Paper 104-27

Seeing Red: Tips for Debugging the SAS® Data Step

Michael Yee, Quintiles Late Phase, San Francisco, CA

ABSTRACT

Debugging SAS® software programs is a fact of life. Rarely does even the most talented SAS programmer create an error-free piece of code the first time. Understanding how SAS processes the DATA step and how that processing is presented in the SAS LOG is crucial to efficient debugging of SAS code. Several analytical techniques can reduce the frustration involved with debugging. Stepwise block submission of code can systematically locate hard to find program errors. Use of the RUN CANCEL statement can evaluate the DATA step for syntactic errors without actually reading the input data set. Data subsets can uncover functional SAS issues that arise because specific data elements cannot be evaluated properly by syntactically correct code. The line and column indicators in the SAS log can flag regions of code where errors are located. DATA step techniques such as the PUT statement can help identify run time changes in data values. These techniques and others can increase the efficiency of DATA step debugging.

INTRODUCTION

We have all experienced the frustration of being under deadline, writing DATA step code frantically in an attempt to obtain our SAS data set. When we push the little running man on the SAS toolbar, we want to see the SAS log and SAS output window pop up before us error free. Our blood pressure rises when red text appears in the SAS log. An error is identified and a correction is made. The program is resubmitted. Another red message appears, this time presenting an extensive collection of variables and their values and a message that the INPUT statement could not be completed. Why is SAS being so difficult?

There are systematic ways of debugging the code that can reduce time and frustration. By reading the notes and warnings in red, green, and black text we can gain a stronger understanding as to how the input data set is evaluated.

STEPWISE BLOCK SUBMISSION OF SAS CODE

When a data step fails and the error cannot be quickly pinpointed, then the easiest way of finding the erroneous code is stepwise block submission. This systematic means of evaluating code in blocks divides the DATA step into sections and evaluates each section for validity.

The steps are as follows (Valid code is in **BOLDFACE**):

1. Block off the lower half of the data step with comment blocks (/* *code* */). (Figure 1)
2. Submit code. If code runs flawlessly then unblock the top half of the blocked code so that only the bottom quarter of code is commented off. Repeat by half until error is located. (Figure 2 and 3)

Figure 1:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  codeline6;
  codeline7;
  codeline8;
/*
  codeline9;
  codeline10;
  codeline11;
  codeline12;
  codeline13;
  codeline14;
  codeline15;
  codeline16;
*/
RUN;

Submit
Valid
```

Figure 2:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  codeline6;
  codeline7;
  codeline8;
  codeline9;
  codeline10;
  codeline11;
  codeline12;
/*
  codeline13;
  codeline14;
  codeline15;
  codeline16;
*/
RUN;

Submit
Valid
```

Figure 3:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  codeline6;
  codeline7;
  codeline8;
  codeline9;
  codeline10;
  codeline11;
  codeline12;
  codeline13;
  codeline14;
/*
  codeline15;
  codeline16;
*/
RUN;

Submit
Valid
```

3. If an error occurs, the error is located between the last set of validated code and the commented out section of the program. (example part 4-6)

Figure 4:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  errorline;
  codeline7;
  codeline8;
/*
  codeline9;
  errorline10;
  codeline11;
  codeline12;
  codeline13;
  codeline14;
  codeline15;
  codeline16;
*/
RUN;

Submit
Valid
```

Figure 5:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  codeline6;
  codeline7;
  codeline8;
  codeline9;
  errorline10;
  codeline11;
  codeline12;
/*
  codeline13;
  codeline14;
  codeline15;
  codeline16;
*/
RUN;

Submit
Fail
```

Figure 6:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  codeline6;
  codeline7;
  codeline8;
  codeline9;
  codeline10;
  codeline11;
  codeline12;
/*
  codeline13
  codeline14
  codeline15;
  codeline16;
*/
RUN;

Fix Line 10
Submit
Valid
```

4. If the error is located in the first half, block of the last three-quarters and submit only the first quarter of the program (figure 7). If the program fails in the first quarter, then the error is in the first quarter. If it runs successfully, the error is in the second quarter (figure 8).

Figure 7:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  errorline6;
  codeline7;
  codeline8;
/*
  codeline9;
  errorline;
  codeline11;
  codeline12;
  codeline13;
  codeline14;
  codeline15;
  codeline16;
*/
RUN;
```

Submit Fail

Figure 8:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
/*
  codeline4;
  codeline5;
  errorline6;
  codeline7;
  codeline8;
  codeline9;
  codeline10;
  codeline11;
  codeline12;
  codeline13;
  codeline14;
  codeline15;
  codeline16;
*/
RUN;
```

Submit Valid

Figure 9:

```
DATA temp01;
SET inputds;
  codeline1;
  codeline2;
  codeline3;
  codeline4;
  codeline5;
  codeline6;
  codeline7;
  codeline8;
  codeline9;
  codeline10;
  codeline11;
  codeline12;
  codeline13;
  codeline14;
  codeline15;
  codeline16;

RUN;
```

Fix Line 6
Submit
Valid

5. Every time a code error is corrected, run the program in its entirety. If an error occurs, start again by commenting off the lower half of the unvalidated sections of code and resubmitting. If the program passes, then the error is in the bottom section of unvalidated code. If the program fails, then the error is in the top section of the unvalidated code.
6. Repeat submissions of each half of unvalidated code until the program runs error free.

Evaluation of code by halves until all code is valid is an efficient and orderly means of debugging your SAS program.

SYNTAX PROBLEM OR DATA EVALUATION ISSUE?

Understanding whether the program fails because of a syntactic error or a data-related error is important because there are different methods of identifying errors. Syntax problems are errors in the code that prevent the evaluation of any observation in the input data set. Syntax problems include spelling errors, unmatched parentheses, missing quotation marks, dropped semicolons, and a plethora of other problems. These problems are independent of the input data set's structure.

Data evaluation issues occur on syntactically correct code that fails when certain invalid data values are encountered (e.g. character/numeric mismatch, function parameters out of bounds, etc. . .) These evaluation errors arise when the DATA step encounters a value for a function that the DATA step cannot resolve. These types of DATA step problems are data set specific.

USING RUN CANCEL TO TEST SYNTAX

The RUN CANCEL statement at the end of a DATA step will test the DATA step syntax without bringing any data into the Program Data Vector. You can debug a program that incorporates a large data set without fear of having to evaluate the entire data set. The DATA step will stop prior to reading in the first observation and will indicate in the SAS LOG that the DATA step was not executed.

USE DATA SUBSETS TO TEST DATA VALUES

A subset of the input data set can be used to test the program's ability to evaluate data values. A small subset may indicate systematic data evaluation problems, such would occur when using a character function to evaluate a numeric variable. Non-systematic problems such as those that would occur when only certain data values are out of range may not be uncovered until larger subsets of data are evaluated.

LINE AND COLUMN INDICATORS

Understanding how the SAS log indicates line and column numbers will enhance the programmer's ability to pinpoint exactly where errors and automatic conversions occur. The line number indicates the numbered line in the SAS log where the problem exists and the column number indicates the problem's starting character.

```
Missing values were generated as a
result of performing an operation on
missing values.
Each place is given by: (Number of
times) at (Line):(Column). 32 at
4004:16   51 at 4075:18
```

This message indicates that the SAS operation performed at line 4004, column 16 and line 4075, column 18 in the SAS LOG were performed on missing values 32 and 51 times, respectively, and generated missing values based on assumptions regarding the handling of missing values.

More problematic are the line and column messages that appear when operations are performed within macros. The code below was generated while performing an operation within a macro program. The line number and the column number no longer correspond to the line and number in the SAS LOG.

```
NOTE: Missing values were generated as a result of performing
an operation on missing values. Each place is given by: (Number
of times) at (Line):(Column).
290 at 1:150   527 at 1:149   611 at 1:149   638
at 1:149   660 at 1:149
667 at 1:149   672 at 1:149   530 at 1:150   610
at 1:149   640 at 1:149
659 at 1:149   667 at 1:149   672 at 1:149   610
at 1:150   639 at 1:149
660 at 1:149   667 at 1:149   672 at 1:149   639
at 1:150   659 at 1:149
```

The line and column numbers now reference some other numbering scheme. Exactly how they refer to the macro program is unknown to this author. If anybody has the algorithm by which the messaging for macro programs assigns line:column numbers, please contact me in the email address at the end. I am genuinely interested.

THE PUT STATEMENT

A SAS programmer's best friend, the PUT statement allows run-time output of variable values to the SAS LOG. The usefulness of this statement is not to be underestimated. Being able to identify the value of a variable at specific locations in a SAS data step allows the SAS programmer to evaluate whether a SAS function is working properly. For example the code

```
DATA _NULL_;
LENGTH X 8;
X=1500;
PUT X;
Y=ROUND(X,2);
PUT Y;
RUN;
```

will put the values of X and Y to the SAS LOG, enabling the programmer to determine at a glance whether the ROUND function is being performed as expected.

By placing PUT statements before and after an evaluation, the SAS LOG will indicate how data values are transformed.

The PUT statement can also be used to set up alert messages which are SAS notes that indicate that abnormal or questionable values occur. Most SAS functions will generate missing values when the inputs cannot be evaluated. Rather than simply let the values revert to missing, the PUT statement can deliver an alert message indicating exactly which observation the missing value was assigned.

The following SAS program takes the individual month, day, and year variables and uses the MDY function to create a SAS date. If the month, the day, or the year variable is/are missing, then a message is written to the SAS LOG indicating which variable is missing. The program also issues an alert message when a calculated birth date is earlier than a predefined lower limit.

```
DATA temp01;
INPUT id $ birthmon birthday birthyr;
DATALINES;
A100    10      11      1969
A200    .       14      1945
B100    8       22      1977
B200    .       .       1975
C100    6       16      1877
;
RUN;

DATA temp02;
SET temp01;
IF birthmon^=. AND birthday^=. AND birthyr^=.
  THEN birthdat=MDY(birthmon,birthday,birthyr);
  ELSE DO;
    IF birthday=. THEN PUT "Cannot calculate
       birthdate: "id= birthday=;
    IF birthmon=. THEN PUT "Cannot calculate
       birthdate: "id= birthmon=;
    IF birthyr=.  THEN PUT "Cannot calculate
       birthdate: "id= birthyr=;
  END;
  IF birthdat<MDY(1,1,1900) and birthdat^=.
    THEN PUT "Calculated birthdate is prior to
       1/1/1900. Please confirm. "id= birthdat=;
  FORMAT birthdat date9.;
RUN;

PROC PRINT DATA=temp02;
RUN;
```

When the program is run, the following is written to the SAS LOG:

```
1    data temp01;
2    input id $ birthmon birthday birthyr;
3    datalines;

NOTE: The data set WORK.TEMP01 has 5
observations and 4 variables.
NOTE: DATA statement used:
      real time           0.04 seconds
      cpu time            0.03 seconds

9    ;
10   run;
11
12   DATA temp02;
13     SET temp01;
14     IF birthmon^=. AND birthday^=. AND
birthyr^=. THEN
birthdat=MDY(birthmon,birthday,birthyr);

15       ELSE DO;
16         IF birthday=. THEN PUT "Cannot
calculate birthdate: "id= birthday=;
17         IF birthmon=. THEN PUT "Cannot
calculate birthdate: "id= birthmon=;
18         IF birthyr=.  THEN PUT "Cannot
calculate birthdate: "id= birthyr=;
19            END;
20        IF birthdat<MDY(1,1,1900) and
birthdat^=.
21          THEN PUT "Calculated birthdate was
earlier than 1/1/1900. Please confirm. "id=
birthdat=;

22      format birthdat date9.;
23   RUN;

Cannot calculate birthdate: id=A200 birthmon=.
Cannot calculate birthdate: id=B200 birthday=.
Cannot calculate birthdate: id=B200 birthmon=.
Calculated birthdate was earlier than 1/1/1900.
Please confirm. id=C100 birthdat=16JUN1877
NOTE: There were 5 observations read from the
data set WORK.TEMP01.
```

```
NOTE: The data set WORK.TEMP02 has 5
observations and 5 variables.
NOTE: DATA statement used:
      real time           0.03 seconds
      cpu time            0.03 seconds

24
25    PROC PRINT DATA=temp02;
26    RUN;

NOTE: There were 5 observations read from the
data set WORK.TEMP02.
NOTE: PROCEDURE PRINT used:
      real time           0.03 seconds
      cpu time            0.03 seconds
```

CONCLUSION

Rather than haphazardly debugging a program that fails in the SAS compiler, the smart SAS programmer has a plan for systematically identifying and correcting program errors. Running a SAS program in discrete units quickly identifies the location of errors. Discerning whether a SAS error is the result of incorrect syntax or invalid data will determine the specific solution to implement. Understanding the line and column notes in the SAS LOG will quickly pinpoint the location of errors in the SAS code. And using of the PUT statement can demonstrate run-time how the values of a variable are transformed in the DATA step. The SAS programmer should be aware of these techniques and others when formulating a plan for efficiently writing programs.

CONTACT INFORMATION

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

Michael Yee, M.P.H.
Quintiles Late Phase
475 Brannan Street, Suite 400
San Francisco, CA
415-633-3100
michael.yee@quintiles.com