# Conditional Lags Don't Have to be Treacherous

Howard Schreier, Howles Informatics, Arlington VA


## ABSTRACT

The LAG and DIF functions are useful, but they are often not well understood. In particular, their interactions with conditional processing have long been a source of confusion. In SAS®9, new inline conditional functions (IFC and IFN) make it easier to use LAG and DIF functions appropriately.


## DATA FOR THE EXAMPLE

We'll use an example to illustrate the behavior and use of the LAG function (we won't explicitly discuss DIF, but its behavior parallels that of LAG with regard to the issues and techniques to be discussed). Suppose that there is some kind of competition, with two competitors (A and B), three rounds of competition, and a score for each competitor in each round. We run the following DATA step to load made-up data into a SAS data set.

```
data demo;
input ID $ Round Score;
cards;
A 1 21
A 2 31
A 3 29
B 1 23
B 2 30
B 3 33
;
```

The table DEMO looks like this:

| Obs | ID | Round | Score |
|---|---|---|---|
| **1** | A | 1 | 21 |
| **2** | A | 2 | 31 |
| **3** | A | 3 | 29 |
| **4** | B | 1 | 23 |
| **5** | B | 2 | 30 |
| **6** | B | 3 | 33 |

## THE PROBLEM

Now suppose that we want to have each competitor's previous-round score juxtaposed with his or her current score, to facilitate comparison. We utilize the LAG function to do this:

```
data prevscores;
set demo;
PrevScore = lag(score);
run;
```

Here is the result:

| Obs | ID | Round | Score | PrevScore |
|---|---|---|---|---|
| **1** | A | 1 | 21 | . |
| **2** | A | 2 | 31 | 21 |
| **3** | A | 3 | 29 | 31 |
| **4** | B | 1 | 23 | 29 |
| **5** | B | 2 | 30 | 23 |
| **6** | B | 3 | 33 | 30 |

The only problem is the value of 29 in the fourth observation. Since that is the first observation for competitor B, there should be a missing value instead. So we'll take what would seem to be the obvious corrective measure and code an IF statement to prevent values from appearing in the first observation for each competitor.

```
data prevscores;
set demo;
by id;
if not first.id then PrevScore = lag(score);
run;
```

Now the output data set looks like this:

| Obs | ID | Round | Score | PrevScore |
|---|---|---|---|---|
| **1** | A | 1 | 21 | . |
| **2** | A | 2 | 31 | . |
| **3** | A | 3 | 29 | 31 |
| **4** | B | 1 | 23 | . |
| **5** | B | 2 | 30 | 29 |
| **6** | B | 3 | 33 | 30 |

The conditional effect of the LAG function succeeded in keeping values from appearing in the first observation for each competitor. However, one of our correct values has disappeared, and another correct value has been replaced by an incorrect one. It's time to examine exactly what the LAG function is doing.

The SAS documentation states that the LAG function "stores a value in a queue and returns a value stored previously in that queue". Since we are working with the simple LAG (as distinguished from second, third, and higher lags), the queue has a length of precisely 1 (one). This table repeats the results shown immediately above and also traces the actions of the LAG function:

| Obs | ID | Round | Score | Prev Score | First.ID | Is LAG Called? | LAG Returns What? | LAG Stores What? |
|---|---|---|---|---|---|---|---|---|
| 1 | A | 1 | 21 | . | 1 | No | | |
| 2 | A | 2 | 31 | . | 0 | Yes | . | 31 |
| 3 | A | 3 | 29 | 31 | 0 | Yes | 31 | 29 |
| 4 | B | 1 | 23 | . | 1 | No | | |
| 5 | B | 2 | 30 | 29 | 0 | Yes | 29 | 30 |
| 6 | B | 3 | 33 | 30 | 0 | Yes | 30 | 33 |

The first column simply counts the observations as they are processed. The next five columns reflect values stored in the DATA step's program data vector. The last 3 columns explain the behavior of the LAG function; specifically

- whether it is called,
- what value it returns from its queue, and
- what value it stores in its queue for later use.

We can make some observations:

- The LAG function is not called when the DATA step is processing the first observation for a competitor. That's the direct effect of the IF statement.
- When the LAG function is called, it stores the current value of SCORE in its queue.
- When the LAG function is called for the first time, it returns a missing value. On subsequent calls, it returns the last value it stored in its queue. It does not matter how many iterations of the DATA step have occurred, or how many observations have been processed, since that value was stored.

We can now see that the LAG function is not a simple "look-back" device to access variable values from the immediately preceding observation of a data set. It sometimes behaves as if it were, but conditional execution clearly interferes with that.

So how do we get the correct results? We can be wishful and run this code:

```
data prevscores;
set demo;
by id;
PrevScore = lagby(score);
run;
```

Here we posit a function, LAGBY, which works like LAG but which reinitializes its queue to missing values each time a BY-group boundary is crossed. That would neatly eliminate the problem. Unfortunately, the LAGBY function does not exist, so the result appearing in the log is

```
      prevscore = lagby(score);
                  -----
                  68
ERROR 68-185: The function LAGBY is unknown, or cannot be ac-
cessed.
```

We will instead have to craft a solution.

### WELL-KNOWN SOLUTIONS

The essence of the problem is to have the LAG function **called** for each observation, even when the returned result is not needed. If the LAG function is **called** for each observation, it will always **return** a value from the immediately preceding observation, which is what we need. There are two proven approaches.

The first technique is to call the LAG function unconditionally, place the value it returns in a variable, but over-write it with a missing value when it is unwanted or inappropriate. In our example, we would have

```
data prevscores;
set demo;
by id;
PrevScore = lag(score);
if first.id then PrevScore = . ;
run;
```

The output is:

| Obs | ID | Round | Score | PrevScore |
|---|---|---|---|---|
| **1** | A | 1 | 21 | . |
| **2** | A | 2 | 31 | 21 |
| **3** | A | 3 | 29 | 31 |
| **4** | B | 1 | 23 | . |
| **5** | B | 2 | 30 | 23 |
| **6** | B | 3 | 33 | 30 |

While the fourth observation is processed, PREVSCORE will, for just an instant, contain the value 29.

The other remedy is to unconditionally invoke the LAG function, but to store its result in a throwaway variable, the value of which is **conditionally** assigned to the actual target variable. The code would be like this:

```
data prevscores;
set demo;
by id;
lagscore = lag(score);
if not first.id then PrevScore = lagscore;
drop lagscore;
run;
```

Again, the result is:

| Obs | ID | Round | Score | PrevScore |
|---|---|---|---|---|
| **1** | A | 1 | 21 | . |
| **2** | A | 2 | 31 | 21 |
| **3** | A | 3 | 29 | 31 |
| **4** | B | 1 | 23 | . |
| **5** | B | 2 | 30 | 23 |
| **6** | B | 3 | 33 | 30 |

### A New and Better Solution

SAS9 offers two new functions which can be put to use in problems like the one we have been exploring. The IFC and IFN functions evaluate TRUE/FALSE conditions and branch accordingly. Rather than elaborating on that explanation, let's illustrate. Because the variable we are deriving, PREVSCORE, is numeric, we'll use the IFN function; IFC is for character results. The code which solves our problem using this new tool is:

```
data prevscores;
set demo;
by id;
PrevScore = ifn( first.id , . , lag(score) );
run;
```

Let's examine the IFN function call. The first argument is the condition to be evaluated. The second argument is the value to be returned if the first argument is TRUE. The third argument is the value to be returned if the first argument is FALSE.

This code is a bit more streamlined than the code we used earlier. There are no throwaway variables created for interim use and no conditional statements to overwrite incorrect results.

However, looking at the code, we might be concerned that it is afflicted with the fundamental problem which earlier gave incorrect results when we used the LAG function conditionally. Specifically, when the first argument is TRUE, so that the value from the second argument is returned, won't the third argument be bypassed? If the third argument is bypassed, won't the LAG function's queue contain the wrong value when the next observation is processed?

It turns out that these concerns are unwarranted. When the IFN and IFC functions are called, SAS evaluates **all** of the arguments, even those which end up unused. Thus, using these new functions to conditionally refer to the LAG function avoids the problem which arose when we used an IF statement for this purpose.

Let's trace the effects, as we did earlier. We add a column to show what IFN returns.

| Obs | ID | Round | Score | Prev Score | First.ID | Is LAG Called? | IFN Returns What? | LAG Returns What? | LAG Stores What? |
|---|---|---|---|---|---|---|---|---|---|
| 1 | A | 1 | 21 | . | 1 | Yes | . | . | 21 |
| 2 | A | 2 | 31 | .21 | 0 | Yes | 21 | 21 | 31 |
| 3 | A | 3 | 29 | 31 | 0 | Yes | 31 | 31 | 29 |
| 4 | B | 1 | 23 | . | 1 | Yes | . | 29 | 23 |
| 5 | B | 2 | 30 | 23 | 0 | Yes | 23 | 23 | 30 |
| 6 | B | 3 | 33 | 30 | 0 | Yes | 30 | 30 | 33 |

Notice that LAG is called every time, regardless of the value of FIRST.ID. As a consequence, LAG always stores the current value of SCORE. In turn, LAG always returns the immediate preceding value of SCORE (except of course during the processing of the first observation). The IFN function has the job of deciding whether to pass along the LAG results. The end result is that the correct values are generated:

| Obs | ID | Round | Score | PrevScore |
|---|---|---|---|---|
| **1** | A | 1 | 21 | . |
| **2** | A | 2 | 31 | 21 |
| **3** | A | 3 | 29 | 31 |
| **4** | B | 1 | 23 | . |
| **5** | B | 2 | 30 | 23 |
| **6** | B | 3 | 33 | 30 |

There's more to say about the IFC and IFN functions. We'll get to that later. First, we'll digress a bit about the interaction of the LAG function and the IF statement.

### MORE ABOUT LAG AND IF

Because of the problems which can arise when LAG is invoked conditionality, some SAS users conclude that LAG should never be used conditionally. However, there are situations where such usage produces correct results. It's really a matter of analyzing the circumstances.

To illustrate, let's sort our test data to put it in a different sequence.

```
proc sort data=demo out=sorted_by_round;
by round;
run;
```

The data set SORTED_BY_ROUND looks like this:

| Obs | ID | Round | Score |
|---|---|---|---|
| **1** | A | 1 | 21 |
| **2** | B | 1 | 23 |
| **3** | A | 2 | 31 |
| **4** | B | 2 | 30 |
| **5** | A | 3 | 29 |
| **6** | B | 3 | 33 |

We can get correct previous values for competitor A's SCORE by setting up a conditional assignment statement in this fashion:

```
data prevscores_A;
set sorted_by_round;
if id='A' then PrevScore = lag(score);
run;
```

This will produce correct results because it consistently exercises the LAG function for **each** of A's observations, and **only** for those observations. Here is the output:

| Obs | ID | Round | Score | PrevScore |
|---|---|---|---|---|
| **1** | A | 1 | 21 | . |
| **2** | B | 1 | 23 | . |
| **3** | A | 2 | 31 | 21 |
| **4** | B | 2 | 30 | . |
| **5** | A | 3 | 29 | 31 |
| **6** | B | 3 | 33 | . |

We can add a separate statement to pick up B's previous SCORE values. The code would then look like this:

```
data prevscores_AB;
set sorted_by_round;
if id='A' then PrevScore = lag(score);
if id='B' then PrevScore = lag(score);
run;
```

The result is:

| Obs | ID | Round | Score | PrevScore |
|---|---|---|---|---|
| **1** | A | 1 | 21 | . |
| **2** | B | 1 | 23 | . |
| **3** | A | 2 | 31 | 21 |
| **4** | B | 2 | 30 | 23 |
| **5** | A | 3 | 29 | 31 |
| **6** | B | 3 | 33 | 30 |

The code produces correct values because each separately coded reference to the LAG function has its own queue. That is true even if the separate references have the same arguments, as they do here.

Some users who become apprehensive about using the LAG function with IF statements lose sight of the difference between conditionally invoked LAG references and LAG references which appear **in** conditions. Consider this DATA step, which uses our example data in the original order. The purpose is to flag increases in SCORE.

```
data upscores;
set demo;
by id;
if score > lag(score) and not first.id then Up = 'Yes';
run;
```

The LAG function here is **not** conditional; it just happens to be used **in** a condition. It is evaluated every time, so the queue will always be ready to return the value from the immediately preceding observation. Consequently, the results will be correct:

| Obs | ID | Round | Score | Up |
|---|---|---|---|---|
| **1** | A | 1 | 21 | |
| **2** | A | 2 | 31 | Yes |
| **3** | A | 3 | 29 | |
| **4** | B | 1 | 23 | |
| **5** | B | 2 | 30 | Yes |
| **6** | B | 3 | 33 | Yes |

### MORE ABOUT IFC AND IFN

The IFC and IFN functions are capable of distinguishing missing values, giving rise to "three-branch" evaluations. To illustrate, we need a different example than the one we have been using. It can be pretty simple; we just run this code:

```
data nine_zero_dot;
do Trigger = 9 , 0 , . ; output; end;
run;
```

which produces this table:

| Obs | Trigger |
|---|---|
| 1 | 9 |
| 2 | 0 |
| 3 | . |

Keep in mind that SAS ordinarily considers zeroes and missing values to be FALSE, and positive and negative values to be TRUE. To see how two-branch and three-branch evaluations work, we run this code:

```
data captions;
set nine_zero_dot;
Two_Caption =
           ifc( trigger , 'Is True' , 'Is False');
Three_Caption =
           ifc( trigger , 'Is True' , 'Is False' , 'Unknown' );
run;
```

We've used IFC rather than IFN because we are specifying character values rather than numerics to be returned. Our output is:

| Obs | Trigger | Two_ Caption | Three_ Caption |
|---|---|---|---|
| 1 | 9 | Is True | Is True |
| 2 | 0 | Is False | Is False |
| 3 | . | Is False | Unknown |

The assignment statement for TWO_CAPTION uses only three arguments in the IFC call. In that respect it is like the IFN calls in our earlier example. In the first observation, the value 9 is a representation of TRUE, so the character value "Is True" is returned. Similarly, in the second observation, the 0 (zero) indicates a FALSE value, resulting in "Is False". The most interesting case arises in the third observation. In the IFC function call, there are results coded only for TRUE and FALSE; Trigger is missing, and missing values are considered to be FALSE, so the result is the same as in the second observation.

When we turn to the assignment statement for THREE_CAPTION, things are different when we get to the third observation. The IFC function has been coded with a fourth argument, one which allows differentiation between missing values and 0 (zero). Only a zero will be considered FALSE and cause the third argument to be returned. Missing values push the function along a distinct path using the optional fourth argument.

## CONCLUSIONS

Conditional use of LAG functions can be treacherous if the mechanics are not well understood and the application is not carefully analyzed. The new (SAS9) IFC and IFN functions provide a simpler and more natural way to correctly implement conditional LAG calls. IFC and IFN are also advantageous when it is necessary to have missing values follow a third path (neither TRUE nor FALSE).

## REFERENCES

SAS Institute Inc. 2006. *SAS® 9.1.3 Language Reference: Dictionary, Fifth Edition*. Cary, NC: SAS Institute Inc.

## ACKNOWLEDGEMENTS

SAS is a Registered Trademark of the SAS Institute, Inc. of Cary, North Carolina.

## CONTACT INFORMATION

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

Howard Schreier
Howles Informatics
Arlington VA

703-979-2720

hs AT howles DOT com
http://howles.com/saspapers/