Retaining, Lagging, Leading, and Interleaving Data

Toby Dunn
Division of Performance Reporting, Texas Education Agency

Abstract

Many times a programmer must examine previous and/or future observations. When this occurs, the programmer is generally inclined to search the help docs for pre-defined statements, functions and techniques to create a quick solution to the immediate problem. However, this approach often leaves the programmer a headache, increased frustration and stress, and mistakes in their program. In an attempt to "go beyond the help docs," this paper examines frequently (mis)used techniques. Specifically, it addresses the retain statement and lag function as used to examine previous values, and the lead and interleaving techniques to look ahead at future values. The intent of this paper is not to provide the programmer with a comprehensive examination of these tools, but rather the fundamental understanding of their uses and subtleties required to make them an effective part of the programmer's tool chest. The intended audience is beginner to intermediate programmers with a sound foundation in Base SAS®.

Keywords: Retain statement, Lag function, Lead technique, Interleaving, DoW

Introduction

Every programmer finds at some point they need to look at a previous or future observation's value. It is at this point that they start looking at the Retain statement, LAG function, lead technique, or even interleaving a data set back upon itself. However, little do they realize that many of these have "gotchas" that befuddle or perplex the uninitiated programmer. This paper discusses some of the problems with using these tools; how to use them properly to look at a previous/future observation's variable value.

Discussion and Review

The Retain Statement

"The Retain statement does not retain."

What do *"Crack this"*, *"Help With Retain"*, and -- my personal favorite -- *"Pimpy Question"* have to do with the Retain statement? They all are all subject lines from posts to SAS-L from those who have tried -- and failed -- to get the Retain statement to work "properly." Most people think that the Retain statement does what its name implies; that is, it retains something. However, those initiated will quickly tell you that the Retain statement does not retain. That's right! **The Retain statement does not, will not, and (unless the underlying base SAS® code is rewritten) can never retain anything.** This fact is precisely where most uninitiated programmers programs go awry.

If the Retain statement doesn't retain, what exactly does it do? Logic suggests that it has to retain some variable's value from one iteration to the next, doesn't it? Not exactly. The Retain statement simply mimics retaining values by telling the SAS Supervisor not to reset certain variables to missing at the beginning of each iteration of the data step. (This is not to say that a retained variable cannot be overwritten -- it can and often does).

To better understand how the Retain statement mimics retaining values we need to look at what variables are set to missing and when. *"Missing Secrets"* (Jones and Whitlock) states that all data step variables can be put into one of four groups: unnamed variables, automatic variables, variables from a SAS® data set, and all other variables. The first group pertains to _temporary_ array variables; the second and third groups are automatically retained. Thus, our attention falls to the fourth and final group that includes, but is not limited to, variables from input statements, assignment statements, and function calls. Variables included in this group have values that must be reset to missing with each iteration of the data step. For their values to be retained, they must be explicitly stated in a Retain statement.

Now that we have identified the type of variables the Retain statement affects, it is time to investigate the effect of the Retain statement on these variables. To do this we need to look into how the data step works. The basic structure of the data step is as follows: 1) initialize the variables; 2) read a record; 3) do something with the data; 4) write out the transformed data; and 5) repeat steps 2 through 4 until all records have been read, processed and written. ("*A SAS Programmer's view of the SAS Supervisor*", Whitlock). Note that, in the basic data step structure, one observation gets read in, something is done to that observation, the observation is then written out. After the processing of that observation is complete, then the next observation gets read in and the process is repeated until there are no more observations.

As each record is read in the data step without the Retain statement, variables from input statements, assignment statements, and function calls are reset to missing. It is only these variables that are not implicitly retained because these are the only variables that will not necessarily have a value. The introduction of the Retain statement prevents these variable's values from being reset to missing.

So now that we know how and what variables the Retain statement affects, let us explore some examples to show the Retain statement in action. For these examples, we will be using the following data set named *one*.

Example 1 Data Set

x	y
1	a
2	a
.	a
4	a
1	b
.	b
.	b
1	c
2	c
1	d

Example 1

The most prevalent question I have seen regarding the Retain statement not working is from programmers trying to get a variable to copy the previous observation's value to the current observation's value when the current observation's value is missing or null.

Example 1 Code

```
data aaa;
   set one;
   by y;
   retain x;
run;

proc print
   data = aaa;
run;
```

Example 1 Output

Obs	x	y
1	1	a
2	2	a
3	.	a
4	4	a
5	1	b
6	.	b
7	.	b
8	1	c
9	2	c
10	1	d

As you can see from the output above, the previous value of x is not retained when the current value of x is missing. This is due to the fact that, while the value of x from the previous observation were actually retained by SAS®, it was also overwritten by the value of x from the current (missing) observation.

This begs the question: just how would a programmer fill in those missing values with the values of the previous observation? To explore this question, let us turn to Example 2.

Example 2

Well to do this a new variable will be created and dropped thus creating a temporary variable (if you will), to hold the value of the last observation. One important thing to notice is that we have by group processing going on, so we do not want the value of the last observation if that observation is not in the same by group as the current observation.

Example 2 Data Set

x	y
1	a
2	a
.	a
.	a
.	b
2	b
.	b
1	c
2	c
1	d

Example 2 Code

```
data bbb (drop = _:);
   set one;
   by y;
   retain _x;

   if (x ne .) then _x = x;
   if (x eq .) and (first.y ne 1) then x = _x;

run;
```

Example 2 Output

Obs	x	y
1	1	a
2	2	a
3	2	a
4	2	a
5	.	b
6	2	b
7	2	b
8	1	c
9	2	c
10	1	d

By conditionally setting the value to be retained, _x, and by then conditionally setting the current observation's value of x to the retained value, _x, you will get the desired result. By using the automatic variable `first.y` to control the conditional processing, we insure that when the retained value is not in the same by group as the current observation, it will not

overwrite the value in the current observation. Observation 5 shows that indeed we are doing just that.

The Lag Function

"There is a snag in the Lag…"

The Lag function, like the Retain statement, does not do what its name implies. One cannot expect the Lag function to perform a true lag. However, the Lag function can, like the Retain statement, in certain instances be tricked into acting like a true Lag function. So how does the Lag function perform? It creates a queue that is populated by values every time the Lag function is called and works off the FIFO or first in first out principle. As this is not immediately obvious, let us turn to an example for further clarification.

Example 3

Consider lagging a variable only if it is missing.

Example 3 Code

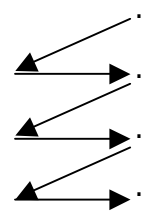
```
data ccc;
   set one;
   if x = . then x = lag(x);
run;
```

Example 3 Output

```
Obs   x  y

1  1  a
2  2  a
3  .  a
4  4  a
5  1  b
6  .  b
7  .  b
8  1  c
9  2  c
10 1  d
```

From the above code, you can see that x was not lagged when it was missing. So what happened? Recall that the Lag function creates a queue and populates it with values only when called. In our example, the first time the Lag function is called (i.e., the first time x has a missing value,) the queue only has a missing value in it since no other value had been passed to it yet. When the missing value for x is encountered, the lag function looks to the queue to get the lagged value (of missing) and passes it back to x. The queue stores the new (missing) value of x. Thus, the second time the Lag function is called, it populates x with the (missing) value from the last call of the Lag function and passes a (missing) value from x to the queue. The third time the Lag function is called the same process takes place.

Time Missing is Encountered	X	Queue
		
1st	.	
2nd	.	
3rd	.	

Example 4

So just how do you actually achieve a true lag? Glad you asked. If we were to assign a variable to the lag of x outside of the condition of x being missing, it would do just that. Thus the queue would not be populated with only missing values. Then we reassign that lagged (non-missing) value to x when x is missing.

Example 4 Code

```
data ccc;
   set one;
   z = lag(x);

   if x = . then x = z;
run;
```

Example 4 Output

Obs	x	y	z
1	1	a	.
2	2	a	1
3	2	a	2
4	4	a	.
5	1	b	4
6	1	b	1
7	.	b	.
8	1	c	.
9	2	c	1
10	1	d	2

As you can see from the above code, this indeed does the trick. However, you will notice when x has more than one missing value in consecutive observations, the result will be missing values as in the problem we encountered in the previous example. This is due to how values are passed and pulled from the queue created by the Lag function. To fix this problem, one can either use a retain statement with a lag (which is less efficient than the code in the previous section,) or one could simply tell SAS to lag x and then check for x having a missing value twice.

Example 5

No more than one observation was lagged in the previous example because the lag statement and the conditional re-assignment of x were performed only once. This means that the queue for the lag of x was only filled once and populated once per observation. Thus, missing values were passed to and taken from the queue of the lag of x. If we could perform this whole process once and then perform it again, we should be able to get a non-missing value in the queue for the lag of x thereby filling all observations beyond the first missing value of x to a non-missing

value. This can be accomplished by simply placing the lag and re-assignment of `x` inside a do loop set to iterate over the code twice.

Example 5 Code

```
Data one;
   Input x y $1.;
   cards;
      1  a
      .  a
      .  a
      2  b
      .  b
      3  b
      .  b
      .  b
      .  b
      .  b
run;

data ccc (drop = z twice);
   set one;
   by Y;

    do twice = 1 to 2;
      Z = lag(X);

      if X = . then X = Z;
   end;
run;
```

Example 5 Output

```
Obs   x  y

1     1  a
2     1  a
3     1  a
4     2  b
5     2  b
6     3  b
7     3  b
8     3  b
9     3  b
10    3  b
```

As you can see from the above output, by telling SAS to perform the code twice for each observation, we get the desired result. This is method does have one drawback: it has to loop through every observation twice. Thus, when the data set is sufficiently large enough performance maybe affected.

Example 6

Up until now we have only looked at one variable at a time, but in real life you will likely want to work with many variables. To do that, we will have to leave the relative comfort of SAS® statements and functions and use a few other SAS® concepts. First, it is common to set two datasets together in one data step, in essences stacking on top of the other. Also, we will need to use pointer controls. A pointer control here tells the SAS Supervisor what observation to get from the data set.

Example 6 Code

```
data one;
   input x $1. y;
   cards;
      a1
      a2
      a3
      b3
      b4
      c9
      d1
      d1
      d2
      d1   ;
run;
```

Example 6 Code (Continued)

```
data two;
   set one;
   by x;

   if first.x ne 1 then do;
       pnt = _N_ - 1;

       set one (rename = (x = _x
                          y = _y))
                       point = pnt;
   end;
   else do;
      _x = " ";
      _y = .;
   end;

run;
```

Example 6 Output

Obs	x	y	_x	_y
1	a	1	.	
2	a	2	a	1
3	a	3	a	2
4	b	3	.	
5	b	4	b	3
6	c	9	.	
7	d	1	.	
8	d	1	d	1
9	d	2	d	1
10	d	1	d	2

Thus from the above output one can see that we have essentially lagged both variables x and y by one observation. Thus, allowing for further processing if you desire.

The Lead Technique

"Lead your data."

I know the quote above sounds a little weird, but indeed in the case of SAS® it is sometimes true. In a SAS® Talk paper discussing the lead technique, Ian Whitlock once asked "Well, if the Lag function merely remembers the last argument then the opposite must predict the next argument used and that, of course, cannot be done. Do you know why?" The astute reader will remember the basic outline of how the data step works from the Retain section of this paper. (In that outline one observation comes in; something is done to it; it is written out to the output data set; and this is repeated for future observations.) In this framework, it is impossible to know what a future value of an observation will be. If we can't know what the next observation's values will be then how can we have a lead technique? Next we explore the basic techniques to achieve this: the lead technique and interleaving a dataset back with itself.

Example 7

In practice, when someone says they want a lead function, what they really want is to be able to see what the next observations values will be. When we set a data set into a data step it will bring in one observation at a time. What happens when we set that same data set twice?

Example 7 Code

```
data one;
   a = 1;
   b = 2;
run;

data two;
   set one;
   set one (rename = (a = _a b = _b));
run;
```

Example 7 Output

Obs	a	b	_a	_b
1	1	2	1	2

You will notice that I renamed the variables coming from the second data set with an underscore. This is done so that the second set statement does not over write the variables from the first set statement. By setting a data set twice in this way, we get two sets of the exact same variables.

Example 8

Now suppose that we had some way of setting the second data set so that it only pulled the next observations values in, instead of the current one.

Example 8 Code

```
data one;
   input x y $1.;
   cards;
   1 a
   2 b
   3 c
   4 d
   5 e   ;
run;

data two;
   set one nobs = number_of_total_obs;

   pointer = _N_ + 1;

   if 1 <= pointer
        <= number_of_total_obs then
   set one (keep = x y
          rename = (x = _x
                    y = _y))
           point = pointer;
   else do;
      _x = .;
      _y = " ";
   end;
run;
```

Example 8 Output

Obs	x	y	_x	_y
1	1	a	2	b
2	2	b	3	c
3	3	c	4	d
4	4	d	5	e
5	5	e	.	.

Example 9

Now that we have shown that you can in fact create a lead variable, what about the case where you want to examine all future observations. To help make this example more realistic and manageable, let's add-in by group processing. Consider the case where you have a data set of class schedules.

Example 9 Code A

```
data one;
   input id $1. class $7. time $5.;
   cards;
      1  math     01:00
      1  english  02:00
      1  science  03:00
      1  writing  01:00
      1  social   02:00
      2  english  01:00
      2  math     02:00
      2  writing  01:00
      3  math     01:00
      4  english  01:00
      4  math     01:00
      4  writing  01:00
      5  english  02:00
   ;
run;
```

Now let's suppose that you want to find out if two classes overlap in time by id. Once again we will deploy the use of the lead technique. However, instead of the normal lead technique, we will need to wrap the second set statement inside a do-loop so that all future observations by variable (id) will be considered.

Example 9 Code B

```
data overlap (drop = pid pclass ptime first_id);
   set one_s;
   by id;

   retain first_id;

   if first.id = 1 then first_id = _N_;

   if _N_ > first_id then do;

     do i = (_N_ - 1) to first_id by -1;
        set one_s (rename = (id = pid
                             class = pclass
                              time = ptime))
                      point = i;

      if ((pid = id) and (ptime = time)) then overlap = pclass;
     end;
   end;
run;
```

Example 9 Output

Obs	id	Class	Time	Overlap
1	1	math	01:00	
2	1	english	02:00	
3	1	science	03:00	
4	1	writing	01:00	math
5	1	social	02:00	english
6	2	english	01:00	
7	2	math	02:00	
8	2	writing	01:00	english
9	3	math	01:00	
10	4	english	01:00	
11	4	math	01:00	english
12	4	writing	01:00	english
13	5	english	02:00	

Interleaving A Dataset Back On Itself

I first saw this technique explained in a paper entitled *"Interleaving a Dataset with Itself: How and Why?"* by Howard Schreier . This paper and use of the technique in examples on SAS-L made it popular. This technique involves interleaving (a common well known technique found in the SAS® documentation) with itself. The reason for doing this is to get a cumulative total by some by group and have the total attach itself to the data being interwoven.

Example 10

A practical example would be getting aggregate level data merged with disaggregated data.

Example 10 Code

```
data one;
   input x $1. y;
   cards;
      a1
      a2
      a3
      b3
      b4
      c9
      d1
      d1
      d2
      d1
   ;
run;

data two;
   set one (in = first_pass)
       one (in = second_pass);
   by x;
```

Example 10 Output

Obs	x	y	cum_cnt
1	a	1	6
2	a	2	6
3	a	3	6
4	b	3	7
5	b	4	7
6	c	9	9
7	d	1	5
8	d	1	5
9	d	2	5
10	d	1	5

Example 10 Code Continued

```
   if first_pass then do;
     if first.x then cum_cnt = .;
     cum_cnt + y;
   end;

   if  second_pass then do;
     output;
   end;

run;
```

Example 11

The same thing can be accomplished by simply using a SQL statement.

Example 11 Code

```
proc sql;
 create table two as
  select *, sum(y) as cum_cnt
   from one
    group by x
     order by x;
quit;
```

Example 11 Output

Obs	x	y	cum_cnt
1	a	1	6
2	a	2	6
3	a	3	6
4	b	3	7
5	b	4	7
6	c	9	9
7	d	1	5
8	d	1	5
9	d	2	5
10	d	1	5

When using the SQL method, you have the added benefit of the `order by` statement (which is the same as `proc sort` in SAS.) However, the down side is that you and those coming after you will have to know SQL and if you want to further process your data after the sum of variable (y) you will have to have a more complex SQL statement.

DoW

I know one or more of you may be saying "Hey, I need to get aggregated data attached to disaggregated date, but the SQL statement is a whole thing to learn and I don't want to have to worry about whether or not I reset the count variable every time a new by-group is encountered with the Schreier interleave technique." There is hope for you yet with the DoW loop. No, no, not the Dow Jones Industrial Average, but the Do-Loop of Whitlock and more to the point a double Dow.

Recall the basic structure of the data step and how the data flow through it. Ian Whitlock some time back realized that we can use that to our advantage and set data in a data step inside a do-loop. This has many added advantages and, in our case, adds some structural integrity to our problem.

Example 12

Example 12 Code

```
data two;

   do i = 1 by 1 until (last.x);
      set one;
      by x;
      cum_cnt = sum(cum_cnt, y, 0);
   end;

   do _n_ = 1 to i;
      set one;
      output;
   end;

run;
```

Example 12 Output

Obs	x	y	cum_cnt
1	a	1	6
2	a	2	6
3	a	3	6
4	b	3	7
5	b	4	7
6	c	9	9
7	d	1	5
8	d	1	5
9	d	2	5
10	d	1	5

This yields the same output data set as the last methods. This time however, we do not have to reset the cumulative every time we encounter a new by group. This method does, however, have one major drawback: it will fail when where = filters are used one the incoming dataset. When that is the case, the Schreier interleaving method is the preferred method by this author. Its structure is better a accommodating such asymmetrical data structures.

Conclusions

We have seen that there are many ways to look not only back but forward through your datasets. There are three major considerations that will affect what technique you use. First, know your data structure. I have trivialized the examples because I wanted to highlight the technique and some of their associated pitfalls, not certain examples. Secondly, fully understand your problem before you decide on a technique. As I have tried to demonstrate here, each method has its advantages and disadvantages that should dictate when you use them. So know what needs to be done and then choose the appropriate method to obtain your results. And lastly, understand your chosen method fully. As I did not want to write a book, I haven't fully discussed each and every technique or method in this paper; it would simply be too long. But I hope that I have inspired you to not be afraid of looking forwards and backwards through your data. I have added a further reading section at the end of this paper for those interested in learning more about these techniques.

Notes

Special Thanks

To my wife, Sara, without whose patience and editing skills this paper would have not been completed.

Also, I would like to thank Ron Fehd and Howard Schreier, for being ever so patient with me and reviewing this paper. Without their keen eye and knowledge, many mistakes would have slipped through.

SAS-L and all those wonderful people on there (there are too many individuals to list!) whose knowledge and patience has enlightened me on more than one occasion. Their questions and knowledge sparked the writing of this paper.

Further Reading and Resources

SAS-L

http://www.listserv.uga.edu/archives/sas-l.html

Data Step

"A SAS Programmer's View of the SAS Supervisor" by Ian Whitlock
http://www2.sas.com/proceedings/sugi22/ADVTUTOR/PAPER34.PDF

"Missing Secrets" by Robin Jones and Ian Whitlock
Presented at the 2002 SESUG (poster)

Retain Statement

"The RETAIN Statement: One Window into the SAS Data Step" by Paul Gorrell
http://www.ats.ucla.edu/stat/sas/library/nesug99/bt064.pdf

Lag Function

SAS Talk Paper DCSUG by Ian Whitlock
http://dc-sug.org/DCjun03.pdf

"Lag With a WHERE" and Other DATA Step Stories" by Neil Howard

Interleaving a Data Set with Itself

"Interleaving a DataSet with Itself: How and Why?" by Howard Schreier
http://www.nesug.org/html/Proceedings/nesug03/cc/cc002.pdf

Lead technique and DoW

SAS Talk Paper DCSUG by Ian Whitlock
http://dc-sug.org/DCjun03.pdf

"The Magnificent Do" by Paul Dorfman

Contact Information

Author: Toby Dunn **Email:** tdunn@tea.state.tx.us

Address: 1701 North Congress Ave.
Division of Performance Reporting C/O Toby Dunn
Austin, TX 78701