Paper PO–06

Mastering the Basics: Preventing Problems by Understanding How SAS® Works

Imelda C. Go, South Carolina Department of Education, Columbia, SC

ABSTRACT

There are times when SAS programmers might be tempted to blame undesirable results on a SAS error when the problem actually occurred because they did not understand how SAS works. This paper provides a few examples of how misunderstanding SAS data processing can produce unexpected results. Examples include those involving the program data vector, syntax, and behavior of PROCs. These examples emphasize the need for programmers to have a solid understanding of what their SAS code produces. Making the assumption that one's code is perfect before testing can lead to inadequate testing and costly but preventable mistakes. A safer approach is to assume that one's code might result in mistakes until testing proves otherwise.

Note: The sample code in this paper was tested using SAS Version 9.2.

First of all, this paper is not one about undesirable programming habits. The paper provides a few examples of situations where a clear understanding of how SAS works can prevent problems. Seeing these examples will hopefully provide SAS programmers with incentives to continue learning about SAS, which includes learning things that are new, reviewing past concepts, and searching for better ways to program.

Fortunately for SAS programmers, SAS continues to enhance its products. A popular example is the ability to use long variable names since Version 6. Prior to Version 6, variable names could not exceed eight characters in length. The author remembers conducting a SAS training session where all of the trainees were unaware that long variable names had already been available for a few years. At the end of the session, one trainee remarked that it was worth the time just to learn about the long variable names.

As SAS continues to add to the arsenal of tools available to SAS programmers, investing the time to learn more about SAS can increase efficiency and productivity. The author can think of a number of programming situations, which after certain PROC features took effect, old code could be replaced with code that was much easier to maintain and understand. There are also older features that are no longer supported, which emphasize the need to stay current with one's SAS knowledge.

With a good knowledge of SAS foundations and hopefully more, a programmer can proceed to write SAS programs with confidence that the code will produce the intended results. Unfortunately, knowledge is not enough. The programmer would be wise to develop good programming habits, such as validating the results against the code that generated the results. Sometimes extra variables need to be created and extra PROCs need to be run to perform checks. This step-by-step validation is extra work but is necessary to make sure that what the programmer intended was indeed reflected in the results.

The paper has the following examples:

1. Beware of syntax
2. Don't miss the missing values
3. Keep track of the last data set created
4. Know the rules
5. Think like SAS
6. Beware of features you may not need
7. Understand step boundaries
8. Know the difference between many-to-many merges in the DATA STEP and PROC SQL
9. Remember that many things can go wrong when manipulating data sets
10. Realize that not all numbers can be represented exactly on the computer

1. BEWARE OF SYNTAX

Just because a SAS statement compiles without an error in the SAS log does not mean it produces the intended results. To a new SAS programmer, the following statements might appear to calculate the same value when they actually may not all necessarily offer the same results.

	SAS Statement	Notes
A.	`XMEAN = (x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10)/10;`	XMEAN is missing if any values from x1 through x10 are missing.
B.	`XMEAN = mean (x1, x2, x3, x4, x5, x6, x7, x8, x9, x10);`	The mean is calculated using nonmissing values from x1-x10. The denominator is the number of nonmissing values, which is not necessarily 10 if x1-x10 have missing values.
C.	`XMEAN = mean (OF x1-x10);`	Because of the keyword OF, SAS interprets x1-x10 as a numbered range list consisting of 10 variables (x1, x2, …, x10).
D.	`XMEAN = mean (x1-x10);`	XMEAN is the mean of the difference between two variables (x1 and x10). Without the OF keyword, x1-x10 was treated as a difference and not as a numbered range list. If the programmer left out the OF keyword, then this statement is incorrect but will still compile because the syntax is still correct.
E.	`XMEAN = mean (OF X:);`	XMEAN is the average of the name prefix list that refers to all variables that begin with the specified character string (i.e., X in this case) preceding the colon. In the context of the above examples, XMEAN is the mean of x1-x10 if there are no other data set variables with names that begin with X.

Let's look at the following data set and code:

Obs	cars	boats
1	1	1
2	1	.

Given the note in row A above, one would expect to see one observation for:

```
proc print;
where cars+boats>0;
```

Given the note in row B above, one would expect to see two observations for:

```
proc print;
where sum(cars,boats)>0;
```

Given the note in row C above, one would expect to see two observations for:

```
proc print;
where sum(of cars boats)>0;
```

But in reality the last PROC PRINT statement has a syntax error as evidenced by the log below. Usage Note 14554 confirms that a syntax error results when using the OF operator within a WHERE statement. It also states: "The syntax for WHERE statements is derived from SQL, and in some cases does not provide for certain features otherwise available in SAS, such as the OF keyword. To prevent the error, specify each of the variables rather than using OF and a variable list."

```
618  proc print;
619  where cars+boats>0;
620
NOTE: There were 1 observations read from the data set WORK.ONE.
      WHERE (cars+boats)>0;
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


621  proc print;
622  where sum(cars,boats)>0;
623
NOTE: There were 2 observations read from the data set WORK.ONE.
      WHERE SUM(cars, boats)>0;
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


624  proc print;
625  where sum(of cars boats)>0;
                  ----
                  22
                  76
ERROR: Syntax error while parsing WHERE clause.
ERROR 22-322: Syntax error, expecting one of the following: !, !!, &, (, ), *, **, +, ',', -, /,
              <, <=, <>, =, >, >=, ?, AND, BETWEEN, CONTAINS, EQ, GE, GT, IN, IS, LE, LIKE, LT,
              NE, NOT, NOTIN, OR, ^, ^=, |, ||, ~, ~=.
ERROR 76-322: Syntax error, statement will be ignored.
626  run;

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

2. DON'T MISS THE MISSING VALUES

SAS offers 28 different ways to represent a numeric missing value.

Missing Value Type	Representation	Description
Regular Numeric	.	Single period
Special Numeric	.a .b .c . . . .x .y .z	Special representation: Single period followed by a letter Special numeric missing values are not case-sensitive (.A is equivalent to .a).
Special Numeric	._	Special representation: Single period followed by an underscore

Sooner or later a new SAS programmer is going to find out that a numeric missing value in SAS has a value less than any actual numeric value. Suppose x is a numeric variable. The condition of x<10 is true when x is a missing value or is an actual number less than 10. The numeric missing values also have an order as shown in the following table.

Increasing Sort Order of Numeric Values
._
.
.A
.B
.C
.
.
.
.X
.Y
.Z
nonmissing values

The condition below is true for all y values that are numeric missing values. (If the z below was not preceded by a period, z would be interpreted as a variable name.)

```
y<=.z
```

	SAS Statements	Notes
A.	`if grade <70 then lettergrade='F';`	The lettergrade value is F even when the grade value is missing. Be very careful with your conditional statements and make sure they reflect your intentions.
B.	`If .<grade <70 then lettergrade='F';`	Only nonmissing grade values less than 70 will result in a lettergrade value of F.

3. KEEP TRACK OF THE LAST DATA SET CREATED

The programmer has the option of specifying explicitly which data set should be used in a PROC step. If the data set is not specified, SAS will use the last created data set. To avoid the use of the wrong data set, it helps to always specify which data set should be used. It helps during troubleshooting especially when the last data set creation happened a while back.

XandY data set

Obs	x	COUNT	PERCENT
1	1	1	50
2	2	1	50

In the example below, the variable y is not found because PROC MEANS is attempting to perform the calculations using data set `Xonly` as the most recently created data set. The error in the log alerts the programmer that something is wrong. It would be even worse if there was no error in the log and the wrong data set was used for PROC MEANS. Without the error in the log, there's no alert that something might have gone wrong especially if the programmer is unaware the wrong data set was applied to PROC MEANS.

```
750  proc freq data=XandY; tables x/out=Xonly;
751
NOTE: There were 2 observations read from the data set WORK.XANDY.
NOTE: The data set WORK.XONLY has 2 observations and 3 variables.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

752  proc means; var x y;
ERROR: Variable Y not found.
753
NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

This is what the `Xonly` data set contains:

Xonly data set

Obs	x	COUNT	PERCENT
1	1	1	50
2	2	1	50

4. KNOW THE RULES

Many rules govern any programming language. In the example below, the OBS system option and OBS= data set option are used in the same program. In order to be able to anticipate the correct results, one should remember that the OBS= data set option in the SET statement overrides the OBS= system option. System options also remain in effect until they are changed.

Consider the two following data sets:

first data set

Obs	x
1	1
2	1

second data set

Obs	y
1	2
2	2

When OBS = 1 is set as a systems option, SAS will only process the first member of the data set within the SAS job. On the other hand, the OBS = 2 data set option in the SET statement overrides the OBS = 1 system option. In the DATA step, two observations will be processed from the `first` data set and the first observation will be processed from the `second` data set. PROC PRINT and PROC MEANS will each only use the first observation in the data set. PROC CONTENTS reports that the `final` data set has three observations.

```
289  options obs=1;
290  data final;
291  set first(obs=2) second;
292  ctr=1;
293
NOTE: There were 2 observations read from the data set WORK.FIRST.
NOTE: There were 1 observations read from the data set WORK.SECOND.
NOTE: The data set WORK.FINAL has 3 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

294  proc print data=final;
295
NOTE: There were 1 observations read from the data set WORK.FINAL.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
296  proc means; var ctr;
297
NOTE: There were 1 observations read from the data set WORK.FINAL.
NOTE: PROCEDURE MEANS used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

```
298  proc contents data=final;
299  run;

NOTE: PROCEDURE CONTENTS used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


                          The CONTENTS Procedure

Data Set Name        WORK.FINAL                              Observations          3
Member Type          DATA                                    Variables             3
Engine               V9                                      Indexes               0
Created              Thursday, August 02, 2012 11:34:42 AM   Observation Length    24
Last Modified        Thursday, August 02, 2012 11:34:42 AM   Deleted Observations  0
Protection                                                   Compressed            NO
Data Set Type                                                Sorted                NO
Label
Data Representation  WINDOWS_64
Encoding             wlatin1  Western (Windows)

                       Engine/Host Dependent Information

Data Set Page Size          4096
Number of Data Set Pages    1
First Data Page             1
Max Obs per Page            168
Obs in First Data Page      3
Number of Data Set Repairs  0
Filename                    C:\Users\igo\AppData\Local\Temp\SAS Temporary
                            Files\_TD3976\final.sas7bdat
Release Created             9.0202M3
Host Created                X64_VSPRO

                    Alphabetic List of Variables and Attributes

                #    Variable    Type    Len

                3    ctr         Num       8
                1    x           Num       8
                2    y           Num       8
```

5. THINK LIKE SAS

Let us consider the following data sets and the DATA step that combines the two using the SET statement.

first data set

Obs	race	gender
1	B	F
2	B	

second data set

Obs	group	race	gender
1	BM	B	M
2	BF	B	F

```
data final;
set first second;
if race='B' and gender='F' then group='BF';
else if race='B' and gender='M' then group='BM';
```

The `final` data set includes the variable `group` that appears to be the concatenation of the `race` and `gender` field values. But look at the second record and note that BF appears when `gender` is missing. Is this an error? It is, if this is not the type of result you want!

data set final

Obs	race	gender	group
1	B	F	BF
2	B		BF
3	B	M	BM
4	B	F	BF

In the context of how SAS processes data, this is ***not*** *an error.* The *SAS® 9.2 Language Reference: Dictionary* warns that all variables that are read with a SET, MERGE, MODIFY, or UPDATE statement are automatically retained. In the second record, `race` is B and `gender` is blank. Using just the logic of the IF-THEN/ELSE statements, the `group` value for the second record would be blank. However, the `group` value of BF was assigned to the second record through the effects of values being retained. The automatic retention of variables whenever the SET, MERGE, MODIFY, or UPDATE statements are used in a DATA step is a key piece of information vital to understanding how the program data vector (PDV) behaves.

The *SAS® 9.2 Language Reference: Concepts* defines the program data vector (PDV) as "a logical area in memory where SAS builds a data set, one observation at a time. When a program executes, SAS reads data values from the input buffer or creates them by executing SAS language statements. The data values are assigned to the appropriate variables in the program data vector. From here, SAS writes the values to a SAS data set as a single observation."

If the intention is to process data without the effects of retaining variables, the following code will do just that. In the first DATA step, the SET statement is first applied to the two data sets. In a second data set, the IF-THEN/ELSE processing is applied.

```
data final;
set first second;

data final2;
set final;
if race='B' and gender='F' then group='BF';
else if race='B' and gender='M' then group='BM';
```

data set final2

Obs	race	gender	group
1	B	F	BF
2	B		
3	B	M	BM
4	B	F	BF

6. BEWARE OF FEATURES THAT YOU MAY NOT NEED

Consider the following data set and PROC MEANS code.

test data set

Obs	school	student	score	gender
1	ABC	Leslie	81	F
2	ABC	Chris	82	M
3	ABC	Brandon	82	M
4	ABC	Judy	95	F
5	XYZ	Lane	81	M
6	XYZ	Susan	82	F
7	XYZ	Doug	82	M
8	XYZ	Angela	95	F

```
proc means noprint data=test;
id gender;
class school;
var score;
output out=stats1;
run;
```

The ID statement is used to include additional variables in an output data set. In this PROC MEANS example, the ID statement will include in the output data set the maximum value of *gender*. Since `gender` is a character variable, the maximum of F and M is M based on alphabetical or dictionary ordering. (Note: To get the minimum value, use the IDMIN instead of the ID statement.)

One may argue that the ID statement, in this example, is not appropriate because PROC MEANS produces only summary output. Regardless of what one's position is or reasons are for using the ID statement, the programmer is responsible for determining which parts of the output are meaningful in the context of what the programmer is trying to achieve. Just because SAS produces the output does not mean all parts of the output are relevant to the task at hand.

The following shows the output produced by the PROC MEANS statements above.

stats data set

Obs	school	gender	_TYPE_	_FREQ_	_STAT_	score
1		M	0	8	N	8.0000
2		M	0	8	MIN	81.0000
3		M	0	8	MAX	95.0000
4		M	0	8	MEAN	85.0000
5		M	0	8	STD	6.1875
6	ABC	M	1	4	N	4.0000
7	ABC	M	1	4	MIN	81.0000
8	ABC	M	1	4	MAX	95.0000
9	ABC	M	1	4	MEAN	85.0000
10	ABC	M	1	4	STD	6.6833
11	XYZ	M	1	4	N	4.0000
12	XYZ	M	1	4	MIN	81.0000
13	XYZ	M	1	4	MAX	95.0000
14	XYZ	M	1	4	MEAN	85.0000
15	XYZ	M	1	4	STD	6.6833

The _TYPE_ variable is important because it tells you which variables in the CLASS statement the summary statistics pertain to. For the sake of providing a simpler example, only `school` was used in the CLASS statement. One can more easily determine what the SAS data set means if one can see what code generated the data set. However, by just looking at the data set without looking at the SAS code, the trained eye could infer the following:

- If the reader can safely assume that this is the original output data set from PROC MEANS, then there was only one variable in the CLASS statement because _TYPE_ is either 0 or 1.
- The first five records (clue: _TYPE_=0) are summary statistics on the `score` variable for all the observations regardless of `school` (clue: `school` is blank) value.
- Because `gender` is never blank throughout the data set, that implies it could not have been listed in the CLASS statement. For _TYPE_=0 records, PROC MEANS will calculate the statistics regardless of the values in the CLASS statement. Therefore, the class variables will have no value for _TYPE_=0 records.
- All eight observations in the data set had a `score` value regardless of CLASS categories. (Clues: _FREQ_=8 whenever _TYPE_=0 and `score` is 8 whenever _STAT_ is N).

If the _TYPE_ and _FREQ_ variables were removed from the data set, as shown below, a less experienced reader might misinterpret what is in the data set and make the mistake of thinking that the `gender` might have been in the CLASS statement. Clearly, a solid understanding of how PROC MEANS produces variables in its output data sets can guard against such mistakes.

stats2 data set

Obs	school	gender	_STAT_	score
1		M	N	8.0000
2		M	MIN	81.0000
3		M	MAX	95.0000
4		M	MEAN	85.0000
5		M	STD	6.1875
6	ABC	M	N	4.0000
7	ABC	M	MIN	81.0000
8	ABC	M	MAX	95.0000
9	ABC	M	MEAN	85.0000
10	ABC	M	STD	6.6833
11	XYZ	M	N	4.0000
12	XYZ	M	MIN	81.0000
13	XYZ	M	MAX	95.0000
14	XYZ	M	MEAN	85.0000
15	XYZ	M	STD	6.6833

7. UNDERSTAND STEP BOUNDARIES

Step boundaries determine when SAS statements take effect. SAS executes program statements when SAS crosses a default or an explicit step boundary, such as:

- A DATA statement
- A PROC statement
- A RUN statement

However, there are exceptions. For example, PROC SQL executes without the RUN statement.

Consider the following SAS code written for the purpose of sending the PROC PRINT output to the PDF file named `print.pdf`. Note that there is no RUN statement after PROC PRINT.

```
data one;
input x;
cards;
1
2
;

ods pdf file='C:\Users\igo\Desktop\print.pdf';
proc print;
ods pdf close;
```

Without the RUN statement after PROC PRINT, the `print.pdf` file will be empty. This fact is noted in the log with a suggestion that perhaps the RUN statement did not precede the ODS PDF CLOSE statement.

```
683  data test;
684  input x;
685  cards;

NOTE: The data set WORK.TEST has 2 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds


688  ;
689
690  ods pdf file='C:\Users\igo\Desktop\print.pdf';
NOTE: Writing ODS PDF output to DISK destination "C:\Users\igo\Desktop\print.pdf", printer "PDF".
691  proc print;
692  ods pdf close;
NOTE: ODS PDF printed no output.
      (This sometimes results from failing to place a RUN statement before the ODS PDF CLOSE
      statement.)
693

NOTE: There were 2 observations read from the data set WORK.TEST.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

One way to avoid such a problem is to be explicit and to have the habit of always putting a RUN statement at every step boundary. In this way, the RUN statement is always there when one needs it even if it might have no effect. For example, the RUN statement has no effect on PROC SQL, which is executed immediately, as shown below in the SAS log.

```
NOTE: PROCEDURE SQL used (Total process time):
      real time           29:59.80
      cpu time            1.10 seconds


702  proc sql;
703  select * from test;
704  run;
NOTE: PROC SQL statements are executed immediately; The RUN statement has no effect.
```

8. KNOW THE DIFFERENCE BETWEEN MANY-TO-MANY MERGES IN THE DATA STEP AND PROC SQL

Consider two data sets, `data1` and `data2`.

data1 data set

Obs	gender	name1
1	Female	Linda
2	Female	Marcy

data2 data set

Obs	gender	name2
1	Female	May
2	Female	Gloria

The DATA step can handle one-to-one, one-to-many, and many-to-one matches but not many-to-many matches. For true many-to-many matches, the result should be a cross product. For example, if two records from each data set match the two records from the other data set by gender, the merged results should have $2 \times 2 = 4$ records.

gender	name1	name2
Female	Linda	May
Female	Linda	Gloria
Female	Marcy	May
Female	Marcy	Gloria

The following code merges both data sets in the DATA Step.

```
proc sort data=data1; by gender;
proc sort data=data2; by gender;
data combo; merge data1 data2;
by gender;
proc print;
```

The results only produce two records and do not include all possible combinations.

combo data set

Obs	gender	name1	name2
1	Female	Linda	May
2	Female	Marcy	Gloria

Use PROC SQL to obtain all possible combinations.

```
proc sql;
select data1.gender, name1, name2
from data1, data2
where data1.gender=data2.gender;
```

PROC SQL output

gender	name1	name2
Female	Linda	May
Female	Linda	Gloria
Female	Marcy	May
Female	Marcy	Gloria

9. REMEMBER THAT MANY THINGS CAN GO WRONG WHEN MANIPULATING DATA SETS

Consider two data sets, `time1` and `time2`. In the examples, `ss` stands for scaled score.

time1 data set

Obs	grade	last	first	ss	lunch	ssn
1	4	Garbo	Greta	434	R	111111111
2	3	Davis	Betty	380	R	222222222
3	2	Taylor	Liz	245	R	333333333
4	9	Kidman	Nicole	333	R	444444444

time2 data set

Obs	grade	last	first	ss	lunch	ssn
1	5	Garbo	Greta	533	F	111111111
2	4	Davis	Betty	493	F	222222222
3	3	Taylor	Liz	399	F	333333333
4	8	Loren	Sophia	723	F	555555555

When the two data sets are merged using a DATA step to merge `time1` and `time2`, the two data sets need to be sorted by `ssn` (or the BY-variables). When contributing data sets have variables with the same name, the variables need to be renamed in order to prevent the values of one data set from overwriting the values of the other data set during the merge. The merge results shown below are problematic because the variables with the same name were not renamed or dropped from either data set.

```
proc sort data=time1; by ssn;
proc sort data=time2; by ssn;

data test; merge time1 time2; by ssn;
```

Obs	grade	last	first	ss	lunch	ssn
1	5	Garbo	Greta	533	F	111111111
2	4	Davis	Betty	493	F	222222222
3	3	Taylor	Liz	399	F	333333333
4	9	Kidman	Nicole	333	R	444444444
5	8	Loren	Sophia	723	F	555555555

The following data step shows some variables being dropped and renamed prior to merging. The resulting data set has correct values.

```
data test;
merge time1 (in=in1 drop=grade lunch rename=(ss=ss1))
      time2 (in=in2 drop=grade lunch rename=(ss=ss2));
by ssn;
```

Obs	last	first	ss1	ssn	ss2
1	Garbo	Greta	434	111111111	533
2	Davis	Betty	380	222222222	493
3	Taylor	Liz	245	333333333	399
4	Kidman	Nicole	333	444444444	.
5	Loren	Sophia	.	555555555	723

What happens if the BY statement is *accidentally* omitted from the previous example? No error message will be given because it is valid SAS syntax and SAS does merges without BY statements. The records are merged in the order in which they occur on the data set and without regard to any other criteria. The resulting data are invalid and shown below.

```
data test;
merge time1 (in=in1 drop=grade lunch rename=(ss=ss1))
      time2 (in=in2 drop=grade lunch rename=(ss=ss2));
```

Obs	last	first	ss1	ssn	ss2	source
1	Garbo	Greta	434	111111111	533	both
2	Davis	Betty	380	222222222	493	both
3	Taylor	Liz	245	333333333	399	both
4	Loren	Sophia	333	555555555	723	both

10. REALIZE THAT NOT ALL NUMBERS CAN BE REPRESENTED EXACTLY ON THE COMPUTER

Numeric precision (i.e., the accuracy with which a number can be represented) and representation in computers are the roots of the problem. SAS uses floating-point (i.e., real binary) representation. The original decimal number and the binary-represented number may be very close, but very close is not the same as equal. There happens to be no exact binary representation for the decimal values of 0.1 and 0.3, which accounts for the difference in example #1 below. The advantage of floating-point representation is speed and its disadvantage is representation error.

Repeating decimals and irrational numbers are other obvious problems for exact storage on a computer. For example, 1/3 is equal to a decimal point followed by an infinite number of 3's. Computers cannot store an infinite number of digits.

We need to make a distinction between the expected mathematical result (our decimal values) and what the computer can store (binary values) and program accordingly. Readers may refer to two SAS technical support references (TS-230 and TS-654) listed at the end of this paper for in-depth explanations and examples regarding floating-point representation.

EXAMPLE #1

We know, as a mathematical fact, that 3 multiplied by 0.1 is 0.3. Therefore, when we examine the following code below, it would seem reasonable to expect that the `equal` variable will have a value of Y because both variables resolve to 0.3 (at least mathematically).

```
data one;
value1=0.3;
value2=3*0.1;
difference=value1-value2;
if 0.3=3*0.1 then equal='Y';
             else equal='N';
```

If you are thinking the only possible answer is Y, then you are in for surprise! Let's look at the PROC PRINT output for the data set above.

Obs	value1	value2	difference	equal
1	0.3	0.3	-5.5511E-17	N

If we use the following statement with PROC PRINT,

```
format value1 value2 32.31;
```

we get the following output.

Obs	value1	value2	difference	equal
1	.300000000000000000000000000000	.300000000000000000000000000000	-5.5511E-17	N

The two values are both 0.3, but that is only as far as the PROC PRINT output goes. The two values are stored in the computer differently. In a later page, we note that SAS formats round and that is why we are not able to see the difference in the values.

To see how the values differ, let us use the HEX16. format with PROC PRINT.

```
format value1 value2 hex16.;
```

We get the following output that shows the difference between the two values.

Obs	value1	value2	difference	equal
1	3FD3333333333333	3FD3333333333334	-5.5511E-17	N

EXAMPLE #2

Here is still another example. The difference for both pairs of numbers is mathematically 3.8, but the comparisons fail.

```
data test;
input value1 value2;
difference=value1-value2;
if difference=3.8
   then equalto3point8='Y';
   else equalto3point8='N';
cards;
16.3 12.5
15.7 11.9
;
```

Without specifying a format, we get the following PROC PRINT results.

Obs	value1	value2	difference	equalto3point8
1	16.3	12.5	3.8	N
2	15.7	11.9	3.8	N

If we use the following statements:

```
proc print;
format difference 32.31;
```

we get the following output.

Obs	value1	value2	difference	equalto3point8
1	16.3	12.5	3.80000000000000000000000000000000	N
2	15.7	11.9	3.79999999999999000000000000000000	N

EXAMPLE #3

Representation error can become a serious problem when one is unaware it could even happen and takes no precautions against it. Unaccounted for, the size of the errors or discrepancies could accumulate over multiple operations. Let's take the simple example of adding 0.1 ten trillion times. We know the result should be one trillion.

```
data test;
do i=1 to 10000000000;
sum+0.1;
end;
integer=1000000000;
diff=sum-integer;
drop i;
```

After adding all those numbers, SAS produces the following.

Obs	sum	integer	diff
1	1000000163.1	1000000000	163.124

Over so many, many calculations, the difference accumulated to 163.124. How serious is that? It all depends on your data. This might still be tolerable for some and totally unacceptable for others. Something else to think about is what happens to other results when the tainted sum is used in other calculations.

COPING WITH THE PROBLEM

We are responsible for our data, programs, and results. The first step in solving the problem is identifying the problem and being aware of the conditions under which the problem might create undesirable results. As far as the computer science field is concerned, this is a known problem. "Most of the published algorithms for numerical analysis are designed to account for and minimize the effect of representation error."(TS-230)

"Unfortunately there is no one method that best handles the problems caused by numeric representation error. What is a perfectly good solution for one application may drastically affect the performance and results of another application." (TS-230) Hence, this paper focuses on the simplest examples of this problem.

Coping Strategy #1: Keep It Whole

The safest way is to just deal with integers or whole numbers. If on a computer, the results of operations on integers are always integers, then there is no problem because an integer can be stored exactly in computers as long as the largest integer value the computer can represent has not been exceeded.

Whether you can stay within the realm of integers depends on what data are involved and what needs to be done to the data. Unless you're just adding, subtracting, and multiplying integers with integers, you could encounter a noninteger when it's time to divide an integer with another integer.

Consider the following example, which could be monetary amounts, such as dollars and cents.

```
data test;
input value;
integerversion=int(100*value);
cards;
18.1
118.18
;
```

The input values were multiplied by the scale factor of 100 (to "transfer" the digits after the two decimal places to the integer side of the number). The INT function, which returns the integer value of the argument, is then applied to remove the representation error that might have been introduced by the decimal or fractional portion of the input.

Obs	value	integerversion
1	18.10	1810
2	118.18	11818

You can proceed to apply integer arithmetic to the integer values. When you reach the last integer arithmetic result, you can divide it by 100 to regain the decimal portion. You can also apply a similar strategy to percentages. Percentages, such as 18%, can be multiplied by 100 and stored as 18.

Coping Strategy #2: Dare to Compare with Rounded Numbers

In examples #1 – #3 above, representation error manifested itself in the comparison of values. TS-654 recommends that you keep the following in mind when working with nonintegers or real numbers in general,

- ✓ Know your data.
- ✓ Decide on the level of significance you need.
- ✓ Remember that all numeric values are stored in floating-point representation.
- ✓ Use comparison functions, such as ROUND.

You can apply the ROUND function at strategic points in the calculation process (e.g., at the end of a series of calculations, after each calculation). What you do depends on the nature of the data, what you have to do with the data, and when representation error might become an issue. Before making an equality comparison, you can round one or both of the operands. An alternative to rounding is specifying to what degree two values are close enough so that they can be considered good as equal as far as your SAS programming is concerned. This process is called fuzzing the comparison. Refer to TS-230 for examples.

The ROUND function has the following syntax:

ROUND (*argument* <,*rounding-unit*>)

It rounds the first argument to the nearest multiple of the second argument. When the rounding unit is unspecified, it rounds to the nearest integer.

The *SAS® 9.2 Language Reference: Dictionary* reassures us that, in general, we can expect to produce decimal arithmetic results if the result has no more than nine significant digits and one of the following conditions is true:

- ✓ The rounding unit is an integer or is a power of 10 greater than or equal to 1E-15.
- ✓ The expected decimal arithmetic result has no more than four decimal places.

Refer to the *SAS® 9.2 Language Reference: Dictionary* for more details regarding the ROUND function. Should the ROUND function fail to meet your needs, you may specify your own fuzz factor to use with the ROUND function. TS-230 provides examples of how to do this.

Let us modify EXAMPLE #1 to include the ROUND function for both values.

```
data test;
value1=0.3;
value2=3*0.1;
difference=value1-value2;
if round(value1,0.1)=round(value2,0.1)
   then equal='Y';
   else equal='N';
```

This time we can expect the correct mathematical results because the ROUND function returns the value that is based on decimal arithmetic by rounding the values to the first decimal place.

Obs	value1	value2	difference	equal
1	0.3	0.3	-5.5511E-17	Y

Let us modify EXAMPLE #2 to include the ROUND function at the point of comparison.

```
data test;
input value1 value2;
difference=value1-value2;
if round(difference,0.1)=3.8
   then equalto3point8='Y';
   else equalto3point8='N';
cards;
16.3 12.5
15.7 11.9
;
```

Obs	value1	value2	difference	equalto3point8
1	16.3	12.5	3.8	Y
2	15.7	11.9	3.8	Y

Let us modify EXAMPLE #3 to include the ROUND function each time addition occurs.

```
data test;
retain sum 0;
do i=1 to 10000000000;
sum=round(sum+0.1,0.1);
end;
integer=1000000000;
diff=sum-integer;
drop i;
```

Obs	sum	integer	diff
1	1000000000	1000000000	0

REFERENCES

SAS Institute Inc. 2009. *Base SAS® 9.2 Procedures Guide.* Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2010. *SAS® 9.2 Language Reference: Concepts, Second Edition.* Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2011. *SAS® 9.2 Language Reference: Dictionary, Fourth Edition.* Cary, NC: SAS Institute Inc.

TS-230: Dealing with numeric representation error in SAS applications. Retrieved July 1, 2008, from the SAS Web site: http://support.sas.com/techsup/technote/ts230.html

TS-654: Numeric precision 101. Retrieved July 1, 2008, from the SAS Web site: http://support.sas.com/techsup/technote/ts654.pdf

Usage Note 14554: Syntax error when using OF operator within a WHERE statement. Retrieved August 1, 2012, from the SAS Web site: http://support.sas.com/kb/14/554.html

TRADEMARK NOTICE