalines;
20070301 1
20070401 2
20070401 3
20070501 4
20070601 5
20070601 6
20070701 7
;run;

data gooddates;
  format date yymmddn8.;
  informat date yymmdd8.;
  input date;
  datalines;
20070401
20070501
20070601
;run;
```

We can use PROC SQL to copy each distinct value of DATE in data set GOODDATES to a comma-separated macro variable, then use the macro variable with an IN operator to subset data set ONE.

```
proc sql noprint;
  select distinct date
  into :all_date separated by ','
  from gooddates;
quit;

  /* subset data set ONE, select only observations with the dates we want */
data finalone;
  set one;
  where date in (&all_date);
```

```
run;
```

We expect data set FINALONE to have 5 observations - observations 2-6 from data set ONE. However, FINALONE has no observations.

The problem is that DATE in data set GOODDATES has the format YYMMDDN8. PROC SQL applies the format when it writes values of DATE to macro variable &ALL_DATE. To illustrate, submit the statement *%put all_date= &all_date;*, which writes the value of the macro variable to the SAS log, as follows.

```
all_date= 20070401,20070501,20070601
```

The WHERE statement resolves to the following.

```
where date in (20070401,20070501,20070601);
```

DATE has SAS date values (17257 for April 1, 2007, 17287 for May 1, 2007, and so on) and does not match the values in the IN operator.

One way to prevent this problem is to specify the default format, BEST12., as follows.

```
proc sql noprint;
  select distinct date
  format=best12.
  into :all_date separated by ','
  from gooddates
quit;
```

Now, the statement *%put all_date= &all_date;* writes the value of the macro variable to the SAS log as follows.

```
all_date= 17257,17287,17318
```

The WHERE statement compares SAS date values (17257 for April 1, 2007, 17287 for May 1, 2007, and so on) to the values in ALL_DATE (17257, 17287, 17318), and data set FINALONE has the 5 observations from April, May, and June 2007, as expected.

```
Obs      date      var1
 1     20070401      2
 2     20070501      3
 3     20070501      4
 4     20070601      5
 5     20070601      6
```

4. Assigning a variable name as the value of another variable.

CALL VNAME allows you to create a character variable whose value is the name (not the value) of another variable.
The syntax of CALL VNAME is as follows.

CALL VNAME(*variable1*, *variable2*) where the value of *variable2* is set to the name of *variable1*.

For example, CALL VNAME (xxx, varname1) sets the value of VARNAME1 to “xxx”.

If *variable1* is an array reference, *variable2* is set to the name of the corresponding array element, as shown in the example below.

One use of CALL VNAME is to generate meaningful messages while looping through an array. When printing out which array element(s) met or didn't meet specified criteria, using variable names makes the output much more meaningful. This is illustrated in the following example.

Data set OLD contains the following values.

```
Obs   var1   var2   var3   charvar1

 1      1    -100    99    abc
 2     -2     200   999    def
```

We want to loop through the numeric variables in the data set, VAR1, VAR2, and VAR3, and print a message whenever a negative value occurs. The message will have the observation number, the name of the variable (obtained using CALL VNAME), and the value, allowing easy follow-up.

```
data new (drop=i badvar);
  set old ;
  array array1(*) _numeric_;
  do i=1 to dim(array1);
      /* check if array value is negative */
    if array1(i) < 0 then do;
        /* Negative value found, CALL VNAME sets variable
           BADVAR to name of current array element. */
      call vname(array1(i),badvar);
      put 'Observation: ' _n_  'Variable: ' badvar ' Value: ' array1(i);
    end;
  end;
run;
```

The DATA step writes the following to the SAS log. The value of BADVAR is a numeric missing value (.), not VAR2 in the first observation and VAR1 in the second observation as expected.

```
NOTE: Numeric values have been converted to character
      values at the places given by: (Line):(Column).
      25:28
Observation: 1 Variable: .  Value: -100
Observation: 2 Variable: .  Value: -2
```

The problem is that when the second argument to CALL VNAME (BADVAR in this case) is not previously defined, SAS defines it as a numeric variable. CALL VNAME expects the second argument to be character, so it sets BADVAR to missing.

To prevent this problem, specify BADVAR as a character variable of length 32. 32 is chosen because variable names can contain up to 32 characters. The following code returns the desired result; it differs from the earlier code because the LENGTH statement has been added.

```
data new (drop=i badvar);
  set old ;
  length badvar $32;
  array array1(*) _numeric_;
  do i=1 to dim(array1);
     /* check if array value is negative */
    if array1(i) < 0 then do;
        /* Negative value found, CALL VNAME sets variable
           BADVAR to name of current array element. */
      call vname(array1(i),badvar);
      put 'Observation: ' _n_  'Variable: ' badvar ' Value: ' array1(i);
    end;
  end;
run;
```

The following is written to the SAS log.

```
Observation: 1 Variable: var2  Value: -100
Observation: 2 Variable: var1  Value: -2
```

Notes.

1. The DROP= option is not required, but prevents extraneous variables from being included in the output data set.

2. The automatic variable _N_ is created in every DATA step, but is not included in the output data set. It contains the number of times the DATA step has executed (and is also the observation number in straightforward DATA steps).

CONCLUSION

This paper reviewed and showed how to fix some common mistakes made by SAS users, and, in the context of discussing these mistakes, provided details about SAS system processing. It is hoped that reading this paper enables users to better understand SAS system processing and thus employ the SAS system more effectively in the future.

For more information, contact the author, Bruce Gilsen, by mail at Federal Reserve Board, Mail Stop 157, Washington, DC 20551; by e-mail at bruce.gilsen@frb.gov; or by phone at 202-452-2494.

REFERENCES

Gilsen, Bruce (2003), "Deja-vu All Over Again: Common Mistakes by New SAS Users," *Proceedings of the Sixteenth Annual NorthEast SAS Users Group Conference*. <http://www.nesug.org/Proceedings/nesug03/bt/bt010.pdf>

Gilsen, Bruce (2007), "More Tales from the Help Desk: Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2007 Conference*. <http://www2.sas.com/proceedings/forum2007/211-2007.pdf>

Gilsen, Bruce (2009), "Tales from the Help Desk 3: More Solutions for Common SAS Mistakes," *Proceedings of the SAS Global Forum 2009 Conference*. <a href="http://support.sas.com/resources/papers/proceedings09/137-2009.pdf">

Go, Imelda C (2002), "Reordering Variables in a SAS® Data Set," *Proceedings of the Tenth Annual SouthEast SAS Users Group Conference*. <http://analytics.ncsu.edu/sesug/2002/PS12.pdf>

SAS Institute Inc. (2004), "*Base SAS 9.1.3 Procedures Guide*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Language Reference: Concepts*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), "*SAS 9.1.3 Language Reference: Dictionary, Volumes 1, 2, and 3*," Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2004), SAS Note 8395, "How to reorder variables in a SAS data set." <http://support.sas.com/kb/8/395.html>

ACKNOWLEDGMENTS

The following people contributed extensively to the development of this paper: Heidi Markovitz and Donna Hill at the Federal Reserve Board, Ronald Johnson at the Bureau of Labor Statistics, and Jason Secosky at SAS Institute. Their support is greatly appreciated.

TRADEMARK INFORMATION