Retaining, Lagging, Leading, and Interleaving Data

Toby Dunn, Manor, TX
Chang Y. Chung, Princeton, NJ

ABSTRACT

Many times a programmer must examine previous and/or future observations. When this happens, 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 SAS®/BASE.

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. 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.

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 the 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 SAS/BASE 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* the 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 be and often is).

To better understand how the RETAIN statement mimics retaining values, we need to look at what variables are set to missing and when. Jones and Whitlock (2002) observe 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 RETAIN'ed. 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 implicit DATA step loop. For their values to be "retained" (i.e., not-to-be-reset-to-missing), 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 (See Whitlock(1997)). 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 examples, we show a input data set on left, the output data set on right, along with the `DATA` step code that takes the input and creates the output.

EXAMPLE 1

The most prevalent question we have seen regarding the `RETAIN` statement 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.

In01		Example 1		Out01	

```
Obs     x

 1      1
 2      2
 3      .
 4      4
 5      5
 6      .
 7      .
 8      8
```

```
data Out01;
  set In01;
  retain x;
run;
```

```
Obs     x

 1      1
 2      2
 3      .
 4      4
 5      5
 6      .
 7      .
 8      8
```
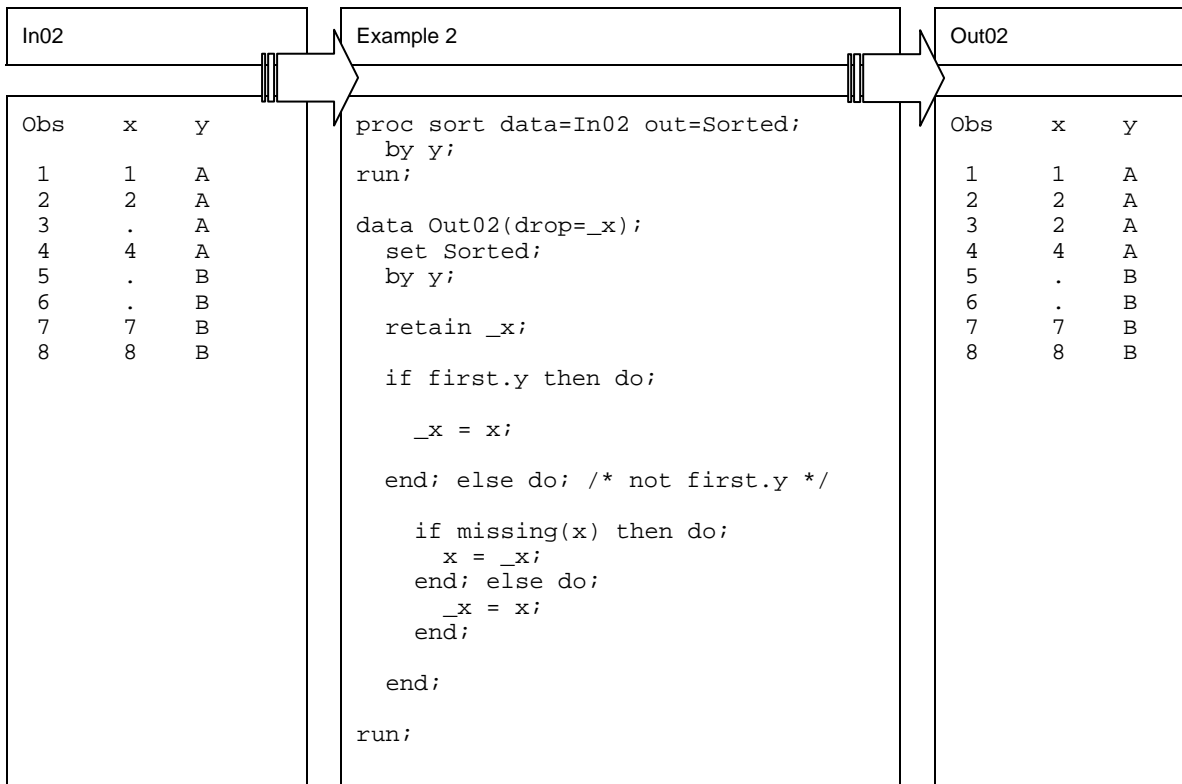
As you can see from the output above, the previous value of `x` is not retained when the current value of `x` is missing. This is due to the fact that, while the value of `x` from the previous observation was actually retained by SAS, it was also over-written by the value of `x` from the current observation (i.e., missing).

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

In02			Example 2		Out02		

```
Obs     x     y

 1      1     A
 2      2     A
 3      .     A
 4      4     A
 5      .     B
 6      .     B
 7      7     B
 8      8     B
```

```
proc sort data=In02 out=Sorted;
  by y;
run;

data Out02(drop=_x);
  set Sorted;
  by y;

  retain _x;

  if first.y then do;

    _x = x;

  end; else do; /* not first.y */

    if missing(x) then do;
      x = _x;
    end; else do;
      _x = x;
    end;

  end;

run;
```

```
Obs     x     y

 1      1     A
 2      2     A
 3      2     A
 4      4     A
 5      .     B
 6      .     B
 7      7     B
 8      8     B
```

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.

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 ensure 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.
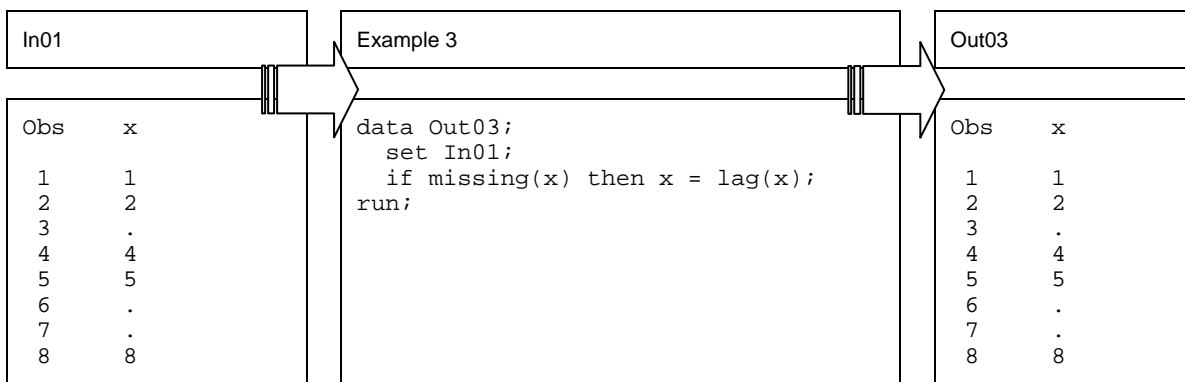
THE LAG FUNCTION

"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 FIFO (first-in, first-out) queue that is populated by a value. As this is not immediately obvious, let us turn to an example for further clarification.
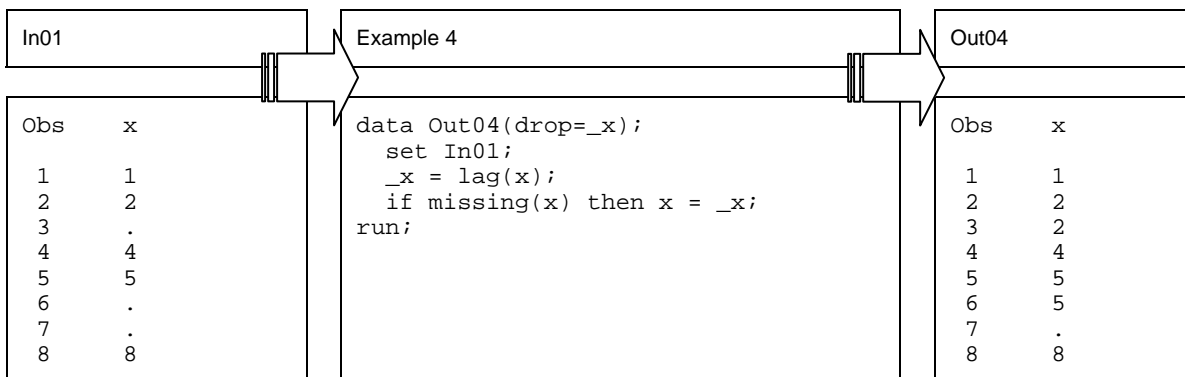
EXAMPLE 3
Consider lagging a variable only if it is missing.

In01		Example 3	Out03	
Obs	x		Obs	x
| 1 | 1 | ```
data Out03;
  set In01;
  if missing(x) then x = lag(x);
run;
``` | 1 | 1 |
2	2		2	2
3	.		3	.
4	4		4	4
5	5		5	5
6	.		6	.
7	.		7	.
8	8		8	8

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 a value *only* when called. In our example, the first time the LAG function is called (i.e., the first time X has a missing value, observation 3) 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. Only the missing values go in and come out as the result.

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. The function should be called once for each and every observation and the queue would not be populated with only missing values (unless all the values are missing!). Then we reassign that lagged value to X when X is missing.

In01		Example 4	Out04	
Obs	x		Obs	x
| 1 | 1 | ```
data Out04(drop=_x);
  set In01;
  _x = lag(x);
  if missing(x) then x = _x;
run;
``` | 1 | 1 |
2	2		2	2
3	.		3	2
4	4		4	4
5	5		5	5
6	.		6	5
7	.		7	.
8	8		8	8

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 the LAG function lags by one (LAG2 function lags by two, and so on). To fix this problem, one can use either a RETAIN statement as shown earlier, or multiple lags (in this case, use both LAG and LAG2).
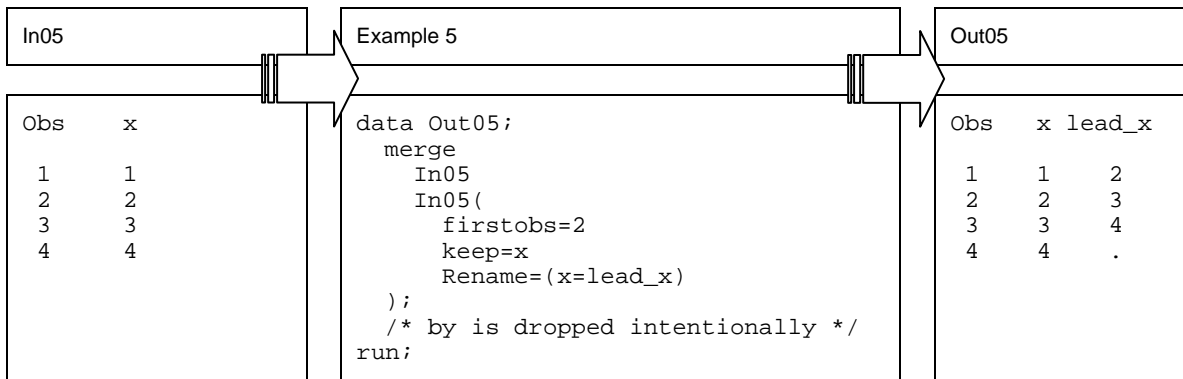
THE LEAD TECHNIQUE

"Lead your data."

I know the quote above sounds a little weird, but indeed in the case of SAS it sometimes is true. In a SAS Talk paper (Whitlock 2003) discussing the lead technique, Ian Whitlock asks "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 the value(s) in a future observation. 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.
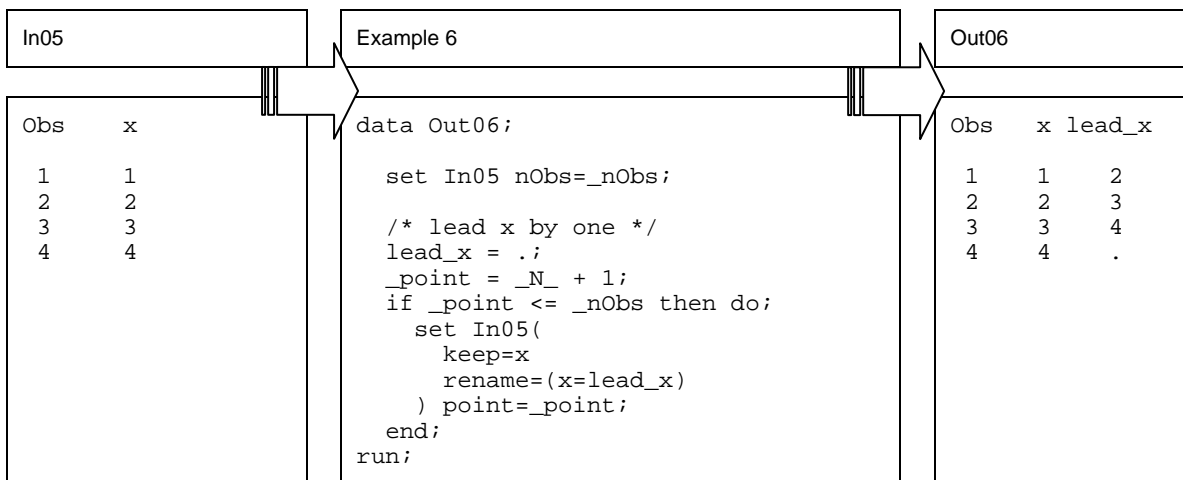
EXAMPLE 5
A classic way of doing the "lead" a variable is MERGE'ing the data set with itself with *no* BY variable.

In05	Example 5	Out05

Obs   x
 	
1   1
2   2
3   3
4   4
	`data Out05;`
	`  merge`
	`    In05`
	`    In05(`
	`      firstobs=2`
	`      keep=x`
	`      Rename=(x=lead_x)`
	`  );`
	`  /* by is dropped intentionally */`
	`run;`
		Obs x lead_x
		
		1 1 2
		2 2 3
		3 3 4
		4 4 .

EXAMPLE 6
When the data set is not huge, the POINT= option in the SET statement comes in very handy. It basically allows us to fetch *any* observation with the observation number in any order. For example, a simple "lead" as shown in the Example 5 can be implemented as below:
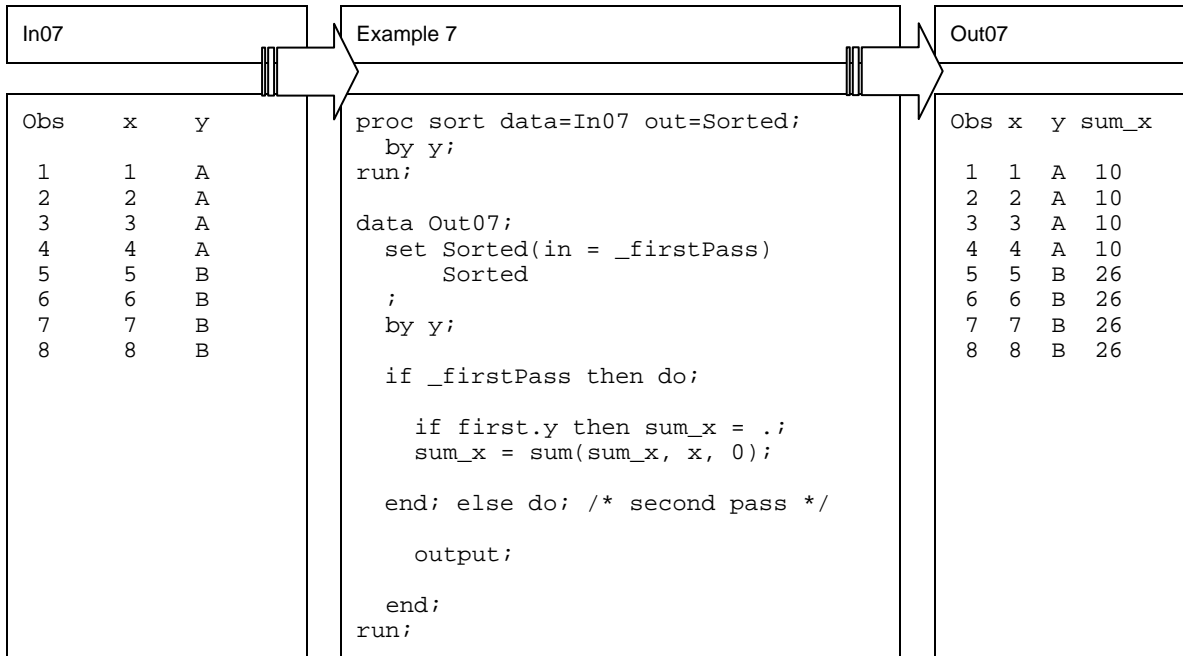
In05	Example 6	Out06

Obs   x
 	
1   1
2   2
3   3
4   4
	`data Out06;`
	
	`  set In05 nObs=_nObs;`
	
	`  /* lead x by one */`
	`  lead_x = .;`
	`  _point = _N_ + 1;`
	`  if _point <= _nObs then do;`
	`    set In05(`
	`      keep=x`
	`      rename=(x=lead_x)`
	`    ) point=_point;`
	`  end;`
	`run;`
		Obs x lead_x
		
		1 1 2
		2 2 3
		3 3 4
		4 4 .

SAS is smart enough to automatically DROP the temporary pointer variable, _POINT. (In fact, it will issue a warning if you DROP it explicitly.) You have to reset the LEAD_X variable (to missing) for each observation yourself, because the LEAD_X value of the next to the last observation will be carried over to last observation otherwise.

INTERLEAVING A DATASET BACK ON ITSELF

One of us first saw this technique explained in a paper entitled "Interleaving a Dataset with Itself: How and Why?" by Howard Schreier(2003). This paper and SAS-L posts made it popular. This technique involves taking a well-documented technique of interleaving observations and creatively applying it – to interleave one data set with itself. It is convenient when we try to calculate by-group summary statistics and to attach them back to the member observations.
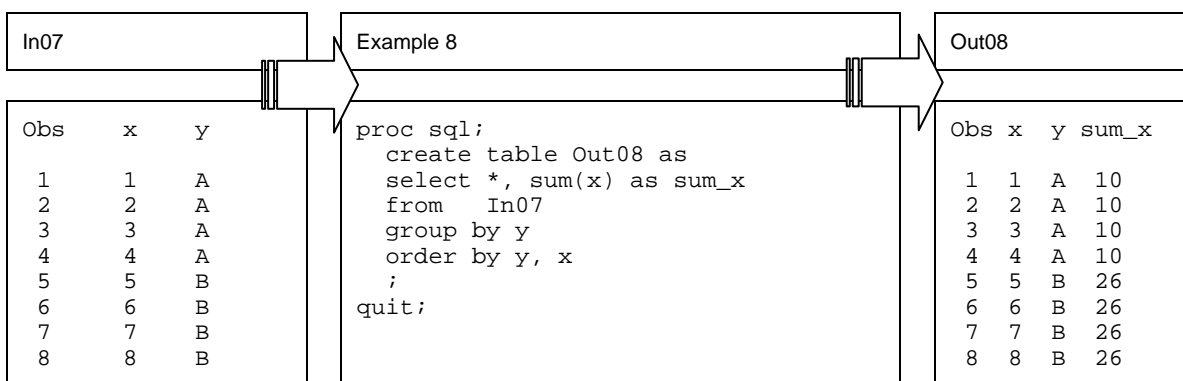
EXAMPLE 7

The following calculate and attach the sum of x values within each by-group.

In07			Example 7	Out07			

```
Obs     x     y
 1      1     A
 2      2     A
 3      3     A
 4      4     A
 5      5     B
 6      6     B
 7      7     B
 8      8     B
```

```
proc sort data=In07 out=Sorted;
  by y;
run;

data Out07;
  set Sorted(in = _firstPass)
      Sorted
  ;
  by y;

  if _firstPass then do;

    if first.y then sum_x = .;
    sum_x = sum(sum_x, x, 0);

  end; else do; /* second pass */

    output;

  end;
run;
```

```
Obs x  y sum_x
 1  1  A  10
 2  2  A  10
 3  3  A  10
 4  4  A  10
 5  5  B  26
 6  6  B  26
 7  7  B  26
 8  8  B  26
```
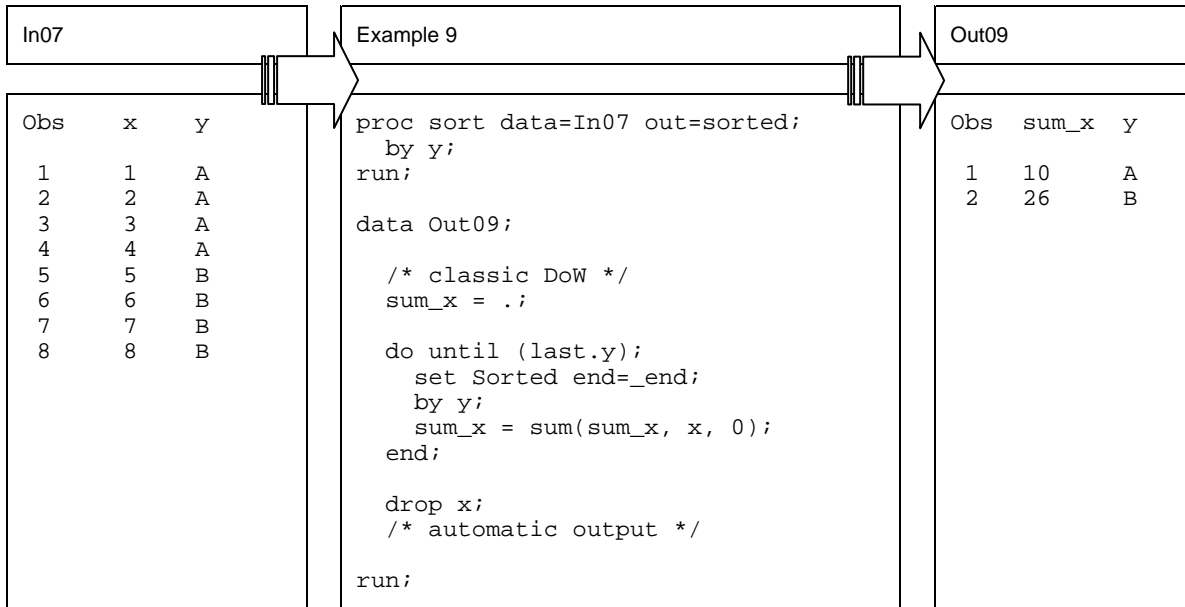
EXAMPLE 8

The same thing can be accomplished by simply using PROC SQL, thanks to PROC SQL's automatic re-merging the summary statistics back to the observations under certain conditions (for instance, when we SELECT non-CALCULATED fields as well).

In07			Example 8	Out08		

```
Obs     x     y
 1      1     A
 2      2     A
 3      3     A
 4      4     A
 5      5     B
 6      6     B
 7      7     B
 8      8     B
```

```
proc sql;
  create table Out08 as
  select *, sum(x) as sum_x
  from   In07
  group by y
  order by y, x
  ;
quit;
```

```
Obs x  y sum_x
 1  1  A  10
 2  2  A  10
 3  3  A  10
 4  4  A  10
 5  5  B  26
 6  6  B  26
 7  7  B  26
 8  8  B  26
```

DOW

I know one or more of you may be saying "Hey, I need to get aggregated data attached to disaggregated date, but the PROC SQL 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 Self-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 (also known as the Whitlock Do-Loop). The DoW is commonly used to BY-group process in natural way. For example, aggregating a dataset by calculating the sum of the variables can be done using DoW.
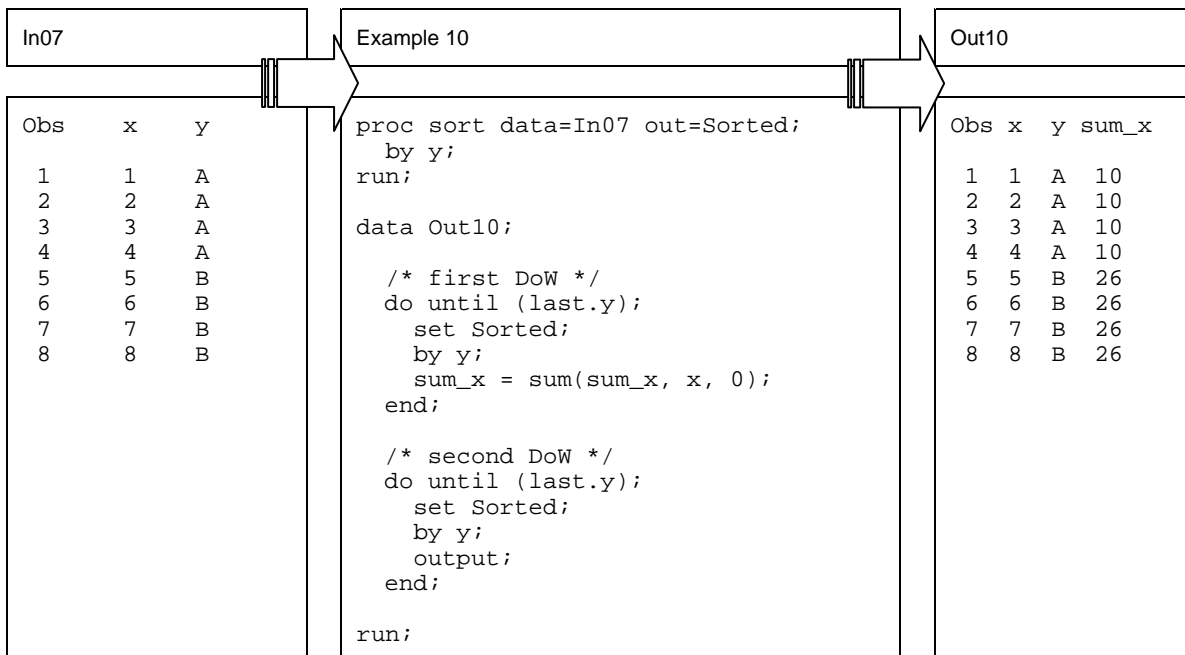
EXAMPLE 9

In07		
Obs	x	y
1	1	A
2	2	A
3	3	A
4	4	A
5	5	B
6	6	B
7	7	B
8	8	B

Example 9

```
proc sort data=In07 out=sorted;
  by y;
run;

data Out09;

  /* classic DoW */
  sum_x = .;

  do until (last.y);
    set Sorted end=_end;
    by y;
    sum_x = sum(sum_x, x, 0);
  end;

  drop x;
  /* automatic output */

run;
```

Out09		
Obs	sum_x	y
1	10	A
2	26	B

The DoW is characterized by the SET statement wrapped by a DO loop and the automatic OUTPUT done once for each value of the BY-variable. The same thing can be done using a more traditional technique of (1) SET the data set near the top of the DATA step; (2) initialize if FIRST.BY-variable is true; (3) do something for each observation; then finally; (4) OUTPUT when LAST.BY-variable is true. So what is the big deal about DoW? As Dorfman(2002) so eloquently puts it: "It is all in the logic. The DoW-loop programmatically separates the before-, during-, and after-group actions in the same manner and sequence as does the stream-of-the-consciousness logic […]"

EXAMPLE 10
But how about merging the aggregate statistics back to the original data set? For this, the expert consensus seems to be using PROC SQL is the shortest way. If we stick to DoW, then there also is a variation called "Double DoW," shown below. Here the first DoW reads one BY-group observation at a time and computes the sum of X. The second DoW loop reads the same group and OUTPUT the observations with the calculated BY-group statistic (SUM_X) attached to each observation. Neat, isn't it?

In07		
Obs	x	y
1	1	A
2	2	A
3	3	A
4	4	A
5	5	B
6	6	B
7	7	B
8	8	B

Example 10

```
proc sort data=In07 out=Sorted;
  by y;
run;

data Out10;

  /* first DoW */
  do until (last.y);
    set Sorted;
    by y;
    sum_x = sum(sum_x, x, 0);
  end;

  /* second DoW */
  do until (last.y);
    set Sorted;
    by y;
    output;
  end;

run;
```

Out10			
Obs	x	y	sum_x
1	1	A	10
2	2	A	10
3	3	A	10
4	4	A	10
5	5	B	26
6	6	B	26
7	7	B	26
8	8	B	26

CONCLUSIONS

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 is your data. We have trivialized the examples because we wanted to highlight the technique and some of their associated pitfalls. In reality, data are much larger and more complicated. Without knowing your data, you will get lost even when you are well-equipped with fancy programming technique. Second is the problem that you want to solve. Before you decide on a technique, you have to understand your problem better. As demonstrated, each method has its advantages and disadvantages. So, know what needs to be done. Then, only then, choose the appropriate method to obtain your results. And lastly, understand your chosen method fully. As a paper, not a book, this paper does not cover each and every technique fully. We just hope that we inspired you enough not to be intimidated by data step code that looks forwards and backwards. Better yet, we hope you try to take advantage of these and other programming technique and solve your data problems a little bit more easily. We have added a further reading section at the end of this paper for those who are interested in learning more about these techniques.

REFERENCES

Dorfman, Paul, (2002), "The Magnificent Do," Proceedings of The SouthEast SAS Users Group (SESUG) Conference 2002. <http://www.devenezia.com/papers/other-authors/sesug-2002/TheMagnificentDO.pdf>

Gorrell, Paul. (1999). "The RETAIN Statement: One Window into the SAS DATA Step," *Proceedings of NorthEast SAS Users Group(NESUG) Conference 1999.* <http://www.ats.ucla.edu/stat/sas/library/nesug99/bt064.pdf>

Jones, Robin and Ian Whitlock. (2002), "Missing Secrets," Paper PS05. *Proceedings of the SouthEast SAS Users Group (SESUG) Conference.*

Schreier, Howard, (2003), ""Interleaving a Data Set with Itself: How and Why?" Proceedings of NorthEast SAS Users Group (NESUG) Conference 2003. <http://www.nesug.org/html/Proceedings/nesug03/cc/cc002.pdf>

Schreier, Howard, (2004), "Re: Help with By group processing, thanks." A SAS-L email discussion group post on Jun 24, 2004. Archived at UGA < http://www.listserv.uga.edu/cgi-bin/wa?A2=ind0406D&L=sas-l&D=1&H=0&O=D&T=1&P=30739>

Whitlock, Ian. (1997), "A SAS Programmer's View of the SAS Supervisor," *Proceedings of the Twenty-Second Annual SAS[®] Users Group International Conference,* 22, 170-179. <http://www2.sas.com/proceedings/sugi22/ADVTUTOR/PAPER34.PDF>

Whitlock, Ian, (2003), "SAS Talk," A column in the Washington, DC SAS Users Group (DCSUG) Newsletter. Second Quarter, 2003. <http://dc-sug.org/DCjun03.pdf>

FURTHER READING

General Interest:

SAS-L is a very active e-mail discussion group about SAS. Nice web access to an archive is found at <http://www.listserv.uga.edu/archives/sas-l.html>.

Whitlock (1999) is a must read for serious SAS users.

On Retain:

Read Jones and Whitlock (2002) for learning about four different types of SAS variables. Did you know that SAS distinguishes between setting a value to missing at the top of the data step loop from setting a value to missing when data is not read?

Gorrell (1999) has nice tutorial with lots of examples.

On `LAG`ging:

Whitlock(2003) is short but sweet and it is exclusively on the `LAG` function.

On Interleaving:

Schreier (2003) is widely considered as the inventor of the technique of this "Schreier Self-interleave," even though he rather humbly credits Bob Vigile for teaching him the technique of merging a dataset with itself. See the SAS-L thread regarding this topic archived at Schreier(2004).

On DoW:

Even though Whitlock invented the DoW, the classic paper on Dow is by Dorfman(2002). See also Whitlock(2003).

CONTACT INFORMATION

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

Toby Dunn
11604 Marshal St
Manor, Texas 78653
(512) 289-1824
tobydunn@hotmail.com

Chang Y. Chung
Princeton University
#216 Wallace Hall
Princeton, NJ 08544
(609) 258-2360
cchung@princeton.edu
http://changchung.com/