1 The RETAIN Statement

Introduction

Suppose you have a SAS data set of clinical data. Each observation corresponds to a visit to the clinic and contains such information as a patient number, the date of the visit, and some information about the visit. It would be useful to compare information from one visit to another, for a given patient. How many days were there from the previous visit? Did this patient's blood pressure go up or down? Questions such as these are traditionally more difficult to answer using SAS software than comparisons within an observation. This first chapter introduces one of the most useful tools for "remembering" information from a previous observation—the RETAIN statement. You will see, in detail, exactly how this versatile statement works and how to avoid getting in trouble when using it.

Demonstrating a DATA Step with and without a RETAIN Statement

The RETAIN statement is often a mystery to beginning SAS programmers. To understand how the RETAIN statement works, you must first understand the basic operation of the SAS DATA step.

Program 1-1 demonstrates a SAS DATA step where a RETAIN statement is not used.

Program 1-1: Demonstrating a DATA Step without a RETAIN Statement

```
DATA WITHOUT_1;
   PUT "Before the INPUT statement:  " _ALL_;  ❶
   INPUT X @@;
   PUT "After the INPUT statement:   " _ALL_ /;
DATALINES;
1 2 . 3
;
```

By placing the PUT statements at strategic places in the DATA step, you can see what is going on "behind the scenes." A PUT statement writes out text or the value of variables to the location specified by a FILE statement. Since there is no FILE statement in this DATA step, the results of the PUT statements are written to the SAS Log (the default location). The keyword _ALL_ ❶ causes the PUT statement to output the value of all the variables, including some SAS internal variables, such as _N_.

Before we examine the log produced by this program, let's be sure that you understand the meaning of the double @ sign in the INPUT statement. A trailing double @ says to the DATA step, "Hold the line." That is, do not move the pointer to a new line each time the DATA step iterates. Instead, just keep on reading data values until there are no more on the line. Many of the examples in this book will use a trailing double @ to save some space and make the programs more compact (and perhaps easier to read).

Let's look at the SAS log that results from running Program 1-1:

```
78   DATA WITHOUT_1;
79      PUT "Before the INPUT statement:  " _ALL_;
80      INPUT X @@;
81      PUT "After the INPUT statement:   " _ALL_ /;
82   DATALINES;

Before the INPUT statement:  X=. _ERROR_=0 _N_=1
After the INPUT statement:   X=1 _ERROR_=0 _N_=1

Before the INPUT statement:  X=. _ERROR_=0 _N_=2
After the INPUT statement:   X=2 _ERROR_=0 _N_=2

Before the INPUT statement:  X=. _ERROR_=0 _N_=3
After the INPUT statement:   X=. _ERROR_=0 _N_=3

Before the INPUT statement:  X=. _ERROR_=0 _N_=4
After the INPUT statement:   X=3 _ERROR_=0 _N_=4

Before the INPUT statement:  X=. _ERROR_=0 _N_=5
```

Notice that the value of X is a missing value at the top of the DATA step (before the INPUT statement). This is the usual way that a SAS DATA step operates. Notice also that the DATA step does not stop until it tries to read a fifth data value and realizes that there are no more data values to read.

Let's modify Program 1-1 by adding a RETAIN statement.

Program 1-2: Demonstrating a DATA Step with a RETAIN Statement

```
DATA WITH_1;
   RETAIN X;
   PUT "Before the INPUT statement:  " _ALL_;
   INPUT X @@;
   PUT "After the INPUT statement:   " _ALL_ /;
DATALINES;
1 2 . 3
;
```

The resulting log is shown next:

```
86   DATA WITH_1;
87      RETAIN X;
88      PUT "Before the INPUT statement:  " _ALL_;
89      INPUT X @@;
90      PUT "After the INPUT statement:   " _ALL_ /;
91   DATALINES;

Before the INPUT statement:  X=. _ERROR_=0 _N_=1
After the INPUT statement:   X=1 _ERROR_=0 _N_=1

Before the INPUT statement:  X=1 _ERROR_=0 _N_=2
After the INPUT statement:   X=2 _ERROR_=0 _N_=2

Before the INPUT statement:  X=2 _ERROR_=0 _N_=3
After the INPUT statement:   X=. _ERROR_=0 _N_=3

Before the INPUT statement:  X=. _ERROR_=0 _N_=4
After the INPUT statement:   X=3 _ERROR_=0 _N_=4

Before the INPUT statement:  X=3 _ERROR_=0 _N_=5
```

Notice that the value of X is missing the first time, but for each additional iteration of the DATA step, it retains the value it had in the previous iteration. Notice that the value of X is missing after the third value of X is read. This missing value is retained just as all the other nonmissing values in this example.

In its simplest form, a RETAIN statement takes the form RETAIN *list_of_variables*; by the way, it doesn't matter where we place the RETAIN statement in the DATA step—the effect is the same.

There was no reason to use a RETAIN statement in Program 1-2 other than to demonstrate how a DATA step runs with and without such a statement. The next example, Program 1-3, is an attempt (that does not work) to use a value from a previous observation whenever a missing value is read from the input data.

Program 1-3: **Demonstrating a DATA Step That Does Not Work without a RETAIN Statement**

```
***If there is a missing value for X, use the value
   from the previous observation ;
***Note: This program does NOT work as planned;
DATA WITHOUT_2;
   PUT "Before INPUT:      " _ALL_ ;
   INPUT X @@;
   IF X NE . THEN OLD_X = X;
   ELSE X = OLD_X;
   PUT "After assignment:  " _ALL_ /;
DATALINES;
1 2 . 3
;
```

The goal of this program is to substitute a value of X for a previous nonmissing value of X whenever the current value of X (read from the input data) is a missing value. However, as we can see in the log below, this program does not work because OLD_X is set to a missing value at the top of the DATA step for every iteration.

```
94   ***If there is a missing value for X, use the value
95      from the previous observation ;
96   ***Note: This program does NOT work as planned;
97   DATA WITHOUT_2;
98      PUT "Before INPUT:      " _ALL_ ;
99      INPUT X @@;
100     IF X NE . THEN OLD_X = X;
101     ELSE X = OLD_X;
102     PUT "After assignment:  " _ALL_ /;
103  DATALINES;

Before INPUT:      X=. OLD_X=. _ERROR_=0 _N_=1
After assignment:  X=1 OLD_X=1 _ERROR_=0 _N_=1
Before INPUT:      X=. OLD_X=. _ERROR_=0 _N_=2
After assignment:  X=2 OLD_X=2 _ERROR_=0 _N_=2

Before INPUT:      X=. OLD_X=. _ERROR_=0 _N_=3
After assignment:  X=. OLD_X=. _ERROR_=0 _N_=3

Before INPUT:      X=. OLD_X=. _ERROR_=0 _N_=4
After assignment:  X=3 OLD_X=3 _ERROR_=0 _N_=4

Before INPUT:      X=. OLD_X=. _ERROR_=0 _N_=5
```

You can see that OLD_X is assigned the nonmissing values as planned, but the PUT statement right after the DATA statement shows that it is set to missing each time the DATA step iterates and the program fails to work as desired.

Look at Program 1-4 to see how a RETAIN statement changes things.

Program 1-4: Adding a RETAIN Statement to Program 1-3

```
***If there is a missing value for X, use the value
   from the previous observation;
***Note: With the added RETAIN statement, the program now works;
DATA WITH_2;
   RETAIN OLD_X;
   PUT "Before INPUT:      " _ALL_ ;
   INPUT X @@;
   IF X NE . THEN OLD_X = X;
   ELSE X = OLD_X;
   PUT "After assignment:  " _ALL_ /;
DATALINES;
1 2 . 3
;
```

This is the resulting output:

```
106  ***If there is a missing value for X, use the value
107     from the previous observation;
108  ***Note: With the added RETAIN statement, the program now works;
109  DATA WITH_2;
110     RETAIN OLD_X;
111     PUT "Before INPUT:      " _ALL_ ;
112     INPUT X @@;
113     IF X NE . THEN OLD_X = X;
114     ELSE X = OLD_X;
115     PUT "After assignment:  " _ALL_ /;
116  DATALINES;

Before INPUT:      OLD_X=. X=. _ERROR_=0 _N_=1
After assignment:  OLD_X=1 X=1 _ERROR_=0 _N_=1

Before INPUT:      OLD_X=1 X=. _ERROR_=0 _N_=2
After assignment:  OLD_X=2 X=2 _ERROR_=0 _N_=2

Before INPUT:      OLD_X=2 X=. _ERROR_=0 _N_=3
After assignment:  OLD_X=2 X=2 _ERROR_=0 _N_=3

Before INPUT:      OLD_X=2 X=. _ERROR_=0 _N_=4
After assignment:  OLD_X=3 X=3 _ERROR_=0 _N_=4

Before INPUT:      OLD_X=3 X=. _ERROR_=0 _N_=5
```

Notice that the variable OLD_X holds on to the value from the previous iteration of the DATA step, and when X is missing (the third data value), the previous X value (2) is substituted for the missing value.

Generating Sequential SUBJECT Numbers Using a Retained Variable

Let's look at a very common programming requirement: adding sequential SUBJECT numbers to a set of data. As we did before, the first attempt will be without a RETAIN statement to demonstrate why a RETAIN statement is needed.

Program 1-5: Attempting to Generate a Sequential SUBJECT Number without Using a RETAIN Statement

```
***Attempting to generate a sequential SUBJECT number without
   using a RETAIN statement;
DATA WITHOUT_3;
   PUT "Before the INPUT statement: " _ALL_ ;
   INPUT X @@;
   SUBJECT = SUBJECT + 1;  ❶
   PUT "After the INPUT statement: " _ALL_ /;
DATALINES;
1 3 5
;
```

The programmer is trying to generate a subject number in line ❶. Before any data values (X's) have been read, SUBJECT (and X) are both set to missing. Therefore, when you attempt to add 1 to a missing value, the result is a missing value. The log below shows that SUBJECT is missing in every observation:

```
119  ***Attempting to generate a sequential SUBJECT number without
120     using an assignment statement;
121  DATA WITHOUT_3;
122     PUT "Before the INPUT statement: " _ALL_ ;
123     INPUT X @@;
124     SUBJECT = SUBJECT + 1;
125     PUT "After the INPUT statement: " _ALL_ /;
126  DATALINES;

Before the INPUT statement: X=. SUBJECT=. _ERROR_=0 _N_=1
After the INPUT statement: X=1 SUBJECT=. _ERROR_=0 _N_=1

Before the INPUT statement: X=. SUBJECT=. _ERROR_=0 _N_=2
After the INPUT statement: X=3 SUBJECT=. _ERROR_=0 _N_=2

Before the INPUT statement: X=. SUBJECT=. _ERROR_=0 _N_=3
After the INPUT statement: X=5 SUBJECT=. _ERROR_=0 _N_=3

Before the INPUT statement: X=. SUBJECT=. _ERROR_=0 _N_=4
```

A RETAIN statement will fix this problem. Besides retaining the value of SUBJECT, you need to supply an initial value as well. This is done in the RETAIN statement. If you follow the variable name with a number, the value of the retained variable will be initialized to that number. That is, it will continue to be the initialized value until it is replaced by another value. Program 1-6 shows the corrected version of this program with the RETAIN statement added.

Program 1-6: Adding a RETAIN Statement to Program 1-5

```
***The same program with a RETAIN statement;
DATA WITH_3;
   RETAIN SUBJECT 0;  ❶
   PUT "Before the INPUT statement: " _ALL_ ;
   INPUT X @@;
   SUBJECT = SUBJECT + 1;  ❷
   PUT "After the INPUT statement: " _ALL_ /;
DATALINES;
1 3 5
;
```

Notice that SUBJECT is retained and the initial value is set to 0 (line ❶). Therefore, the first time line ❷ is executed, you are adding 0 + 1 = 1 and assigning this value to SUBJECT. Since SUBJECT is retained, the next time line ❷ executes, you will be adding 1 to the previous value

(1) and SUBJECT will have a value of 2, and so forth. To be certain this is clear, you can inspect the log below:

```
131  *The same program with a RETAIN statement;
132  DATA WITH_3;
133     RETAIN SUBJECT 0;
134     PUT "Before the INPUT statement: " _ALL_ ;
135     INPUT X @@;
136     SUBJECT = SUBJECT + 1;
137     PUT "After the INPUT statement: " _ALL_ /;
138  DATALINES;

Before the INPUT statement: SUBJECT=0 X=. _ERROR_=0 _N_=1
After the INPUT statement: SUBJECT=1 X=1 _ERROR_=0 _N_=1

Before the INPUT statement: SUBJECT=1 X=. _ERROR_=0 _N_=2
After the INPUT statement: SUBJECT=2 X=3 _ERROR_=0 _N_=2

Before the INPUT statement: SUBJECT=2 X=. _ERROR_=0 _N_=3
After the INPUT statement: SUBJECT=3 X=5 _ERROR_=0 _N_=3

Before the INPUT statement: SUBJECT=3 X=. _ERROR_=0 _N_=4
```

The program is now working as planned.

Using a SUM Statement to Create SUBJECT Numbers

Creating SUBJECT numbers or other counters in a SAS DATA step is such a common operation, the thoughtful folks at SAS came up with a special SUM statement that makes this process easier. Look at Program 1-7, which uses a SUM statement to accomplish the same objective as Program 1-6.

Program 1-7: Demonstrating the SUM Statement

```
DATA WITHOUT_4;
   PUT "Before the INPUT statement: " _ALL_ ;
   INPUT X @@;
   SUBJECT + 1;  /* SUM statement */  ❶
   PUT "After the INPUT statement: " _ALL_ /;
DATALINES;
1 3 5
;
```

Line ❶ is a SUM statement. This is a funny-looking statement to people used to programming in other languages, because there is no equal sign. That is what makes this a SUM statement rather than an assignment statement. It accomplishes several objectives. It automatically retains the variable in the statement (SUBJECT here) and sets the initial value to 0. Inspection of the log below shows that it works as designed:

```
155  DATA WITHOUT_4;
156     PUT "Before the INPUT statement: " _ALL_ ;
157     INPUT X @@;
158     SUBJECT + 1;  /* SUM statement */
159     PUT "After the INPUT statement: " _ALL_ /;
160  DATALINES;

Before the INPUT statement: X=. SUBJECT=0 _ERROR_=0 _N_=1
After the INPUT statement: X=1 SUBJECT=1 _ERROR_=0 _N_=1

Before the INPUT statement: X=. SUBJECT=1 _ERROR_=0 _N_=2
After the INPUT statement: X=3 SUBJECT=2 _ERROR_=0 _N_=2

Before the INPUT statement: X=. SUBJECT=2 _ERROR_=0 _N_=3
After the INPUT statement: X=5 SUBJECT=3 _ERROR_=0 _N_=3

Before the INPUT statement: X=. SUBJECT=3 _ERROR_=0 _N_=4
```

Demonstrating That Variables Read with a SET Statement Are Retained

There is another way in which SAS implicitly retains variables. Variables read with a SET statement are automatically retained. Most of the time, we don't really need to think about this. However, there are times when this feature of the SET statement can be used to our advantage. The small demonstration program shown next creates a data set with one observation (X=1 and Y=2). In the DATA step that follows, a SET statement is conditionally executed. Look at Program 1-8 and the resulting log.

Program 1-8: Demonstrating That Variables from a SET Statement Are Retained

```
DATA ONE;
   INPUT X Y;
DATALINES;
1 2
;
```

```
DATA TWO;
   IF _N_ = 1 THEN SET ONE;
   PUT "Before INPUT statement: " _ALL_;
   INPUT NEW;
   PUT "After INPUT statement: " _ALL_ / ;
DATALINES;
3
4
5
;
```

Here is the SAS log from running this program:

```
171  DATA TWO;
172     IF _N_ = 1 THEN SET ONE;
173     PUT "Before INPUT statement: " _ALL_;
174     INPUT NEW;
175     PUT "After INPUT statement: " _ALL_ / ;
176  DATALINES;

Before INPUT statement: X=1 Y=2 NEW=. _ERROR_=0 _N_=1
After INPUT statement: X=1 Y=2 NEW=3 _ERROR_=0 _N_=1

Before INPUT statement: X=1 Y=2 NEW=. _ERROR_=0 _N_=2
After INPUT statement: X=1 Y=2 NEW=4 _ERROR_=0 _N_=2

Before INPUT statement: X=1 Y=2 NEW=. _ERROR_=0 _N_=3
After INPUT statement: X=1 Y=2 NEW=5 _ERROR_=0 _N_=3

Before INPUT statement:  X=1 Y=2 NEW=. _ERROR_=0 _N_=4
```

Notice that the values of X and Y remain at 1 and 2, respectively, in every observation in the data set TWO. Without the implicit retain feature of the SET statement, this would not work.

A Caution When Using a RETAIN Statement

There are some serious pitfalls that you can encounter when using the RETAIN statement. For example, suppose you want to read several observations from one SAS data set and create a single observation in a new data set. Under certain circumstances where you have missing values and you are using retained variables, you may make the mistake of using a retained value from a previous subject instead of a missing value for the present subject. We will demonstrate and discuss an example later in this book (see Program 7-7). So think of the RETAIN statement when you need to "remember" information from previous observations, but be especially cautious and test your programs carefully.