PharmaSUG 2019 - Paper BP-289

Let Your Log Do the Work for You

Yuliia Bahatska, PRA Health Sciences;
Vladlen Ivanushkin, DataFocus GmbH;

ABSTRACT

Of course, there is no magic trick that will completely free you from writing the code, at least the authors of this paper are not aware of such. However, the tips we suggest can help the programmers to avoid struggling through technical details and let them concentrate on more sophisticated tasks which require deep understanding of the subtleties of programming. In this paper we will show different scenarios where the information such as ODS table names or template specifics can be put into the log or read from the log by a couple of lines of code. This means that you can use this part of your brain's CPU to learn something else.

INTRODUCTION

Robert Heinlein once said: "Progress is made by lazy men looking for easier ways to do things.", and we completely agree with this. We believe that a good programmer is a lazy programmer, and in this paper, we would like to share with you how SAS® functionality can be used in order to minimize the amount information you need to learn by heart or derive.

ODS TRACE ON

It is impossible or at least very time consuming to memorize all the ODS table names for all the procedures and the good news is that there is no need for this. Instead of learning the names by heart or referring to the documentation every time you can see all the outputs created by your procedure by adding just three words of code before the procedure itself:

```
ods trace on;

proc means data=data1;
   var variable1;
run;
```

This statement writes to the log the information for each object created by your procedure. This information contains output name, label, template and path:

```
Output Added:
-------------
Name:       Summary
Label:      Summary statistics
Template:   base.summary
Path:       Means.Summary
-------------
```

Output 1: Information about an object

This statement is particularly helpful if you are using some procedure you are not very familiar with or you are not sure where to look for the results you need. If you are no longer interested in the information about the objects created, you can turn off the writing of a trace record using the code:

```
ods trace off;
```

INFORMATION ABOUT THE STYLES

It sometimes happens that the programmers need to know some specific details of the style they are using or for some reason they need to get an idea about all the available default styles at the respective

source. In order to do this one just needs to run the PROC TEMPLATE with the SOURCE statement:

```
proc template;
      source styles.default;
run;
```

This is how the output in the log may look like in this case. All available default styles are listed, and in the log we have a ready-to run piece of code:

```
29         proc template;
30             source styles.default;
define style Styles.Default;
   class fonts
      "Fonts used in the default style" /
      'TitleFont2' = ("<sans-serif>, Helvetica, sans-serif",4,bold italic)
      'TitleFont' = ("<sans-serif>, Helvetica, sans-serif",5,bold italic)
      'StrongFont' = ("<sans-serif>, Helvetica, sans-serif",4,bold)
      …
      class GraphFonts
      "Fonts used in graph styles" /
      'GraphDataFont' = ("<sans-serif>, <MTsans-serif>",7pt)
      'GraphUnicodeFont' = ("<MTsans-serif-unicode>",9pt)
      'GraphValueFont' = ("<sans-serif>, <MTsans-serif>",9pt)
      'GraphLabel2Font' = ("<sans-serif>, <MTsans-serif>",10pt)
       …
end;
31         run;
```

Output 2: Information about styles

In case you are interested in some specific style, similar code can be used:

```
proc template;
    source Base.Freq.OneWayFreqs;
run;
```

And the output produced will look like:

```
2          proc template;
3              source Base.Freq.OneWayFreqs;
define table Base.Freq.OneWayFreqs;
   notes "One-Way Frequency table";
   dynamic pfoot footnote pfoot2 fMissing;
   footer f1 f2;

   define f1;
      text footnote;
      print = pfoot;
      spill_margin;
   end;

   define f2;
      text "Frequency Missing = " fMissing -12.99;
      print = pfoot2;
   end;
   parent = Base.Freq.OneWayList;
end;
```

Output 3: Information about the styles

But how would one know what should be used in the SOURCE statement? It doesn't look like something to be learned by heart. If you take a look at the log output of the previous example with the ODS TRACE ON, there is a line called Template. And this is exactly the name to be used in the SOURCE statement to get information about interested style.

TMPLOUT OPTION

Graphical Template Language or the GTL allows programmers to control every aspect of their figures. However, it is not that easy to learn it from scratch as the code might look cumbersome in comparison to the SG-procedures. If you don't feel confident with the GTL you can start by using an SG-procedure and outputting the template from it to the file so that it can be further modified according to your needs. The TMPLOUT option allows users to create a file with ready-to run PROC TEMPLATE:

```
proc sgplot data=sashelp.class tmplout="&path.\&file..sas";
    title "Basic Plot";
    scatter y=age x=height/ group=sex;
run;
```

All you need to do now is to copy the PROC TEMPLATE from the file and make all the changes you need.

```
proc template;
    define statgraph sgplot;
    begingraph /;
          EntryTitle "Simple Scatter Plot" /;
          layout overlay / yaxisopts=(labelFitPolicy=Split)
          y2axisopts=(labelFitPolicy=Split);
                ScatterPlot X='Height'n Y='Weight'n / primary=true
                LegendLabel="Weight" NAME="SCATTER";
          endlayout;
    endgraph;
    end;
 run;
```

Output 4: Template from SGplot

If you still need to output the template to the log, this is also possible. The code below illustrates how the created file can be read in and further output to the log:

```
data _null_;
    infile "&path.\&file..sas" length=len;
    length text $200;
    input text $varying. len;
    put text;
run;
```

We always recommend adhering to the best programming practice and leaving your session unchanged, so it might be useful to delete the file afterwards:

```
data _null_;
    fname="tempfile";
    rc=filename(fname,"&path.\&file..sas");
    if rc = 0 and fexist(fname) then rc=fdelete(fname);
    rc=filename(fname);
run;
```

ADJUSTING AND FORMATTING LOG MESSAGES

Probably there is no programmer who hasn't ever written a single NOTE, WARNING or ERROR message to the log. We usually write log messages for various purposes: debugging, data checking and cleaning,

data issues identifying, providing information about macro execution etc. However, sometimes the log messages might be quite cumbersome and/or not really well readable. In this case don't forget that there are special symbols which can help manage appearance of your log messages – "/" and "@". These two symbols can make a big difference.

Let's consider the following situation – we want to use SASHELP.CARS data set and put to the log all messages with Make, Model, Type and Origin where Invoice is more than 100k and Make, Model, Origin and Invoice for all cars with more than 450 horse powers:

```
data log_messages;
    set sashelp.cars;
    if invoice>100000 then put "WARNING: The following cars cost more than
                                        100k: " make= model= type= origin=;
    if horsepower>450 then put "WARNING: The following cars have more than
                          450 horse powers: " make= model= origin= invoice=;
run;
```

As expected, we get the following messages in our log (line size=100):

```
WARNING: The following cars are Hybrids: Make=Dodge Model=Viper SRT-10 convertible 2dr Origin=USA
Invoice=$74,451
WARNING: The following cars cost more than 100k: Make=Mercedes-Benz Model=CL600 2dr Type=Sedan
Origin=Europe
WARNING: The following cars are Hybrids: Make=Mercedes-Benz Model=CL600 2dr Origin=Europe
Invoice=$119,600
WARNING: The following cars cost more than 100k: Make=Mercedes-Benz Model=SL55 AMG 2dr Type=Sports
Origin=Europe
WARNING: The following cars are Hybrids: Make=Mercedes-Benz Model=SL55 AMG 2dr Origin=Europe
Invoice=$113,388
WARNING: The following cars cost more than 100k: Make=Mercedes-Benz Model=SL600 convertible 2dr
Type=Sports Origin=Europe
WARNING: The following cars are Hybrids: Make=Mercedes-Benz Model=SL600 convertible 2dr
Origin=Europe Invoice=$117,854
WARNING: The following cars cost more than 100k: Make=Porsche Model=911 GT2 2dr Type=Sports
Origin=Europe
WARNING: The following cars are Hybrids: Make=Porsche Model=911 GT2 2dr Origin=Europe
Invoice=$173,560
```

Output 5: Unformatted log output

Although there are not that many messages and they are not extremely long, it is already difficult to read and understand them. If there are too many different message types in the log, it may also become difficult to distinguish between them. The warning messages aren't grouped together, and all the triggered checks are written one after another. In such situation it may be beneficial to combine all cases of interest for each warning type. Also, with some minor updates it is possible to make the WARNING messages significantly easier to read:

```
data log_messages;
    length warning_message1 warning_message2 $1000;
    set sashelp.cars end=eof;
    retain warning_message1 warning_message2;
    /*Collect all cases for invoice>100000*/
    if invoice>100000 then do;
        if missing(warning_message1) then warning_message1=
                    "'WARNING: The following cars cost more than 100k:";
        else warning_message1=catt(warning_message1,
          "'/@10'Make="||strip(make) || "'@30'Model=" ||strip(model)||
         "'@59'Type="  ||strip(model)|| "'@87'Origin="||strip(origin));
    end;
```

```
    /*Collect all cases for horsepower>450*/
    if horsepower>450 then do;
        if missing(warning_message2) then warning_message2=
          "'WARNING: The following cars have more than 450 horse powers:";
        else warning_message2=catt(warning_message2,
          "'/@10'Make="  ||strip(make)   ||"'@30'Model=" ||strip(model)||
          "'@59'invoice="||strip(invoice)||"'@87'Origin="||strip(origin));
    end;
    /*Put each consolidated warning text into a macro variable*/
    if eof then do;
        call symputx("warning_message1", catt(warning_message1, "'/;"));
        call symputx("warning_message2", catt(warning_message2, "';"));
    end;
run;

/*Print the messages*/
data _null_;
    putlog &warning_message1;
    putlog &warning_message2;
run;
```

The screenshot below shows the log produced by the code.

```
WARNING: The following cars cost more than 100k:
            Make=Mercedes-Benz  Model=SL55 AMG 2dr           Type=SL55 AMG 2dr           Origin=Europe
            Make=Mercedes-Benz  Model=SL600 convertible 2dr  Type=SL600 convertible 2dr  Origin=Europe
            Make=Porsche        Model=911 GT2 2dr            Type=911 GT2 2dr            Origin=Europe

WARNING: The following cars have more than 450 horse powers:
            Make=Mercedes-Benz  Model=CL600 2dr              invoice=119600              Origin=Europe
            Make=Mercedes-Benz  Model=SL55 AMG 2dr           invoice=113388              Origin=Europe
            Make=Mercedes-Benz  Model=SL600 convertible 2dr  invoice=117854              Origin=Europe
            Make=Porsche        Model=911 GT2 2dr            invoice=173560              Origin=Europe
```

Output 6: Formatted log output

Obviously, such grouped warnings are much better to read and understand, even though producing them requires additional effort and time. Our point of view is that it is still worth it if the content of such messages is of high importance, for example, in case of programming edit checks for identifying data issues or inconsistencies.

PARSING LOG

While previous examples illustrated putting something to the log, the following ones are more about getting something from it. Most of the companies have a tool to check the log. The main idea of such tool is to provide a summary of all the suspicious log messages, if there are any. When the users see that suspicious messages are present in their log, they go to the log, find these messages and try to fix them. In our opinion this process can be optimized. Below is the schematic version of the macro that would not only identify the suspicious messages in your log, but also provide a report that contains blocks of code where such messages have occurred. For example, if the error occurred in a DATA STEP, the whole DATA STEP will be present in the report. This macro is not a ready-to-use tool, but rather provides the users with a general idea of how debugging the program can be made easier:

```
%macro check_log(file=, notes='');
    /*Read in log file*/
    data log_input;
        infile "&file" length=len missover;
        length text $200;
        input text $varying. len;
```

```
    if ^missing(text) then do;
        row+1;
        output;
    end;
run;

proc sort data=log_input;
    by descending row;
run;

data log_suspmess;
    set log_input;
    retain block 0 code 0;
    /*Identify suspicious messages and the block of code they are
      caused by*/
    if block=1 and code=1 and not anydigit(substr(text,1,1)) then do;
        block=0;
        code=0;
    end;
    if find(text,'WARNING') or find(text,'ERROR')
       or find(text,"NOTE: MERGE statement has more than one data set
                        with repeats of BY values")
       /*In a live version there should be much more notes*/
       or text in (&notes) then block=1;
    if block=1 and anydigit(substr(text,1,1)) then code=1;
run;

/*Prepare the Output*/
data log_blocks(drop=prev_block);
    set log_suspmess;
    prev_block=lag(block);
    if prev_block=0 and block=1 then do;
        output;
        text='End of Suspicious block';
        row=row+0.5;
        output;
    end;
    else output;
run;


proc sort data=log_blocks;
    by row;
run;

data log_forout;
    set log_blocks;
    prev_block=lag(block);
    if prev_block=0 and block=1 then do;
        output;
        text='Start of Suspicious block';
        row=row-0.5;
        output;
    end;
    else output;
run;
```

```
    proc sort data=log_forout;
        by row;
    run;

    /*Print the output*/
    proc print data=log_forout(where=(block=1));
        var text;
    run;
%mend;
```

For example, in the code below the first and the third DATA STEPs are fine but the second one will produce an error message:

```
data a;
run;

data c;
    set a;
    where not missing(var);
run;

data b;
run;
```

Below is the piece of log of interest:

```
32          data c;
33              set a;
34              where not missing(var);
ERROR: Variable var is not on file WORK.A.
35          run;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.C may be incomplete.  When this step was
stopped there were 0 observations and 0 variables.
WARNING: Data set WORK.C was not replaced because this step was stopped.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

Output 7: Block of log with suspicious messages

So, this is how the output from the macro will look like:

Results of Log check

text
Start of Suspicious block
33
34 data c;
35 set a;
36 where not missing(var);
ERROR: Variable var is not on file WORK.A.
37 run;
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.C may be incomplete. When this step was stopped there were 0 observations and 0 variables.
End of Suspicious block

Page Break

Output 8: Results of the log check

PARSING LOG

SAS log is a paramount part of our programming activities. It is almost impossible to write a program (a good one for sure) without looking into the log. Still there are cases when a programmer could automate some things. Often, we create derived data sets from RAW data. Sometimes CRF forms are cumbersome and keeping track of all the fields (i.e. variables) becomes a challenge. Having written a mapping program many people ask themselves: 'Haven't I forgotten something? Have I mapped all the variables I had in my CRF form? Have any new variables recently been added to the CRF form?'. One could still go case by case and check each field, however the lazy ones will probably look for some better option. Such an option could be parsing the log from your mapping program.

Let's again use the SASHELP.CARS data set. Here is a part of log which is of interest:

```
6    data mapping;
7        set sashelp.cars;
8        call missing(of Make Model Type  DriveTrain MSRP Invoice
EngineSize Cylinders
9             Horsepower MPG_City /*this is a comment*/ Weight Wheelbase
/*one more comment*/ Length);
10       /*    Origin    MPG_Highway*/
11       /*    Origin
12           line
13           one more line
14           MPG_Highway*/
15        length MPG_Highway_dummy $200;
16   run;

NOTE: Variable MPG_Highway_dummy is uninitialized.
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.MAPPING has 428 observations and 16 variables.
NOTE: DATA statement used (Total process time):
      real time           0.05 seconds
      cpu time            0.00 seconds

17
18   proc sort data=mapping;
19       by Origin;
20   run;

NOTE: There were 428 observations read from the data set WORK.MAPPING.
NOTE: The data set WORK.MAPPING has 428 observations and 16 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

Output 9: Log to check

The task now is to find out whether each variable from SASHELP.CARS is mentioned in the log at least once. The first step is to get the list of all variables which is always possible with the DICTIONARY tables.

So, a simple SQL query first:

```
proc sql;
    select distinct name into :vars_exist separated by ' '
           from dictionary.columns
           where libname='SASHELP' and memname='CARS';
quit;
```

Now let's read in the log and check each variable against the log text:

```
data log;
    infile "&outdir/&prog..log" length=len end=eof;
    length text $200;
    input text $varying. len;

   if prxmatch("/^\d/", text) then codeline=1;
    text_upd=text;

    retain comment_start;
    if find(text, '/*') then comment_start=1;
    if find(text, '*/') then comment_end=1;
    if comment_end then call missing(comment_start);

    /*Delete comment lines without start and/or end comment symbols*/
    if ^find(text, '/*') and comment_start
       then text_upd=prxchange("s/\s.*//", -1, text_upd);
    /*Delete all other comment parts from the log*/
    text_upd=prxchange("s/(\/\*)*([^(\*\/)]*)(?(1)(\*\/)*|(\*\/))//", -1,
                                                                  text_upd);
    retain &vars_exist;
    array ds_vars{*} &vars_exist;

    /*Check if each variable name mentions in the log at least once*/
    do i=1 to dim(ds_vars);
        if prxmatch("/\b"||vname(ds_vars{i})||"\b/", text_upd)
           and codeline then ds_vars{i}=1;
        /*If not - print to the log*/
        if eof and ^ds_vars{i} then
           put "WARNING: Variable hasn't been used - " ds_vars{i}=;
    end;
run;
```

The code obviously checks whether each variable name is mentioned at least once, regardless in which statement. Only comments are ignored, but not the ones which start just with an asterisk and end with a semicolon. The result of running such code is the following:

```
WARNING: Variable hasn't been used - MPG_Highway=.
```

Output 10: Warning of interest

The code identified the variable correctly, it is indeed not mentioned. There is a dummy mentioning of MPG_Highway_dummy but everything worked fine for this word and it wasn't falsely considered as if MPG_Highway was mapped, although you probably should avoid the 'uninitialized' note.

Of course, writing a program that would work for each single scenario will require much more effort and will not fit into just one section of the paper. But writing a relatively simple one could already work for many cases and give a general idea of possible problems.

CONCLUSION

They say that asking the right question is half the answer. Statistical programming is no exception to this rule. In this paper we provided some examples of the questions programmers can ask SAS in order to make their work more efficient.

CONTACT INFORMATION

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

Yuliia Bahatska
PRA Health Sciences
bahatskayuliia@prahs.com

Vladlen Ivanushkin
DataFocus GmbH
vladlen.ivanushkin@datafocus.de