PhUSE 2017

Paper IS07

Check your code! Control your data!

Kristina Zweier, Clinipace Worldwide, Eschborn, Germany

ABSTRACT

Beginner SAS programmers learn very early the importance of checking their program code for errors. Although this is understood in theory, practical implementation is sometimes difficult due to lack of knowledge regarding how and when it should be done and lack of experience with the peculiarities of SAS. This paper presents options for program checking by analyzing selected SAS statements and procedures and highlights common mistakes beginners make when using them. This paper provides practical tips for beginners including suggestions for reviewing programs and data.

INTRODUCTION

Generating programming checks, avoiding common and tricky programming mistakes and gaining a reliable "feeling" for data is a skill set that is developed through years of experience with programming and data handling. At the beginning of a career in programming though, quality control and troubleshooting can be a source of frustration that often lets beginners struggle to find a helpful approach to tackle these tasks. If you don't know the possible pitfalls of SAS statements or procedures, you may not find the error in an incorrect output. If you don't know what kind of data errors are possible, you will not be able to screen for them while programming. Simply put, searching for something can be quite difficult if you don't know exactly what you are looking for.

The goal of this paper is to provide an orientation for beginners in data programming by offering a general guideline, practical programming tips and example program code as well as literature recommendations for a deeper understanding of the specific topics.

KEEP YOUR PROGRAMS CLOSE BUT YOUR DATA CLOSER

The solid foundation for correct data handling and generating outputs of high quality is laid before the programming has started. The characteristics of the data you're working with will affect the effort you need to produce your output and thus the effort you need to ensure the quality of your output. Therefore, you should take your time to get to know your data and their attributes:

- Content: what is your data about? – Dealing with demographic data is not the same as dealing with lab data and you would need to focus on different things to ensure correctness and quality of your respective outputs.
- Quantity: how many observations and variables does your data set contain? Are the data condensed in one data set or are they distributed over several data sets? – A small set of data reduces the need of programming checks, large data sets increase it.
- Quality: do you work with cleaned or uncleaned data? Do you expect data issues to get resolved in the future or do you have to develop a work-around? – Your day-to-day programming would be easier and your programs shorter if you could count on the correctness of the data you're receiving. Unfortunately, this is not always the case and you have to consider data issues and ways to solve them.
- Status: is your data complete or do you expect more data to follow? – Your program will look different if you have to keep in mind the possibility of additional observations or values compared to what your program would look like if you know that your data won't change anymore.
- Structure: how is your data set constructed? Are the data presented in a horizontal or vertical structure? Are they coded or written-out? – There are many orders and setups your data could come in which can either facilitate your programming task or make it, and thus quality control, harder.

Keeping these attributes in mind can be helpful to design a rough plan for how to get from your starting point (the data) to your goal (the respective output) before starting to write your program code, and for determining what steps are necessary regarding execution as well as quality control. This doesn't have to be specific but should serve as a common thread, guiding you through your programming task.

If you work with numeric data that can fluctuate over the period under observation (e.g. weight of subjects) it can be helpful to graphically display the data to get a rough idea about the data distribution and maybe to spot potential data errors. The program in the next chapter can help you with this task.

A PICTURE PAINTS A THOUSAND WORDS

Elaborate graphical displays of your data are often the end product of your work but it can also be nice to have a quick glance at the progression of your data over time. The program code attached in Appendix 1 of this paper can serve this

purpose as it produces a line plot of the development of selected parameters over defined time points.

The program code is constructed as a SAS macro and although the code itself might exceed the programming skills and knowledge of a beginner it is built in a way that allows for easy use of the macro for providing a general overview over data progression. The purposes of the individual program steps in the macro are explained through comments. For this reason it should be possible to understand the structure and execution of the code even if individual commands are unknown.

If you are unfamiliar with the SAS macro language it is recommended to read an introduction to the concept of macros, e.g. "SAS® Macro Programming for Beginners"[1] by Susan J. Slaughter and Lora D. Delwiche as an introduction of this topic at this point would exceed the set scope of this paper.
For the correct execution of the macro provided in Appendix 1 you have to define the parameters for the macro call. This will be explained based on an example further below. Your input data set should be arranged in a vertical order, i.e. the name of the parameter of interest should be stored in one variable and the numeric value of the parameter should be stored in another. Furthermore your data set should contain a numeric time point variable, e.g. visit days or the individual days of the considered period.

The line plot works best for a manageable number of values of the grouping variable (e.g. subjects in a clinical study). Therefore it would be advisable to split large data sets by meaningful classifications (e.g. study center or sex).
As an example let's take a look at a section of an ADaM analysis data set of vital signs (ADVS) that is reduced to the variables needed for correct execution of the macro:

SUBJID	ADY	AVISITN	PARAMCD	AVAL
0004	-3	1	HEIGHT	169,5
0004	-3	1	WEIGHT	69,2
0004	-3	1	PULSE	73
0004	20	2	HEIGHT	169,5
0004	20	2	WEIGHT	68,9
0004	20	2	PULSE	75

The first variable, SUBJID, holds the ID numbers of the different subjects in this data set. The variable ADY contains the individual days of a clinical study relative to a defined starting date. The following variable named AVISITN stores the individual visit numbers of the visits at which the values of the parameters of interest were collected.

The variable PARAMCD contains the parameter code names. Finally, the variable AVAL stores the values of the parameters described in PARAMCD.

The macro will create a line plot that plots the values (value variable AVAL) for every parameter (parameter variable PARAMCD) against chosen time points (study day variable ADY or visit number variable AVISITN), grouped by a group variable (subject ID variable SUBJID). If you're only interested in one parameter, you can do a pre-selection which we'll do in our example with the parameter weight:

```
DATA advsgraph;
   SET advs;
   WHERE aval ne . and paramcd = "WEIGHT";
RUN;
```

Now that we've specified the input data set we can run the macro. The beginning of the macro code in Appendix 1 shows the required parameters:

```
%MACRO lplot      (  /* MP = Mandatory parameter                          */
                  in = /* MP Name of input data set                        */
          ,_testvar = /* MP Name of parameter name variable               */
              ,aval = /* MP Name of numeric parameter value variable      */
             ,visit = /* MP Name of chosen numeric date variable (e.g.
                         ADY, AVISITN)                                    */
          ,groupvar = /* MP Name of grouping variable (e.g. SUBJID,USUBJID)*/
           ,outpath = /* MP Name of output file pathname                  */
                    );
```

Applied to our data set ADVSGRAPH the macro call could look like this:

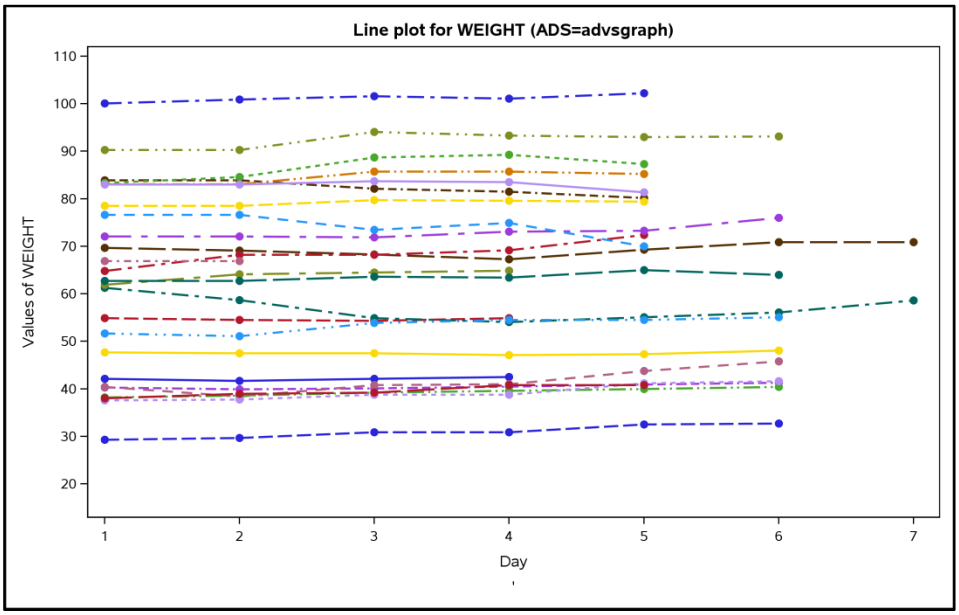
```
%lplot (in=advsgraph❶, _testvar = paramcd❷, aval = aval❸, visit = avisitn❹,
        groupvar = subjid❺,outpath = C:/SAS/mywork/out❻);
```

The macro is called with a percentage sign followed by the macro name. Then, the parameters are specified (in brackets). First, the input data set, which contains the data of interest, will be specified.❶ Next, the variable that contains the name(s) of the parameter(s) that will be displayed in the line plot is specified.❷ In the data set ADVSGRAPH the only

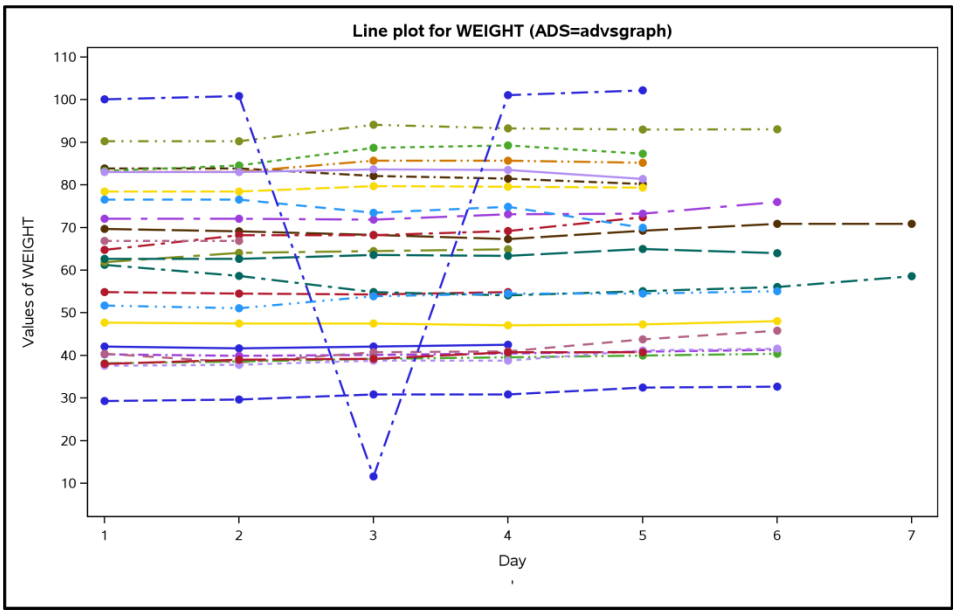
name that is stored in the parameter variable PARAMCD is “WEIGHT”, therefore only one plot will be produced. If this is not pre-specified, the macro will produce one plot for each parameter.

Next, the numeric variable with the values of the parameter(s) is specified.❸ Then the time point variable is chosen.❹ The data set contains two possible time point variables: ADY and AVISITN. For this line plot, the variable AVISITN is chosen. Next, the grouping variable is chosen.❺ That way the plot will show the progression of weight per subject ID. However, the individual subject ID will not be identified in the plot as it should only serve as a rough overview. Also, the line plot should not be crowded with information to maintain clarity.
Finally, the pathname for the output folder is defined.❻

This is the resulting image:



The line plot shows that there are no radical changes in the subjects’ weight across the time points. You see the range of the subjects’ weight with a minimum weight of about 30 kg (which suggests that children are included in this sample) and a maximum weight of about 100 kg. The plot can help you spot data entry errors that are easily seen when graphically displayed but that can be difficult to discover with queries:



In this scenario the weight for a subject for the third visit was entered as 11 kg instead of 101 kg. These kinds of errors

are often overlooked in the data cleaning process.
After you have gotten familiar with your data you can focus on your program code. In the following sections, this paper provides tips on how to produce solid programming code that results in quality output.

THE SOURCE OF ALL THAT IS GOOD – GPP

There are many reasons to apply Good Programming Practice (GPP) to your daily programming work as it helps create correct, clear and efficient programming code. Furthermore, GPP is the basis for successful error checking. As typing errors and other careless mistakes occur more or less regularly, it is important to develop a habit of writing clear, readable program code to make the spotting of those errors easier. The foundation of producing clear and informative program code is the usage of meaningful names for data sets and variables, indentation and comments.

A lot of companies will have their own set of custom GPP rules to follow. Additionally, it is recommended to visit the PhUSE Wiki webpage about Good Programming Practice Guidance to read more about GPP[2]. There you'll also find examples of good and bad programming on that site.

CHECK YOUR LOG, FILL YOUR LOG, EXPAND YOUR QUERY CATALOG

An essential part of GPP is to check the log file to make sure that the program has executed correctly. The log file documents everything you do in a SAS session and helps you to identify problems with your program and to debug it.
You will find three different kinds of messages in the log file that give you information about your program code: errors, warnings and notes[2].

Errors usually occur with syntax or spelling mistakes that prevent your code from executing correctly.
Warnings can for example occur if you want to work with a variable that, for whatever reason, doesn't exist in your data set. Warnings don't hinder your program code from running but your output may very well be incorrect.
Notes won't necessarily mean that your program is incorrect. They provide information and it is up to the programmer to draw the right conclusions from the note. This makes them the trickiest of the messages because you should know what to expect running your different code segments.

For more information about the messages and about common programming mistakes which can lead to them being written into the log file please consult chapter 11 of "The little SAS® Book: A Primer, Fifth Edition"[3] from Lora D. Delwiche and Susan J. Slaughter. I will address avoiding programming mistakes in the next chapter but first I would like to suggest creating your own warnings and errors that you can write into the log file with the aid of the PUT statement.

Creating your own warnings and errors is a good way to familiarize yourself with your data and to ensure their quality on the one hand and to gain a better understanding of the mode of operation of SAS on the other. You can regard them as your own custom made queries.
The syntax of the command is fairly easy, which will be demonstrated with the SASHELP data set CLASS:

OBS	NAME	SEX	AGE	HEIGHT	WEIGHT
1	Alfred	M	14	69	112.5
2	Alice	F	13	56.5	84
3	Barbara	F	13	65.3	98
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83
7	Jane	F	12	59.8	84.5
8	Janet	F	15	62.5	112.5
9	Jeffrey	M	13	62.5	84
10	John	M	12	59	99.5
11	Joyce	F	11	51.3	50.5
12	Judy	F	14	64.3	90
13	Louise	F	12	56.3	77
14	Mary	F	15	66.5	112
15	Philip	M	16	72	150
16	Robert	M	12	64.8	128
17	Ronald	M	15	67	133
18	Thomas	M	11	57.5	85
19	William	M	15	66.5	112

Let's assume this data set shouldn't contain subjects that are younger than twelve years old and we want to be informed if this is the case. We would then use the PUT statement in a data step to generate a warning message in quotes:

```
/*Demonstration of a Warning message*/
DATA class;
   SET sashelp.class;
   IF age < 12 then PUT
   "WAR" "NING: subject " name "is below
   the age of twelve: "  /
   age=;
RUN;
```

This message is written into the log file:

```
WARNING: subject Joyce is below the age of 12:
AGE=11
WARNING: subject Thomas is below the age of 12:
AGE=11
```

Likewise, you can construct an error message instead of a warning the same way:

```
/*Demonstration of an Error message*/
DATA class;
   SET sashelp.class;
   IF age < 12 then PUT "ERR" "OR: subject " name "is below the age of twelve: "  /
   age=;
RUN;
```

This command would generate the following message:

```
ERROR: subject Joyce is below the age of twelve:
AGE=11
ERROR: subject Thomas is below the age of twelve:
AGE=11
```

As you can see, the custom warning and error messages are displayed in the same colors as SAS built-in ones— green for warnings and red for errors.
It is advisable to split the words “WARNING:” and “ERROR:” into two words. This has no impact on the correct display of the message in the log file but this method prevents the keywords “WARNING” and “ERROR” to be tagged and found while searching the log file for warnings and errors manually or automatically via program code. This way it is possible to ensure that only actual errors and warnings are displayed.

It is of interest which students are below the age of twelve and how old they are. Therefore, we include the variables that contain that information (“name” and “age”) into the custom warning message.
We split the warning message to include the variable “name”. As you can see in the log file output, the value of the variable is written out.
The slash after the message generates a return which lets the following information be displayed on the next line. We want the age of the concerned subjects to be displayed as well. By adding an equal sign after the age variable, not only will the value of the variable be written out, but also the name.

There is a broad range of possible situations in which it would be advisable to check on the accuracy of your data by developing custom warning and/or error messages. You can use them to:

- ensure chronological consistency (e.g. date of first medication should not be earlier than the date of study start)
- check if your variables exceed predefined maximum lengths
- check that your calculations are realistic
- ensure that you considered all possible values of variables if the correct execution of your program depends on them
- keep track of possible necessary program adaptations if you expect data changes, i.e. unexpected values
- keep track of unclean data

How, when and where you should use custom warnings and errors depends highly on your data and your program code. Make it a habit to use custom warnings and errors by including them when you use IF/THEN/ELSE statements in data steps. A good way of getting used to creating them is by regularly checking for missing or unexpected data.

Often you will write commands based on criteria the observations meet or don’t meet, e.g. assigning the attribute “obese” to subjects who exceed a certain weight. In some cases it is not important to consider all possible values of your critical variable, e.g. if you just want to flag all subjects of a specific country. But if the (correct) processing of all of your data depends on it, using custom warnings can help you avoid unpleasant surprises.

For the next example the SASHELP.CLASS data set was altered and there are now missing values for two subjects for the variable SEX. As this data set doesn’t contain a lot of observations, those missing values are rather easy to spot but with hundreds or thousands of observations this can get more difficult.

Let’s assume that we now want to split the data set based on the sex of the subjects. Our program code could look something like this:

```
/*Split data set based on sex*/
DATA girls boys;
   SET class;
     IF sex eq "F" then output girls;
     ELSE IF sex eq "M" then output boys;
RUN;
```

We wouldn’t find any warnings or errors in the log file because there is nothing wrong with our code. We would find a hint that something could have gone wrong though:

```
NOTE: There were 19 observations read from the data set WORK.CLASS.
NOTE: The data set WORK.GIRLS has 8 observations and 5 variables.
NOTE: The data set WORK.BOYS has 9 observations and 5 variables.
```

Examining those notes we can conclude that while the original data set CLASS contained 19 observations, the observations in the two new data sets sum up to only 17 observations.

It is fairly easy to avoid overlooking missing – as in this case – or unexpected data by adding custom warnings to the IF-THEN-ELSE statements:

```
/*Split data set based on sex*/
DATA girls boys;
   SET class;
     IF sex eq "F" then output girls;
     ELSE IF sex eq "M" then output boys;
     ELSE PUT "WAR" "NING: missing or unexpected value for SEX. " /
     name= sex=;
RUN;
```

These warnings will be written into the log file:

```
WARNING: missing or unexpected value for SEX.
NAME=Joyce SEX=
WARNING: missing or unexpected value for SEX.
NAME=Thomas SEX=
```

After we were made aware of the problematic data, we now can decide how to proceed and make adjustments to our program code.
For a more detailed explanation of custom warnings and errors and more examples for their implementations it is recommended to read “Best Practices: PUT More Errors and Warnings in My Log, Please!”[4] by Mary F. O. Rosenbloom and Kirk Paul Lafler.
So far we examined the need to get familiarized with source data, write clear program code, check the log file for errors, warnings and notes and to write custom ones if necessary. The last chapter of this paper discusses the probably most dangerous kind of program errors, namely those that produce clear log files but wrong results.

DO WHAT I WANT, NOT WHAT I TOLD YOU TO DO

As a beginner it is expectable that you will make mistakes while you are getting to know the SAS language and learning how SAS operates and processes data. If you are lucky, SAS catches on to your mistakes and you will be warned in the log file. But it’s also possible that you make a mistake that will produce a clear log file because your program code was not problematic per se, but you will still get wrong results because you either didn’t consider all possibilities or didn’t internalize SAS’ way of proceeding.

There are a lot of different scenarios in which you can produce erroneous results while simultaneously producing correct SAS code – just not the correct code for the specific goal at hand. This means basically that SAS did what you told SAS to do, but it wasn’t what you actually wanted it to do. These kind of errors can be dangerous if not discovered as they can lead to false results and therefore to false conclusions.

This chapter attempts to shine light on this problem by listing a few examples of this kind of erroneous programming. The goal is to instill an understanding of the importance of making sure that the written program code results in the anticipated output.

BE(A)WARE OF THE PROGRAM DATA VECTOR

If you haven’t familiarized yourself with the processing of a DATA STEP within SAS and the role of the Program Data Vector (PDV) yet, it is highly recommended to do that. There are excellent papers such as “Essentials of the Program Data Vector (PDV): Directing the Aim to Understanding the DATA Step!”[5] by Arthur Xuejun Li that explain in detail the individual steps that are executed during a DATA STEP.
In the scope of this paper the features of the PDV will be explained in the context of their impact on the SET and MERGE statements and the possible mistakes that can occur when using those statements.

One of the first things you’ll learn as a SAS programmer and will use constantly is how to join data sets by either the SET or the MERGE statement. Often you’ll want to concatenate data sets and edit variables in one step. If you do that, you’ll have to keep some things in mind, as demonstrated in the following example.

We have two data sets, TRT1 and TRT2, with similar data regarding information about treatment of subjects per time point and population flags for each observation. For a better understanding the variable labels are displayed instead of the names:

Obs	Subject Identifier	Planned Treatment	Actual Treatment	Time point (N)	ITT Population Flag	Safety Population Flag
1	1	1	1	201	Y	Y
2	1	1	1	202	Y	Y
3	1	1	1	203	Y	Y
4	2	2	.	201	Y	N
5	2	2	.	202	Y	N
6	2	2	.	203	Y	N

The data set is ordered by subject ID and individual time point for which a planned and received treatment is listed. Furthermore whether each subject meets the criteria to be assigned to various population groups is documented. Because a planned treatment was assigned to each subject, all subjects are assigned to the Intention-To-Treat (ITT) population. But because only the first subject actually received treatment, only this subject is assigned to the Safety population.

The second data set contains almost the same information for different subjects except for missing information about the actual treatment:

Obs	Subject Identifier	Planned Treatment	Time point (N)	ITT Population Flag	Safety Population Flag
1	101	1	201	Y	Y
2	101	1	202	Y	Y
3	101	1	203	Y	Y
4	102	2	201	Y	N
5	102	2	202	Y	N
6	102	2	203	Y	N

We now want to join these two data sets into one using the SET statement and assigning an actual treatment to the subjects from the second data set, TRT2, defining the planned treatment as actual treatment for those subjects who belong to the Safety population.

A seemingly logical, albeit wrong approach would look like this:

```
DATA treatment;
   SET trt1
       trt2 (IN=intrt2);

   *) Actual treatment will be set to planned treatment for the safety patients;
   IF intrt2 and saffl eq 'Y' THEN trta = trtp;
RUN;
```

In this DATA STEP a data set named TREATMENT is created by joining the data sets TRT1 and TRT2 using the SET statement. Using the data set option IN= behind the data set TRT2 in the SET statement, the Boolean variable INTRT2 is created with the value of 1 if the data set TRT2 contributed to the current observation. The IF-THEN statement checks if the individual observations are stored in data set TRT1 or TRT2 by applying the value of INTRT2: 1 for observations from TRT2 and 0 for observations from TRT1. If the individual observation is indeed from data set TRT2 and the Safety population flag is set to 'Y' (Yes) then the actual treatment is set to the value of the planned treatment.

This DATA STEP does not produce any errors or warnings in the log file:

```
NOTE: There were 6 observations read from the data set WORK.TRT1.
NOTE: There were 6 observations read from the data set WORK.TRT2.
NOTE: The data set WORK.TREATMENT has 12 observations and 6 variables.
```

The output data set looks like this:

Obs	Subject Identifier	Planned Treatment	Actual Treatment	Time point (N)	ITT Population Flag	Safety Population Flag
1	1	1	1	201	Y	Y
2	1	1	1	202	Y	Y

3	1	1	1	203	Y	Y
4	2	2	.	201	Y	N
5	2	2	.	202	Y	N
6	2	2	.	203	Y	N
7	101	1	1	201	Y	Y
8	101	1	1	202	Y	Y
9	101	1	1	203	Y	Y
10	102	2	1	201	Y	N
11	102	2	1	202	Y	N
12	102	2	1	203	Y	N

As you can see, the actual treatment variable is populated for subject 102 although this subject is not in the Safety population. The reason for this can be found in the modus operandi of the PDV.
The creation of a new data set through a DATA STEP comprises in two phases: compilation and execution. In the compilation phase, the PVD is created.
The PDV is a storage area in memory which contains the data in variables that are read in with the different available DATA STEP statements such as SET, MERGE or INPUT. At the end of a DATA STEP, the content of the PDV is written into the output data set.
SAS creates a data set one observation at a time, i.e. one row after the other. This ensues by executing the DATA STEP with all its statements for each observation that is to be created, starting with the DATA statement.
Before the first observation is created, the variables in the PDV are set to missing. Then SAS reads the input data into the PDV and executes any statements that are present in the DATA STEP.
After the DATA STEP executed for an observation reaches its end, SAS returns to the top of the DATA STEP and executes it for the next observation until all observations are created.

It is important to remember that while the variables in the PDV are set to missing for each iteration of the DATA STEP if the data are read in with an INPUT statement or assignment statements, they are **not** set to missing if the data is read with a SET or MERGE statement. In the latter cases, the values are automatically retained.[6]

This detail caused our example DATA STEP to produce incorrect output values. First, the observations from the first data set TRT1 are read into SAS. The variables stored in the PDV are set to missing, the DATA STEP executes for the first observation of the read in data set, the IF-THEN statement doesn't apply to the observation, the DATA STEP ends, the observation is put out into the new data set TREATMENT and SAS starts a new iteration of the DATA STEP for the second observation of the data set TRT1. The retained variable values are overwritten with the new information from the second observation and so on.

When the second data set TRT2 is read into SAS, the values from the first observation overwrite the retained values of data set TRT1 in the PDV for the variables the two input data sets share, i.e. all variables except for the actual treatment variable. This variable exists in TRT1 but not in TRT2 and therefore SAS assigns the retained value of the variable from the last observation of TRT1 to the first observation of TRT2. SAS then executes the IF-THEN statement because it applies to the observation which causes the value of the actual treatment variable to change to the value of the planned treatment variable of the same observation, i.e. the first observation of TRT2.

This principle continues until the first observation of subject 102 is read into SAS. As before, the retained values in the PDV are overwritten with the new input data except for the value of the actual treatment variable, which remains the value of the last observation and equals1. Now SAS checks the IF-THEN statement. Although the first part of the IF statement ("IF intrt2") applies to the observation, the second part ("and saffl eq 'Y'") does not and the THEN statement is not executed. Therefore, the retained value remains and is output with the new read in data for the observation. The error now applies to the next few observations until the new data set is created.

There are several ways to avoid this problem. You could, for example, rename the input variable of the actual treatment and create a new one that will be set to missing for each iteration because it is not an input variable:

```
DATA treatment;
   SET trt1 (RENAME= (trta=trta_old))
       trt2 (IN=intrt2);
   LABEL trta = "Actual Treatment";

   *) Actual treatment will be set to planned treatment for the safety patients
      of data set TRT2 and will be set to old Actual Treatment variable otherwise;
   IF intrt2 and saffl eq 'Y' THEN trta = trtp;
   ELSE trta = trta_old;
   DROP trta_old;
RUN;
```

Alternatively, you could use a DO statement to set the actual treatment to the planned treatment for the subjects of the Safety population and set the rest to missing:

```
DATA treatment;
   SET trt1
       trt2 (in=intrt2);

   *) Actual treatment will be set to Planned treatment for the safety patients
      and will be missing otherwise;
   IF intrt2 THEN DO;
      IF saffl eq 'Y' THEN trta = trtp;
      ELSE CALL missing (trta);
   END;

RUN;
```

Either way, the error will be corrected:

Obs	Subject Identifier	Planned Treatment	Actual Treatment	Time point (N)	ITT Population Flag	Safety Population Flag
1	1	1	1	201	Y	Y
2	1	1	1	202	Y	Y
3	1	1	1	203	Y	Y
4	2	2	.	201	Y	N
5	2	2	.	202	Y	N
6	2	2	.	203	Y	N
7	101	1	1	201	Y	Y
8	101	1	1	202	Y	Y
9	101	1	1	203	Y	Y
10	102	2	.	201	Y	N
11	102	2	.	202	Y	N
12	102	2	.	203	Y	N

The same principal of automatically retaining also applies to the MERGE statement.

MISSING VALUES AND THEIR INVISIBLE DAMAGE

One of the first tasks assigned to a beginner programmer is the generation of tables and listings with the help of the PROC REPORT procedure. This procedure offers the convenient possibility to group variables. Doing so, it is crucial not to disregard missing values as this can lead to an incomplete output.

This is demonstrated by the example of the data set “TREATMENT” created above.
Please note that a general understanding of the procedure is assumed as not every detail of PROC REPORT will be examined here. We want to create a listing based on the data set “TREATMENT” including all variables besides the Safety population flag. The program code could look something like this:

```
PROC REPORT DATA = treatment NOWD HEADLINE HEADSKIP;
   COLUMN subjid ittfl trtp trta tptn;

   DEFINE subjid / GROUP order WIDTH=15 FORMAT=3.;
   DEFINE ittfl  /             WIDTH=10 LEFT;
   DEFINE trtp   /             WIDTH=10 FORMAT=3.;
   DEFINE trta   /             WIDTH=10 FORMAT=3.;
   DEFINE tptn   /             WIDTH=13 FORMAT=5.;

   BREAK AFTER subjid / SKIP;
RUN;
```

As we don’t need the subjects ID to appear in every row, we group by the variable SUBJID. This results in the following output:

Subject Identifier	ITT Population Flag	Planned Treatment	Actual Treatment	Timepoint (N)
1	Y	1	1	201
	Y	1	1	202
	Y	1	1	203
2	Y	2		201
	Y	2		202
	Y	2		203
101	Y	1	1	201
	Y	1	1	202
	Y	1	1	203
102	Y	2		201
	Y	2		202
	Y	2		203

After we examine the output, we decide to group by more variables as the only variable with varying values for each observation is the time point variable. We change our program code:

```
PROC REPORT DATA = treatment NOWD HEADLINE HEADSKIP;
   COLUMN subjid ittfl trtp trta tptn;

   DEFINE subjid / GROUP order WIDTH=15 FORMAT=3.;
   DEFINE ittfl  / GROUP order WIDTH=10 LEFT;
   DEFINE trtp   / GROUP order WIDTH=10 FORMAT=3.;
   DEFINE trta   / GROUP order WIDTH=10 FORMAT=3.;
   DEFINE tptn   /             WIDTH=13 FORMAT=5.;

   BREAK AFTER subjid / SKIP;
RUN;
```

The procedure runs without any complications, no warnings or errors are written into the log file. Now the output looks like this:

Subject Identifier	ITT Population Flag	Planned Treatment	Actual Treatment	Timepoint (N)
1	Y	1	1	201
				202
				203
101	Y	1	1	201
				202
				203

The variables were grouped as expected but the information for the subjects with the IDs 2 and 102 is missing completely. What happened? PROC REPORT doesn't include missing values into the grouping of a variable except if told so via the MISSING option. In our case we've noticed the missing observations immediately but in a data set with hundreds or thousands of observations it is very possible to overlook this error.
The corrected program code produces the desired output:

```
PROC REPORT DATA = treatment NOWD HEADLINE HEADSKIP MISSING;
   COLUMN subjid ittfl trtp trta tptn;

   DEFINE subjid / GROUP order WIDTH=15 FORMAT=3.;
   DEFINE ittfl  / GROUP order WIDTH=10 LEFT;
   DEFINE trtp   / GROUP order WIDTH=10 FORMAT=3.;
   DEFINE trta   / GROUP order WIDTH=10 FORMAT=3.;
   DEFINE tptn   /             WIDTH=13 FORMAT=5.;

   BREAK AFTER subjid / SKIP;
RUN;
```

Subject Identifier	ITT Population Flag	Planned Treatment	Actual Treatment	Timepoint (N)
1	Y	1	1	201 202 203
2	Y	2		201 202 203
101	Y	1	1	201 202 203
102	Y	2		201 202 203

HOW THE NODUPKEY OPTION CAN ELIMINATE DUPLICATES AND PRODUCE ERRORS AT THE SAME TIME

The NODUPKEY option in the PROC SORT procedure, together with the NODUP option, provides a great way to eliminate unwanted duplicate observations. While the NODUP option compares all variables of an observation to eliminate duplicates, NODUPKEY checks for observations with the same values in specified BY variables. This means that only the first observation that is unique within the values of the BY variables will be saved. Therefore, it is crucial that you ensure that all variables that require values for the further correct processing of the data are defined as BY variables. Furthermore, you need to make sure that you pick the right BY variables for your purpose.

The data set SASHELP.ZIPCODE will be used for the following examples.
Among others, the data set contains a variable with state names, one with county names, one with city names and a variable with ZIP codes.
We are not interested in individual ZIP code areas in one city, therefore we want to eliminate those duplicates and create a data set with one entry per city:

```
PROC SORT DATA = sashelp.zipcode OUT = cities NODUPKEY;
   BY city;
RUN;
```

The procedure runs without problems and we receive the following notes in the log file:

```
NOTE: There were 41267 observations read from the data set SASHELP.ZIPCODE.
NOTE: SAS sort was used.
NOTE: 22567 observations with duplicate key values were deleted.
NOTE: The data set WORK.CITIES has 18700 observations and 21 variables.
```

What we didn't consider is the fact that it is possible, that different cities may have the same name and if we want every individual city listed in our data set we need to consider possible superior variables that need to be included in our BY variables group. In this case, those variables are those listing the state names and the county names. It is essential to include both those variables if we really want individual variables on city level because we would get different results if we included only one of them:

```
PROC SORT DATA = sashelp.zipcode OUT = cities_per_state NODUPKEY;
   BY statename city;
RUN;
```

With these BY variables, we state that we want one city name per state. When this condition is met the first time, all other observations with this combination will be eliminated, ignoring the fact that one state can contain different counties with the same city name:

```
NOTE: There were 41267 observations read from the data set SASHELP.ZIPCODE.
NOTE: SAS sort was used.
NOTE: 11566 observations with duplicate key values were deleted.
NOTE: The data set WORK.CITIES_PER_STATE has 29701 observations and 21 variables.
```

```
PROC SORT DATA = sashelp.zipcode OUT = cities_per_county NODUPKEY;
   BY countynm city;
RUN;
```

With these BY variables we state that we want one city name per county, ignoring the fact that several states can contain counties with the same name:

```
NOTE: There were 41267 observations read from the data set SASHELP.ZIPCODE.
NOTE: SAS sort was used.
NOTE: 11588 observations with duplicate key values were deleted.
NOTE: The data set WORK.CITIES_PER_COUNTY has 29679 observations and 21 variables.
```

The correct way would be to include all three variables:

```
PROC SORT DATA = sashelp.zipcode OUT = cities_correct NODUPKEY;
   BY statename countynm city;
RUN;
```

```
NOTE: There were 41267 observations read from the data set SASHELP.ZIPCODE.
NOTE: SAS sort was used.
NOTE: 11440 observations with duplicate key values were deleted.
NOTE: The data set WORK.CITIES_CORRECT has 29827 observations and 21 variables.
```

Keep in mind that working with the NODUPKEY option requires you to align your data set so that no more information than permissible is eliminated.

If you want to see if your duplicate definition that you've set through the BY variables is correct, you can use the DUPOUT option, which writes the duplicates into a new data set:

```
PROC SORT DATA = sashelp.zipcode OUT = cities_correct DUPOUT = duplicates NODUPKEY;
   BY statename countynm city;
RUN;
```

```
NOTE: There were 41267 observations read from the data set SASHELP.ZIPCODE.
NOTE: SAS sort was used.
NOTE: 11440 observations with duplicate key values were deleted.
NOTE: The data set WORK.CITIES_CORRECT has 29827 observations and 21 variables.
NOTE: The data set WORK.DUPLICATES has 11440 observations and 21 variables.
```

The three above examples demonstrate the need to understand the modus operandi of SAS and to develop proper programming logic.
The last step of data checking is to ensure that the final output is correct. Because this task is highly dependent on the type of output and varies greatly depending on the type of output, it is beyond the scope of this paper.

CONCLUSION

Understanding your data and making sure that the written programming code leads to correct outcomes is not an easy task. This paper provides advice on how to achieve the correct mindset to tackle this problem with suggestions on how to get to better know your data as well as practical programming tips for producing high quality outputs.

REFERENCES

1 Delwiche, Lora D., and Slaughter, Susan J.: SAS® Macro Programming for Beginners, http://www2.sas.com/proceedings/sugi29/243-29.pdf

2 PhUSE Good Programming Practice Guidance, http://www.phusewiki.org/wiki/index.php?title=Good_Programming_Practice_Guidance

3 Delwiche, Lora D., and Slaughter, Susan J.: The little SAS® Book: A Primer, Fifth Edition, Cary, NC: SAS Institute Inc., page 298

4 Rosenbloom, Mary F. O. and Lafler, Kirk Paul: Best Practices: PUT More Errors and Warnings in My Log, Please!, http://support.sas.com/resources/papers/proceedings13/350-2013.pdf

5 Li, Arthur Xuejun: Essentials of the Program Data Vector (PDV): Directing the Aim to Understanding the DATA Step!, support.sas.com/resources/papers/proceedings13/125-2013.pdf

6 SAS® 9.4 Language Reference: Concepts, Sixth Edition: Overview of DATA Step Processing, http://support.sas.com/documentation/cdl/en/lrcon/69852/HTML/default/viewer.htm#p08a4x7h9mkwqvn16jg3xqwfxful.htm

ACKNOWLEDGMENTS

I would like to thank my colleagues Ellen von der Heyde, Christina Gelhorn, Gabi Lückel, Jana Bender, Sabine Erbslöh and Cenita Ettl for providing examples and program code for this paper as well as reviewing it.

CONTACT INFORMATION

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

Kristina Zweier
Clinipace Worldwide
Helfmann-Park 10
Eschborn / 65760
Work Phone: +49-6196-7709-320
Fax: +49-6196-7709-15320
Email: kzweier@clinipace.com
Web: https://www.clinipace.com/

APPENDIX 1

```
************************************************************************;
* Program name              : f_line_plot_check.sas                    *;
* Description               : Program to create Line Plot for different*;
                              parameters to find incorrect values      *;
* Output type               : Figure                                   *;
* Output files              : &outpath/f_lplot_xxx.png                 *;
* Developed Under           : LINUX SASV9.4                            *;
************************************************************************;



************************************************************************;
***                    Beginning of Code                             ***;
************************************************************************;

*) Macro to print the data for the selected parameter _TESTVAR;

%MACRO lplot      (  /* MP = Mandatory parameter                        */
                  in = /* MP Name of input data set                     */
          ,_testvar = /* MP Name of parameter name variable            */
              ,aval = /* MP Name of numeric parameter value variable   */
             ,visit = /* MP Name of chosen numeric date variable (e.g.
                         ADY, AVISITN)                                 */
          ,groupvar = /* MP Name of grouping variable (e.g. SUBJID,USUBJID)*/
           ,outpath = /* MP Name of output file pathname               */
                  );

   OPTIONS MPRINT; *displays the SAS statements that are generated by macro
                    execution;

   *) Select all parameter into a macro variable LISTCD and number of
      parameter into LISTMAX;
   PROC SQL NOPRINT;
      SELECT distinct &_testvar. into: listcd separated by ","
      FROM &in.;
      SELECT count(distinct &_testvar.) into: listmax
      FROM &in.;
   QUIT;

   %PUT INFO: listcd=&listcd., listmax=&listmax.;

   *) Create line plot for each parameter ;
   %DO i=1 %TO &listmax.;

     *) Select parameter of interest from the parameter list;
     %LET _testcd=%scan("&listcd",&i.,",");

     *) Create the dataset for parameter of interest only;
     DATA x_in;
       SET &in.;
       WHERE &_testvar. eq "&_testcd.";
     RUN;

     *) Calculate MIN and MAX value for the parameter - will be used for the
        Y-axis settings;
     PROC MEANS DATA=x_in min max;
        VAR &aval.;
        OUTPUT out=x_min_max min=min max=max;
     RUN;

     *) Define Y-axis;
     DATA _null_;
       SET x_min_max;
```

```
  diff= max-min;

  *) Calculate intercept Y-axis;
  IF      diff lt 0.01 THEN dec=0.001;
  ELSE IF diff lt  0.1 THEN dec=1.01;
  ELSE IF diff lt    1 THEN dec=0.1;
  ELSE IF diff lt   10 THEN dec=1;
  ELSE IF diff lt  100 THEN dec=10;
  ELSE IF diff lt 1000 THEN dec=100;
  ELSE dec=500;

  y_min=(int(min/dec) * dec); * minimum value of Y-axis;
  y_max=int((max+dec)/dec) * dec ; * maximum value of Y-axis;

  *) Write the values into macro variables - will be used later for the
     graph;
  call symput("y_max", strip(put(y_max, best.)));
  call symput("y_min", strip(put(y_min, best.)));
  call symput("y_dec", strip(put(dec, best.)));
RUN;

%PUT INFO: y_min=&y_min., y_max=&y_max., y_dec=&y_dec.;

************************************************************;
** PRODUCE  PLOT                                          **;
************************************************************;


*) Settings for the output;
****************************;

%LET figname = f_lplot_&in._&_testcd.;


title1 "Line plot for &_testcd. (ADS=&in.)";
footnote1 "";

*)  Set GOPTIONS;
****************;

 ODS LISTING gpath="&outpath." image_dpi=340;

 ODS GRAPHICS ON
        /reset /*you can set any options for customizations*/
         imagefmt=png
         imagename="&figname."  /*your figure name*/
         WIDTH=23.62cm
         height=15cm
         border=on
         ;

PROC SGPLOT DATA=x_in noautolegend; /*legend on X-axis not needed*/

   SERIES x=&visit. y=&aval. / lineattrs=(thickness=2)
   GROUP=&groupvar. markers markerattrs=(symbol=circlefilled size=7)
   ;

   XAXIS FITPOLICY=ROTATE
   LABEL="Day" OFFSETMIN=0.02 OFFSETMAX=0.03;

   YAXIS  VALUES=(&y_min. to &y_max. by &y_dec.)
   LABEL="Values of &_testcd." OFFSETMIN=0.07 OFFSETMAX=0.02;

RUN;

ODS GRAPHICS OFF;
```

```
      *) Delete temporary datasets created in the macro loop;
      PROC DATASETS lib = work;
        DELETE x_in x_min_max;
      QUIT;

   %END;

*) Macro end;
%MEND lplot;

*******************************************************************************;
***                                END OF CODE                                  ***;
*******************************************************************************;
```