Paper 3181-2015

How Best to Effectively Teach the SAS® Language

Arthur X. Li, City of Hope National Medical Center, Duarte, CA

ABSTRACT

Learning a new programming language is not an easy task, especially for someone who does not have any programming experience. Learning the SAS® programming language can be even more challenging. One of the reasons is that the SAS system is comprised of a variety of languages, such as DATA step language, SAS macro language, Structured Query Language for the SQL procedure, etc. Furthermore, each of these languages has its own unique characteristics and simply learning the syntax is not sufficient to grasp the language essence. Thus, it is not unusual to hear about someone who has learned SAS for several years and has never become a SAS programming expert. By using the DATA step language as an example, I would like to share some of my experiences on effectively teaching the SAS language.

INTRODUCTION

A typical introductory SAS course often focuses on accessing different types of data in the DATA step, combining data, conditional processing, and creating basic report writing. An intermediate course often covers more advanced techniques in the DATA step, such as iterative processing, array processing, and restructuring data. These two courses combined generally run about five to six full days.

The teaching philosophy of these types of courses often focus on showing students how to program in various scenarios and teaching students the syntax and the program to achieve the task. Thus, an introductory programming course can often run several days. Since it is impossible to cover all the programming scenarios, it is very common to see some students still unable to solve many programming problems after taking the course.

I believe that an effective teaching method should concentrate on the following:

1. Knowing how to obtain solutions from *SAS Help and Documentation*
2. Learning programming logic and problem solving strategies
3. Letting the students get into the habit of examining data
4. Understanding the DATA step process during the DATA step execution.

Focusing on these key concepts not only will reduce the teaching time, but also enable students to solve most programming problems on their own.

KNOW YOUR AUDIENCES

One of the challenges in teaching the SAS language is that the attending audiences have different programming backgrounds. Before taking the programming course, we often specify the requirements for taking the course. For example, the prerequisite for taking an introductory course often doesn't require the students having any knowledge of the SAS programming language.

When the course starts, we often encounter three types of students based on their programming background: students who didn't have any programming experiences, students who have programmed SAS for a few months, and students who did not know SAS but were proficient in other programming languages, such as C++, R, Fortran, etc. Among these students, students who have programming experiences in other languages will be able learn a new programming language much faster compared to the ones who have no prior programming experience.

Even for the majority of the students who understood the introductory materials, we still have to assume that all the students have no prior knowledge of the SAS language and we should keep teaching at a slow pace to avoid letting students who have no prior programming experience fall behind. Ideally, the first session of the course should be taught in a hands-on format, which shows students how to read simple

data, create variables by using the assignment statement, illustrate how to submit the program, and examine the SAS log and output.

GETTING HELP BY READING SAS HELP AND DOCUMENTATIONS

One of the best resources for learning SAS and searching for SAS programming help is the *SAS Help and Documentations*, which is part of our SAS software installation. However, it is very common to see novice SAS users ask us how to use a certain SAS statement, a procedure, a function, an expression, or a DATA step option. The answers to most of their questions can be easily found in the *SAS Help and Documentation*; however, beginning programmers often do not know where to search for them. Thus, as a SAS instructor, it is important to show students how and where to search answers from the manual.

One of the reasons that a beginning programmer did not know where to obtain answers from *SAS Help and Documentations* is that they did not clearly distinguish different types of language terminologies. For example, they are often confused about the difference between the SUM function and the SUM statement or the difference between the KEEP= DATA step option and the KEEP statement. Thus, the instructor should introduce these terminologies to the students throughout the entire course properly. Let's use program 1 as an example:

Example1.txt:

```
Barbara 61 12D
John    62 175
```

Program 1:

```
data ex1;
    infile 'W:\SAS Book\dat\example_1.txt';
    input name $ 1-7 height 9-10 weight 12-14;
    BMI = 700*weight/(height*height);
    output;
run;
```

Program 1 consists of six statements, including DATA, INFILE, INPUT, assignment, OUTPUT, and RUN statements. After showing the semantics of each statement, the instructors should also illustrate how to locate these statements from SAS Help and Documentation. For example, in SAS 9.4, the INFILE statement is located under SAS Product →Base SAS →SAS 9.4 Statements Reference → Dictionary of SAS Statements. Once the entry for the INFILE statement is located, we should show the students how to read the documentations. For example, the INFILE statement has the following form:

INFILE *file-specification* <*device-type*> <*options* > <*operating-environment-options*>;

The words in bold, such as INFILE, in the syntax is the keyword, which cannot be modified. Words that are italicized need to be modified by the user, such as file-specification. In this example, file-specification is used to specify the location of the input data; the common form of file-specification is the physical file name along with its full path. The language elements that are enclosed between a less than sign (<) and a greater than sign (>) are options for the SAS statement. At the end of the statement, there is a semicolon, which ends the INFILE statement.

Most of the SAS statements have a large selection of options. In the introductory SAS course, it is not necessary to go through all the options in great detail. The instructors should just show one or two options to the students. For example,

OBS=*record-number*

This optional OBS option is used to specify the first number of records to be read. Notice that there is no semicolon at the end of the OBS= option because this is not a statement.

The assignment statement is the most common method for creating a variable, but it is often ignored in lectures. The assignment statement in Program 1 is used to create the BMI variable. The instructor should introduce the SAS expression when teaching the assignment statement. The assignment statement has the following form:

variable=expression;

The purpose of using an expression in a SAS statement is to create variables, assign values, perform calculations, transform variables, and/or perform conditional processing. All expressions will return a result with a character, numeric, or Boolean value. An expression is formed by a sequence of operands and operators. Operands are either constants or variables. In Program 1, 700, WEIGHT, and HEIGHT are the operands and multiplication and division symbols are operators. Operators are not limited to the arithmetic operators. In SAS, it also includes operators for comparisons and logical operations. SAS functions and grouping parentheses are also part of operators. The instructors should emphasize these terminologies to the students so that the students would be able to locate these documentations in the *SAS Help and Documentation*.

PROGRAMMING LOGIC AND ALGORITHM

For students who have never programmed before, it is difficult for them to start writing a program, even for a simple problem. Understanding the programming logic is essential for writing programs. Thus, when we teach a programming course, we should focus on teaching programming algorithms.

For example, the goal of Program 1 is to read an external file, which contains NAME, HEIGHT, and WEIGHT variables. Then Program 1 creates the BMI variable based on HEIGHT and WEIGHT. In order to write this program, we have to start with the INFILE statement, followed by the INPUT statement. The INFILE statement is used to identify the location of the external file and the INPUT statement instructs SAS how to read each observation. Thus, you must place the INFILE statement before the INPUT statement because SAS needs to know where to find the external file before it can read it. After reading the HEIGHT and WEIGHT variables, you can then create the BMI variable.

Once the students understand the programming logic, they would then be able to write some simple programs.

DATA CHECKING

When creating a new variable, novice programmers tend to simply create the variables without checking the accuracy of the variables. Thus, data checking should be introduced at the beginning of the course. A proper way to create a variable based on an existing variable should consist of three important steps:

1. Evaluating the existing variable

2. Creating the new variable

3. Checking the accuracy of the newly-created variable

Using these three steps is especially important when creating variables. Numerous SAS procedures can be used to evaluate either the existing or the newly-created variables. For example, PROC PRINT will give you a general idea what the data looks like. Instead of printing the entire data set, one can use the OBS= data set option to print only the first few observations of the data set. To examine the variable attributes and the total number of observations, PROC CONTENTS can be used. The two most useful procedures to examine data are PROC MEANS and PROC FREQ. These two procedures are easy to learn for beginning programmers. PROC MEANS is used to evaluate the distribution of numeric variables; using the NMISS option in PROC MEANS is especially useful for examining missing numerical values. To examine the frequency of a categorical variable, one can use PROC FREQ. Using the MISSING or MISSPRINT options in PROC FREQ to checks whether a categorical variable contains missing values.

Suppose that we would like to create an indicator variable (AGE_HI) based on the numeric AGE variable. We would like to assign a value of 1 to AGE_HI when AGE is greater than its median value; otherwise AGE_HI is set to zero (0).

To follow the three-step procedure above, we need to examine the AGE variable first. In addition to knowing the median value of the AGE variable, it is important to know the number of missing and non-missing values for the AGE variable because if the AGE variable is missing for an observation, AGE_HI should be assigned with a missing value as well. All this information can be found from PROC MEANS by using the MEDIAN (median value), N (number of non-missing value), and NMISS (number of missing values) options. See Program 2.

Program 2:

```
title 'Evaluate the AGE variable';
proc means data=hearing n nmiss median maxdec=2;
    var age;
run;
```

```
                    Evaluate the AGE variable

                       The MEANS Procedure
                    Analysis Variable : Age
                            N
                    N    Miss          Median
                    --------------------------
                    33       1           26.00
                    --------------------------
```

Output 1. Output from Program 2.

The output from PROC MEANS from Program 2 shows that the median age is 26. There are 33 non-missing values and one missing value for AGE. In the second step, Program 3 creates the variable AGE_HI by using the IF-THEN/ELSE statement.

Program 3:

```
data hearing2_1;
    set hearing;
    if age > 26 then age_hi = 1;
    else age_hi = 0;
run;
```

Once the AGE_HI variable is created, the final (and most important) step is to validate that the newly-created variable (AGE_HI) is indeed correctly created. To check the accuracy of AGE_HI, we need to make sure that the maximum value for AGE within the low AGE_HI group (AGE_HI = 0) is 26 and that the minimum value of AGE within the high AGE_HI group (AGE_HI = 1) is greater than 26. Program 4 uses PROC MEANS to exam the minimum and the maximum values of AGE by each category of AGE_HI. Furthermore, Program 4 also examines the number of missing and non-missing values within each level of AGE_HI by using the N and NMISS options.

Program 4:

```
title 'Checking AGE_HI is created correctly';
proc means data=hearing2_1 n nmiss min max maxdec=2;
    class age_hi;
    var age;
run;
```

```
              Checking AGE_HI is created correctly
                        The MEANS Procedure
                    Analysis Variable : Age
                  N             N
      age_hi    Obs     N    Miss         Minimum         Maximum
-------------------------------------------------------------------
           0     18    17       1           15.00           26.00
           1     16    16       0           28.00           36.00
-------------------------------------------------------------------
```

Output 2. Output from Program 4.

Based on the output from Program 4, the AGE range is correct when AGE_HI equals 1. There is a problem, however, when AGE_HI is set to zero in the IF-THEN/ELSE statement. Since missing (.) is a number, in fact the smallest numeric value in SAS, the missing value for AGE incorrectly translates to a non-missing zero in AGE_HI. In this situation, we can use the IF-THEN/ELSE statement to exclude the missing value when creating the AGE_HI variable (Program is not shown).

DATA STEP EXECUTION

Knowing how to follow the syntax and understanding simple programming logic will not assure how to solve all the programming problems. Furthermore, some of the SAS statements not not easy to comprehend because they are closely related to understanding the DATA step execution, such as OUTPUT, subsetting IF, RETAIN, and SUM statements. These statements are closely related to knowing how the DATA steps execute and understanding what happens in the program data vector (PDV) during the execution phase.

We were often asked by students to debug their programs. If the errors are syntax related, they can be easily identified from the SAS log. One the other hand, it is difficult to diagnose a logic error. Their programs run without showing any syntax error, however the data set that is generated is not what was intended, such as there are more or less observations . In order to debug a logic error, programmers have to understand the concept of DATA step execution. The DATA step execution concept is often taught in an intermediate SAS course. I believe that students need to be exposed to this concept in the early stage of learning.

Teaching DATA execution consists of four main components; including understanding how the PDV works, by-group processing, loop and array.

UNDERSTANDING HOW THE PDV WORKS

To illustrate how the DATA step executes, we can use a simple example, like the example in Program 1. During the demonstration, it is better to show the contents in the input buffer and PDV at each step during the DATA step execution. The best way to show the changes in the PDV is to use the animation features in the PowerPoint presentation. Figure 1 demonstrates the conceptual figures of the input buffer and the PDV.

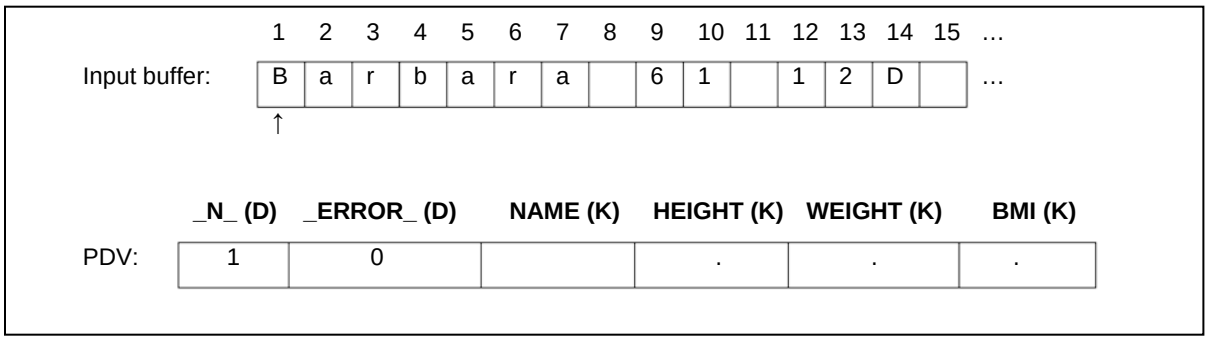

Figure 1. Input buffer and PDV.

There are several important concepts that instructors need to emphasize when teaching the DATA step execution:

- The input buffer is used to hold raw data and it is used for reading raw data, and it is not used when reading a SAS data set.
- The DATA step execution works like a loop; it repetitively reads data and creates observations one at a time. This implicit loop is different from the explicit iterative DO loop.
- The automatic variable, _N_, is used to indicate the currently-processed observation. The automatic variable _ERROR_ is used to signal the data error of the currently-processed observation.
- When creating a data set that is based on reading the raw data, SAS initializes variable values in the PDV to missing at the beginning of each iteration of execution, except for the automatic variables, variables that are named in the RETAIN statement, variables that are created by the SUM statement, data elements in a _TEMPORARY_ array, and variables created in the options of the FILE/INFILE statement.
- When creating a data set by reading an existing SAS data set (via the SET statement), SAS sets each variable to missing in the PDV only before the first iteration of the execution. Variables will keep their values in the PDV until they are replaced by the new values from the input data set. These variables exist in both the input and output data sets. However, the newly-created variables will be set to missing in the PDV at the beginning of every iteration of the execution. This concept is especially important because in some applications, one needs to use the RETAIN statement to retain the values of newly-created variables from previous iterations.
- Only the variables marked with (K) will be written to the output data set. Automatic variables, on the other hand, are always marked with a (D) so they are never written out.
- Not all SAS statements in the DATA step are executed during the execution phase. Instead, statements in the DATA step can be categorized as *executable* or *declarative*. The declarative statements, such as LENGTH, FORMAT, LABEL, DROP, and KEEP, are used to provide information to SAS and only take effect during the compilation phase. The declarative statements can be placed in any order within the DATA step. In contrast to declarative statements, the order in which executable statements appear in the DATA step matters greatly.

A few SAS statements should be emphasized because these statements control how observations are generated in the PDV. For example, the RETAIN statement is used to retain the numeric variables in the PDV. The variables that are specified in the RETAIN statement are initialized with a value at the first iteration of the DATA step execution. If you do not specify an initial value, the retained variables are initialized as missing before the first execution of the DATA step. The RETAIN statement prevents the variables from being initialized each time the DATA step executes. The RETAIN statement is a declarative statement and does not execute during the DATA step execution phase. An example of using the RETAIN statement is illustrated in Program 5, which creates the variable TOTAL by accumulating the SCORE variable.

Ex1.sas7bdat:

	ID	SCORE
1	A01	3
2	A02	.
3	A03	4

Program 5:

```
data prog5;
    set ex1;
    retain total 0;
    total = sum(total, score);
run;
```

The SUM statement is used to create an accumulating numeric variable. The variable that is created from the SUM statement is automatically set to 0 at the beginning of the first iteration of the DATA step execution and is retained in following iterations.

Program 6:

```
data prog6;
    set ex1;
    total+score;
run;
```

The subsetting IF statement controls whether the observations in the PDV are outputted to the output data set. If the *expression* is true for the current observation, SAS continues to execute statements in the DATA step and includes the current observation in the data set. The resulting SAS data set contains a subset of the external file or SAS data set. On the other hand, if the *expression* is false, then no further statements are processed for that observation and SAS immediately returns to the beginning of the DATA step. That is to say, the remaining program statements in the DATA step are not executed and the current observation is not written to the output data set. For example, Program 7 creates a data set that only contains the observations in which the SCORE variable is not missing.

Program 7:

```
data prog7;
    set ex1;
    total+score;
    if not missing(score);
run;
```

The OUTPUT statement is used to tell SAS to write the current observation from the PDV to a SAS data set immediately. An OUTPUT statement is not actually needed in Program 1 because the contents of the PDV are written out by default with an implicit OUTPUT statement at the end of the DATA step. Placing an explicit OUTPUT statement will override the implicit OUTPUT statement. In some applications, more than one OUTPUT statement in the DATA step can be used. The OUTPUT statement is especially useful when utilizing a loop structure in the program. Program 8 illustrates the data transformation from the wide format to the long format by using multiple OUTPUT statements in one DATA step. Since only two observations need to be read from the WIDE data set, there will be only two iterations for the DATA step processing. That means we need to generate the output up to three times for each iteration. In some iterations, the output might not be generated three times because missing values in variables S1 – S3 will not be outputted in the LONG data set.

Wide.sas7bdat:

	ID	S1	S2	S3
1	A01	3	4	5
2	A02	4	.	2

Long.sas7bdat:

	ID	TIME	SCORE
1	A01	1	3
2	A01	2	4
3	A01	3	5
4	A02	1	4
5	A02	3	2

Program 8:

```
data long(drop=s1-s3);
    set wide;
    time = 1;
    score = s1;
    if not missing(score) then output;
    time = 2;
    score = s2;
    if not missing(score) then output;
    time = 3;
    score = s3;
    if not missing(score) then output;
run;
```

BY-GROUP PROCESSING

BY-group processing is a method of processing records from data sets that can be grouped by the values of one or more common variables. It is one of the most important sections because many applications that involve longitudinal data require utilizing BY-group processing. Understanding BY-group processing is closely related to understanding how the DATA step executes. Here are a few important concepts that need to be emphasized:

- During BY-group processing, SAS creates two automatic variables, FIRST.VARIABLE and LAST.VARIABLE, which are used to indicate the beginning or the end of the measurement within each BY group.
- Both FIRST.VARIABLE and LAST.VARIABLE are temporary variables, so they are not sent to the output data set.
- Both FIRST.VARIABLE and LAST.VARIABLE are initialized to 1 at the beginning of the DATA step execution.
- Then FIRST.VARIABLE is set to 1 in the PDV when SAS reads the first observation in each BY group and is set to 0 when reading the second to the last observation in each BY group. Similarly, LAST.VARIABLE is set to 1 when reading the last observation in each BY group and set to 0 when reading those observations that are not last.

For example, Program 9 illustrates an application for utilizing the BY-group processing method. In this example, we need to calculate the total scores for each subject in the EX2 data set. To create a variable TOTAL that is the total score for each subject, we need to initialize TOTAL to 0 when starting to read the first observation of each subject. Then TOTAL can be accumulated by adding the value from the SCORE variable to TOTAL for each observation. In the end, we can output the TOTAL score when reading the last observation of each subject.

Ex2.sas7bdat:

	ID	SCORE
1	A01	3
2	A01	3
3	A01	2
4	A02	4
5	A02	2

Program 9:

```
proc sort data=ex2;
    by id;
run;

data prog9 (drop=score);
    set ex2;
    by id;
    if first.id then total = 0;
    total + score;
    if last.id;
run;
```

Restructuring data sets from the long format to the wide format is another good example to illustrate BY-group processing. Program 10 begins by sorting the LONG data set by ID and TIME. Sorting the variable TIME within each ID is important because it ensures that the horizontal order of S1 – S3 in the WIDE data set for each subject can be matched correctly with the vertical order of SCORE in the LONG data set. Since we are reading five observations from the LONG data set but only creating two observations, we only need to generate one observation once all the observations for each subject have been processed. The newly-created variables (S1 – S3) in the final data set need to retain their values; otherwise S1 – S3 will be initialized to missing at the beginning of each iteration of the DATA step processing. Notice that subject A02 is missing one observation for TIME equaling 2. The value of S2 from the previous subject (A01) will be copied to the data set WIDE for the subject A02 instead of a missing value because S2 is being retained. To avoid this problem, initialize S1 – S3 to missing when processing the first observation for each subject.

Program 10:

```
proc sort data=long;
    by id time;
run;

data wide (drop=time score);
    set long;
    by id;
    retain s1-s3;
    if first.id then do;
        s1 = .;  s2 = .; s3 = .;
    end;
    if time = 1 then s1 = score;
    else if time = 2 then s2 = score;
    else s3 = score;
    if last.id;
run;
```

EXPLICIT LOOPS

A loop is a basic logical programming concept where one or more statements are executed repetitively until a predefined condition is satisfied. Loop processing in SAS is more complex than other programming languages because SAS has implicit and explicit loops which can cause confusion for novice programmers.

An implicit loop, which was introduced in Understanding HOW the PDV Works section, results when the DATA step repetitively reads data values from the input data set, executes statements, and creates observations for the output data set (one at a time) during the execution phase. On the other hand, an explicit loop utilizes DO, DO WHILE, or DO UNTIL statements to repetitively execute statements within the iteration of DATA step execution.

For example, suppose the names of hospitals are stored in the SAS data set CANCER_CENTER. For each hospital, we need to assign 4 patients where each patient has a 50% chance of receiving either the drug or a placebo. In this situation, we need to read in the value for the CENTER variable via the implicit loop. Then for each CENTER that is being read into the PDV, we need to utilize an explicit loop to create the ID and GROUP variables. Program 11 shows how an implicit and an explicit loop used together deliver what's needed.

Cancer_center.sas7bdat:

	CENTER
1	COH
2	UCLA
3	USC

Program 11:

```
data prog11;
    set cancer_center;
    do id = 1 to 4;
        if  ranuni(2) > 0.5 then group = 'D';
        else group = 'P';
        output;
    end;
run;
```

ARRAY PROCESSING

Array processing is the most advanced topic in an introductory SAS course. One of the reasons for its complication is because of the syntax which has been enriched to allow for nuanced data processing. Even when the syntax is mastered, one still needs to understand how the array interacts with what is going on in the PDV. Furthermore, knowing when it is best to replace existing code with an array presents an additional challenge.

The main purpose of using array processing is to improving the efficiency of the code-writing. And many applications that use array processing need to use the loop structure. The following example illustrates an example of the necessity for utilizing array processing to improve efficiency. In this example, the data set SBP contains six measurements of systolic blood pressure (SBP) measurements for four patients. The missing values are coded as 999. To recode 999 to a standard numeric missing value (.), one needs to use six IF/THEN statements like in Program 12.

Sbp.sas7bdat:

	Sbp1	Sbp2	Sbp3	Sbp4	Sbp5	Sbp6
1	141	142	137	117	116	124
2	999	141	138	119	119	122
3	142	999	139	119	120	999
4	136	140	142	118	121	123

Program 12:

```
data prog12;
    set sbp;
    if sbp1 = 999 then sbp1 = .;
    if sbp2 = 999 then sbp2 = .;
    if sbp3 = 999 then sbp3 = .;
    if sbp4 = 999 then sbp4 = .;
    if sbp5 = 999 then sbp5 = .;
    if sbp6 = 999 then sbp6 = .;
run;
```

Each of the IF statements in Program 12 converts the number 999 to a missing SAS value. These IF statements are almost identical; only the name of the variables are different. If these six variables can be grouped into a one-dimensional array, then we can recode the variables in a DO loop. In this situation, grouping variables into an array will make code writing more efficient. Program 13 is a modified version of Program 12 by using array processing.

Program 13:

```
data prog13(drop=i);
    set sbp;
    array sbpary[6] sbp1-sbp6;
    do i = 1 to 6;
        if sbpary[i] = 999 then sbpary[i] = .;
    end;
run;
```

Students may find it difficult to start writing a program by using the the array processing. In this situation, it is better to simplify the data set, such as keeping small numbers of variables, and then writing a simple program without using arrays. Then we can guide the students to modify the program by utilizing array processing. For example, transforming a data set from the long format to the wide format, or vice versa, was introduced in previous sections. The solutions without using array processing are not efficient if you have a large number of variables that need to be transposed. However, if the students can understand the program that did not use array processing, and then improving that program by using the array processing is not nearly as difficult as starting by writing a program with array processing.

For example, Program 14 is created by modifying Program 8. The variables S1, S2, and S3 are grouped into the array S and the program utilizes the iterative DO loop with the TIME variable as the index variable. The output is generated within the DO loop when the SCORE variable is not missing

Program 15 is an improved version of Program 10. Variables S1, S2, and S3 are created by using the ARRAY statement. When reading the first observation of each subject (FIRST.ID = 1), an iterative DO loop is used to initialize each element in the array (S[I]) to missing. Utilizing array processing simplifies processing shown in Program 10 since the cumbersome IF-THEN/ELSE statement is replaced with an assignment statement that uses TIME as the subscript of the S array.

Program 14:

```
data long(drop=s1-s3);
    set wide;
    array s[3];
    do time = 1 to 3;
        score = s[time];
        if not missing(score) then output;
    end;
run;
```

Program 15:

```
data wide (drop=time score i);
    set long;
    by id;
    array s[3];
    retain s;
    if first.id then do;
        do i = 1 to 3;
            s[i] = .;
        end;
    end;
    s[time] = score;
    if last.id;
run;
```

CREATING AN EFFECTIVE COURSE CURRICULUM

In this section, I will show sample course contents for teaching an introductory SAS course, which contains the following sections:

1. Introduction to SAS
2. Conditionally create variables
3. DATA step processing
4. By-group processing
5. Loops
6. Array processing
7. Combining data
8. SAS functions
9. Useful SAS procedures

In a typical introductory SAS course, reading text files, such as using column, formatted, list, and named input methods spans a large chunk of teaching time. I personally don't think we should focus on this topic. The reason is that reading text files from different formats is syntax-driven. The students can easily learn themselves or follow the syntax from the manual. Secondly, reading the data into SAS only accounts for a very small fraction of our daily programming time. Thus, it is unnecessary to spend lots of teaching time on reading the text files into SAS.

Each section in the list above can span one or two hours depending upon the number of examples that the instructors would like to show during the lecture.

The first two sections serve as prerequisites for the rest of the course. Section 1 covers an introduction and overview of the SAS language, which includes reading data into the SAS system, some Base SAS procedures, creating variables, and subsetting data sets. Some of the topics in these chapters will also be

expanded in detail in the later section. Section 2 covers how to conditionally create variables based on existing variables, such as how best to use the IF-THEN/ELSE statement.

The most important component of the course is from section 3 through section 6. These four sections describe how essential it is to understand the PDV in order to write an accurate program in the DATA step. The topic covered in each section is built-upon the concepts covered in a previous section. Section 3 provides an overview about DATA step processing, how to retain variables by using the RETAIN and SUM statements, and conditional processing in the DATA step. Section 4 covers by-group processing within the DATA step and some applications relating to longitudinal data sets. Section 5 introduces topics on explicit loops in the DATA step, and compares the differences between implicit and explicit loops. Section 6 covers array processing, which includes one- to multi-dimensional arrays.

Chapter 7 covers multiple methods for combining data sets vertically and horizontally. Some examples in this section demonstrate an understanding of the PDV. Section 8 covers DATA step functions and call routines, which represents one of the many strengths of SAS software. Section 9 covers some useful SAS procedures that relate to data management, which includes the SORT, COMPARE, TRANSPOSE, FORMAT, and OPTIONS procedures.

CONCLUSION

To become a successful SAS programmer, one must be able to thoroughly comprehend how DATA steps are processed and know how to solve problems independently. Thus, as an instructor, we should concentrate on teaching students learning programming logic, guiding them to search solutions, and focusing on showing how DATA steps are processed.

REFERENCES

Li, Arthur. 2013. Handbook of SAS® DATA Step Programming. Chapman and Hall/CRC.

ACKNOWLEDGMENTS

I would like to thank Rebecca Ottesen for inviting me to present this paper at SGF 2015.

CONTACT INFORMATION

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

Arthur Li
City of Hope National Medical Center
Division of Information Science
1500 East Duarte Road
Duarte, CA 91010 - 3000
Work Phone: (626) 256-4673 ext. 65121
Fax: (626) 471-7106
E-mail: arthurli@coh.org