PhUSE 2016

Paper IS03

How well can you speak SAS or is what you see what you get?

Yuliia Bahatska, inVentiv Health Clinical, Berlin, Germany

ABSTRACT

We all spend a lot of effort trying to keep up with new features and approaches in statistical programming. However, at the same time we should not forget about polishing our skills. In my paper I will concentrate on the nuances of SAS® that we come across every day. They might be very easy to miss but can be potentially threatening. I suggest you to take a test that consists of several very simple questions that are based on real life experience. They do not contain robust code or hardcore statistics but the obvious answer might not be the correct one. All of us have small gaps in our knowledge and the main purpose of this paper is to help fill in these gaps.

INTRODUCTION

It's done. Several hours of high concentration and hard work and your code is finalised. It is elegant and is implemented in the most effective way. No errors, warnings or suspicious notes are found in the log. Only the results are wrong! After an hour of thorough checks you recognise that somewhere in the middle a variable is being overwritten by a merge statement. Does this sound familiar to you?

After analysing my own experience and that of my colleagues I came to the conclusion that the small mistakes in the code are the most dangerous ones. Of course, global mistakes have a greater impact but they are usually easier to notice during self-checks or validation. However it is the small issues which are more likely to be missed and cause unpredictable results. Big mistakes mostly result from a lack of programming skills, negligence or from misunderstanding data specifics, while small ones have a completely different origin. Statistical programming, just as any other type of programming, can be considered as communicating your thoughts to the programming environment using a certain language. From this point of view small mistakes are caused by a lack of fluency in the language. Have you ever found it difficult to express yourself with the required nuance when speaking a foreign language? Without proper skills the general idea might still be clear but the shades can be completely different and the overall impression can become unpredictably biased. The same in programming, SAS does what you tell it to do but in some cases the problem is that your expectation of what the code will do is different.

In my article I offer the audience to take part in a SAS test which consists of several questions based on real life programming situations. It is designed to help you gain a better overview of how well you speak SAS. Do you know where the dangers lurk? The questions are all fairly simple but the most obvious answer is not necessarily the correct one. Each question will be followed by an explanation and if necessary a way to avoid the potential mistake. You will earn points for correct answers and as in every language test at the end you can calculate a grade which estimates your level of fluency.

QUESTION 1

I decided to start with a classic example which actually gave me the idea for this whole paper. Suppose you have two datasets data1 and data2 as follows and submit the code as written below:

data1

I	J
1	1
2	2

data2

I
1
1
2
2

```
data data3;
    merge data1 data2;
    by I;
    if I=1 then J=J+2;
run;
```

How many records are there in dataset data3 where J=3?

a) 2
b) 1
c) 0
d) This code produces an error message because datasets are not sorted properly

The correct answer here is *b*. The MERGE statement forces the input variables to be retained. First all the input variables values are set to missing at the beginning of each BY group, and then they are retained until being

overwritten by new values which are input from one of the datasets. Therefore variable J is retained, and the output dataset looks as follows:

data3

I	J
1	3
1	5
2	2
2	2

To avoid this it's recommended to create a new variable as for example in the code below:

```
data data3;
    merge data1 data2;
    by I;
    if I=1 then K=J+2;
    else K=J;
run;
```

Now data3 contains a variable k that looks exactly as required:

data3

I	J	K
1	1	3
1	1	3
2	2	2
2	2	2

QUESTION 2

When I was first learning SAS, the MERGE statement seemed much trickier to me than the SET statement. I thought:

"What issues can a simple SET possibly cause?" One day later I changed my mind. This time let's make a set of the same datasets:

data1

I	J
1	1
2	2

data2

I
1
1
2
2

```
data data3;
    set data1 data2;
    if I=1 then J=3;
run;
```

How many records are there in dataset data3 where J=3?

a) 3
b) 1
c) 5
d) This code produces an error message because dataset data2 doesn't contain the variable J

This question contains exactly the same trap as the previous one. Though the log is clean, variable J is automatically retained, so for records from data2 where I=2, J also equals 3 and the correct answer here is *c*. Did you see that one coming? Here is how the new dataset looks:

data3

I	J
1	3
2	2
1	3
1	3
2	3
2	3

If this is not the kind of results you were hoping for, a way to avoid this is to add a new variable:

```
data data3;
    set data1 data2;
    if I=1 then K=3;
run;
```

data3

I	J	K
1	1	3
2	2	.
1	.	3
1	.	3
2	.	.
2	.	.

Now the new variable K equals 3 only if the required condition is fulfilled.

QUESTION 3

The OUTPUT statement is not an exception when it comes to hidden threats. Let's take a look at one of them. Suppose the following code is submitted**:**

```
data data1;
    I=1;
    J=0;
    if I=1 then J=1;
    output;
    I=2;
    if I=1 then J=1;
    output;
run;
```

How many records are there in data1 where J=1?

a) 1
b) 2
c) 0
d) This code produces an error message because there is no set or merge statement

Similarly to the two previous questions this one concentrates on how the program data vector works. The OUTPUT statement tells SAS to write the current observation from the PDV to a dataset immediately. Before the first OUTPUT statement, variable J is set to 1 and it is never overwritten. So the correct answer here is *b*. Here is what we get with the code above:

data1

I	J
1	1
2	1

To avoid this issue you will need to make sure J is assigned the required value before each OUTPUT statement:

```
data data1;
    I=1;
    J=0;
    if I=1 then J=1;
    output;
    I=2;
    if I=1 then J=1;
    else J=0;
    output;
run;
```

The below shows that the updated code provides the required result:

data1

I	J
1	1
2	0

QUESTION 4

I think it is time to move from the program data vector to something else. Suppose the following code is submitted:

```
data data1;
    length STRING1 $250;
    STRING1=repeat('A',210)||repeat('B',5);
    STRING2=tranwrd(STRING1,'B','C');
    C=find(STRING2,'C');
run;
```

What value does the variable C have?

a) 209
b) 210
c) 211
d) 0

Though there are six letter 'B's in the variable STRING1, the TRANWRD function defaults the result to a length of $200 so STRING2 does not contain any 'B's and the correct answer here is *d*. To avoid confusion do not forget to specify lengths for character variables you are using in your program.

QUESTION 5

Is BEST. always the best? Let's figure it out using the code below:

```
data data1;
    format X Y 13.;
    X = input("1234567890123",best.);
    Y = input("1234567890123",13.);
run;
```

Are variables X and Y equal?

a) Yes
b) No
c) Hard to tell, the numbers are too big
d) SAS cannot deal with such big numbers and an error message is written to the log

As you have probably guessed from the way the question was asked the correct answer is *b*. The default width for the BEST. informat is 12 and the INPUT function will just return the first 12 digits: 123456789012. At the same time 13. provides the correct result. I included this question to show how important it is to double check format lengths and how easy it is to miss a mistake caused by using the wrong ones.

QUESTION 6

Forgetting to use a semicolon accounts for probably 75% of all SAS error messages in the log. Two semicolons are twice as good as one, or are they? Suppose you are given the code below:

```
data data1;
    I=1;
    if I=1 then J=2;;
    else J=1;
run;
```

What value will be stored in the variable J if you submit it?

a) 1
b) 2
c) Missing
d) This code produces an error message and the dataset is empty

As you have probably noticed almost every question contains an answer *d* with an error message written to the log. In this case it is not added to cover all possible scenarios but because it is the correct one. SAS interprets the second semicolon as an empty statement and therefore the connection between IF and ELSE is lost and you get an error message: No matching IF-THEN clause.

QUESTION 7

Although there is nothing more natural than an IF-THEN construction as we deal with it both in our programming and non-programming lives it can also surprise us sometimes with unexpected results. Suppose you have a dataset data1 and submit the code as below:

```
data data1;
   set data1;
   if not I=1 then J=1;
run;
```

data1

I
0
1
2
3

How many records are there in data1 where J=1?

a) 1
b) 0
c) 3
d) The code produces an error message because parentheses are missing for the condition

As answer d suggests the main problem is in missing parentheses for I=1. However this code does not give us an error message but actually sets J to 1 for some records in the dataset. With no parentheses the condition is interpreted as NOT(I)=1 and for one of the records in the dataset this condition is true. If I=0 then NOT(I)=1 and J is set to 1 so the correct answer here is *a*.

QUESTION 8

Missing the issue described in this question once cost me an hour of debugging so I decided to include it in my paper. The following code is submitted:

```
%let B=3;
%let A=%eval(&B/2);
```

What value is stored in macro variable A?

a) 1.5
b) An error message is generated
c) 1
d) No error message is generated but the macro variable is empty.

The correct answer is c though *a* seems to be the most obvious one. With the %EVAL function, if a division operation results in a fraction, the fraction is truncated to an integer. You will not receive any error messages or suspicious notes but if 1.5 is the value that is needed for the programming steps which come next, your code will not work as expected. To obtain the actual value of the division you should use the %SYSEVALF function.

QUESTION 9

The NOPRINT option is sometimes very helpful as it helps to avoid unnecessary information being printed in the output and it looks completely harmless, doesn't it? Let's submit the two pieces of code on the dataset data1:

data1

I
0
1
2
3

```
proc sql noprint;
    select * from data1;
    %put {&SQLOBS};
quit;

proc sql;
    select * from data1;
    %put {&SQLOBS};
quit;
```

Are the messages put to the log with %put the same?

a) Yes
b) No, the first piece of code puts nothing in the log
c) No, they put different numbers in the log
d) No, in the first case macro variable &SQLOBS is uninitialized and this results in a warning

When my colleague showed me this example I first thought it was a bug, however after taking a closer look at the information about PROC SQL automatic macro variables everything became clear to me. When the NOPRINT option is specified and no output table, single macro variable, macro variable list, or macro variable range is created, SQLOBS macro variable is set to 1. So the first piece of code puts 1 to the log while the second one predictably puts 4, so answer *c* is correct.

QUESTION 10

This question has already become a classic but I decided to include it with a slight modification which I hope will make it a little more interesting. Let's try to guess what happens if the below code is submitted:

```
data data1;
    X=0.5; Y=0.3;
    if X+Y=0.8 then FLAG=1;
    else FLAG=0;
    output;
    X=0.7; Y=0.1;
    if X+Y=0.8 then FLAG=1;
    else FLAG=0;
    output;
run;
```

What values are stored in flag in the first and second records?

a) 1 for both records
b) 1 for the first record and 0 for the second
c) 0 for the first record and 1 for the second
d) 0 for both records

I have come across quite a lot of papers about 0.1+0.2 not being equal to 0.3. Keeping this in mind my first expectation was that either FLAG is 1 for both records or 0 for both records. However, I was wrong, so let's take a look at the 64-bit representation of 0.8, 0.5 and 0.3 using the BINARY64. format:

0.5= 0|01111111110|0000000000000000000000000000000000000000000000000000

0.3= 0|01111111101|0011001100110011001100110011001100110011001100110011

0.8= 0|01111111110|1001100110011001100110011001100110011001100110011010

I inserted the split symbol '|' to separate the sign, exponent and mantissa. To add the first two numbers I first need them to have the same exponent: 01111111110. This way the mantissa for 0.3 will now look like:

0.1001100110011001100110011001100110011001100110011001100110011

I shift it and take the hidden bit into account. Now the mantissas can be added and the result looks like:

1.1001100110011001100110011001100110011001100110011001100110011

However only 52 bits are available for the mantissa so rounding needs to be applied:

1.1001100110011001100110011001100110011001100110011001100110010

So 0.5+0.3 now looks like:

0|01111111110|1001100110011001100110011001100110011001100110011001100110010

which is exactly how 0.8 looks in 64-bit representation so for the first record FLAG is set to 1.Now let's take a look at the second record:

0.7=0|01111111110|0110011001100110011001100110011001100110011001100110011001100110

0.1=0|01111111011|1001100110011001100110011001100110011001100110011001100110011010

After performing the same set of calculations I get that the mantissa for 0.7+0.1 which looks like:

1.1001100110011001100110011001100110011001100110011001100110011001

And the whole value is represented as below:

0.7+0.1=0|01111111110|1001100110011001100110011001100110011001100110011001100110011001

As you can see this result differs from the representation of 0.8 in last two digits and that's why the FLAG is set to 0 in the second case, so the correct answer is *b*.

QUESTION 11

I know that eleven questions is not a typical number for a test but I couldn't choose which one to exclude so I decided to include all that I had. I suggest you to take a look at the code below and try to predict what results it will produce:

```
data data1;
    X=put(-1,hex2.);
    Y=input(X,hex2.);
run;
```

What value is stored in Y?

a) -1
b) 255
c) Missing
d) The code produces an error message because HEX2. format cannot deal with negative values.

Answer *d* is not correct. HEX2. can deal with negative values but it handles them in an interesting way. It writes negative numbers in two's complement notation, so -1 turns to FF. At the same time the informat HEX2. converts the input value to positive integer binary values, assuming all input values are positive, so FF turns to 255 and the correct answer is *b*. To avoid this SAS has format and informat HEX16. Format HEX16. displays floating-point values in their hexadecimal form and the informat HEX16. converts the input value to real binary (floating-point) value, and is also working with negative values. So if we use HEX16. dataset data1 looks as below:

```
data data1;
    X=put(-1,hex16.);
    Y=input(X,hex16.);
run;
```

data1

X	Y
BFF0000000000000	-1

CALCULATING YOUR FINAL MARK

As you remember, you were supposed to earn points in this paper. Now it is time to sum them up.
If you have 1-3 point this means that your SAS level is elementary. You understand the basics but when it comes to nuances there is still work to be done.
4-6 points is a pretty good result. I would call this intermediate. You still have some gaps but I am sure that with a certain effort you will fill them in. Hopefully, my paper already helped you to identify areas where you can improve your skills.
If you have earned 7-9 points your equivalent level is upper-intermediate and this means that you are an experienced programmer who knows where to be on the look-out in order not to make a mistake.
10-11 points is an excellent result which means you are advanced in SAS. Your colleagues look up to you and seek your advice. Please accept my congratulations.

CONCLUSION

In my paper I arranged the questions by topics to help the readers identify areas where they might need to deepen their knowledge. I hope it was helpful for those willing to improve their SAS knowledge. While working on it I asked my colleagues to let me know if they face a situation when SAS works in an unexpected way and then I tried to figure out why this happened so I also cleared some things for myself.

REFERENCES

C. Thacher, Why .1 + .1 Might Not Equal .2 and Other Pitfalls of Floating-Point Arithmetic, SAS Institute Inc., Cary, NC.

ACKNOWLEDGMENTS

Thank you to Rowland Hale and Anna Asadskaya for reviewing this paper and to Christoph Baumer, Igor Khorlo and Vladlen Ivanushkin for suggesting examples.

CONTACT INFORMATION

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

Yuliia Bahatska
inVentiv Health Clinical
Joachimstaler Str. 10–12,
10719 / Berlin, Germany
Work Phone: +493034506914
Email: yuliia.bahatska@inventivhealth.com