ABSTRACT

A programmer may often overlook simple and useful functions and instead try to manipulate SAS® code in different ways to solve a problem. One such function we use at PerformRx is the LAG function. LAG is an analytical function that can be used to get the value of an attribute of the previous row. This paper will discuss the appropriate usage of the LAG function and includes sample code. Additionally, this paper briefly covers the LEAD, INTCK, and DIF functions as well as the RETAIN statement and the EXPAND procedure.

INTRODUCTION

SAS® version 9.2 has excellent features including over 450 built-in functions to make a programmer’s job easier. It is always important to have a basic understanding of some of these functions while writing SAS code. There are numerous occasions when a programmer is required to refer to the previous or the next data observation for analysis. The LAG and LEAD functions will then come in handy.

This paper illustrates the LAG function in more detail inclusive of using LAG in conjunction with other functions to compute values between data points.

MAIN LOGIC OF THE PROGRAM:

When the LAG function is compiled, SAS allocates memory in a queue to hold the value of the variable listed in the LAG function. The queue for LAG1 is initialized with one missing value, LAG2 is initialized with two missing values, and for LAGn SAS creates a queue of n + 1. Here 1 is the current value and n is for the lagged value. For each occurrence of the LAG function, the SAS program generates a queue of values. To illustrate, if we read observations 1, 2, 3, 4, and 5 below in Figure 1, the red colored arrows point to the lagged values for each observation. Figure 1 illustrates how SAS loops through observations line by line and writes the lagged values in the output data set.

Figure 1 illustrates how an observation in a queue flows through a DATA step
HERE IS A SIMPLE EXAMPLE OF HOW THE LAG FUNCTION WORKS:

Our project requires us to compare each drug prescribed for a member and dispensed from a pharmacy, against the previous drug dispensed, to determine the drug quantity prescribed. In Table 1 below we have included a data set containing the member information and drugs prescribed detail. To complete this simple project, we will use the LAG function. As explained above, when the LAG function is called for each observation, it retrieves the value of the last observation and places this value in the queue.

Let us see how this works in the following examples:

```sas
data Table1;
  input @ MemberID $4.
    DrugFill_Date mmddyy10.
    DrugName $8.
    Qty 8.;
  format DrugFill_Date mmddyy10.;
datalines;
1111 01/01/2010 ABILIFY 10
1111 01/15/2010 ABILIFY 20
1111 03/16/2010 ABILIFY 30
1111 03/30/2010 ABILIFY 40
2222 01/04/2010 GLEEVEC 10
2222 02/10/2010 GLEEVEC 20
2222 02/15/2010 GLEEVEC 30
2222 07/01/2010 GLEEVEC 40
; run;
```

**Table 1**: Pharmacy Informatics data showing the prescription details for each member with different quantities.

<table>
<thead>
<tr>
<th>Obs</th>
<th>Member ID</th>
<th>DrugFill_Date</th>
<th>Drug Name</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>ABILIFY</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>03/16/2010</td>
<td>ABILIFY</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>03/30/2010</td>
<td>ABILIFY</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>2222</td>
<td>01/04/2010</td>
<td>GLEEVEC</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>2222</td>
<td>02/10/2010</td>
<td>GLEEVEC</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>2222</td>
<td>02/15/2010</td>
<td>GLEEVEC</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>2222</td>
<td>07/01/2010</td>
<td>GLEEVEC</td>
<td>40</td>
</tr>
</tbody>
</table>

The DATA step below shows the LAG function code. The resulting Data set is shown in Example 1.

As we can see the LAG values are output in the “LAG_QTY” column. The first observation is null as expected. The LAG function worked as expected, but since the intent is to capture the last drug quantity for each distinct member, then observation 5 for member ID ‘2222’ is not correct, because the quantity of 40 relates to member ID ‘1111’.

```sas
Data Example1;
  Set Table1;
  LAG_QTY = LAG(Qty);
run;
```
In order to categorize LAG values by each member, we modify our code as outlined below using the DATA step code BY statement. Unfortunately this does not work and we incur the same resulting output as before.

```plaintext
data Example1A;
  set Table1;
  by MemberId;
  lag_qty = lag(qty);
run;
```

The next step is to use the conditional field FIRST.ID which will set the first observation of a new MEMBERID to missing, since we do not need the LAG value for those observations. Therefore, the following DATA step is used to create the desired results, but once again the output is incorrect. As we can see the value of LAG_QTY is missing in observation 2 and incorrect in observation 6.

When the LAG function is called conditionally for the first time it returns a missing value. On the subsequent calls, the LAG function returns the last value stored in the queue, and as a result returns two null values at the beginning of the data set and the rest of the values are not shifted correctly. The key concept to remember is the LAG function does not look at the previous observation; rather it looks at the previous value of the observation every time the LAG function is invoked.

```plaintext
data Example2;
  set Table1;
  by MemberId;
  if not first.MemberId then lag_qty = lag(qty);
run;
```

One way to rectify this problem when we use the LAG function conditionally is to set the FIRST.ID to missing.

```plaintext
data Example3;
  set Table1;
  by MemberId;
  LAG_QTY = lag(qty);
  if first. MemberId then lag_qty = .;
run;
```
An alternate approach as illustrated below is to set the LAG value equal to the previous LAG.

```sas
data Example3A;
    set Table1;
    by MemberId;
    lag_qty = lag(qty);
    if first. MemberId then Qty = lag_qty;
run;
```

The output data is correct and the results are as shown in the Example 3 data set.

<table>
<thead>
<tr>
<th>Obs</th>
<th>MemberID</th>
<th>DrugFill_</th>
<th>Drug</th>
<th>Qty</th>
<th>lag_qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>ABILIFY</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>02/16/2010</td>
<td>ABILIFY</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>03/30/2010</td>
<td>ABILIFY</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>2222</td>
<td>01/04/2010</td>
<td>GLEEVEC</td>
<td>10</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>2222</td>
<td>02/10/2010</td>
<td>GLEEVEC</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>2222</td>
<td>02/15/2010</td>
<td>GLEEVEC</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>2222</td>
<td>01/01/2010</td>
<td>GLEEVEC</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>

**PROGRAM DATA VECTOR:**

Let us examine the program data vector, which is the logical framework that SAS uses while creating data sets. The outcome of the below _NULL_ DATA step is displayed in Figure 2. Notice the use of the conditional statement BY MEMBERID creates two logical variables, FIRST.MEMBERID and LAST.MEMBERID. The FIRST.MEMBERID will have the value of 1 when SAS is processing an observation with the first occurrence and all the other observation values will be 0. When FIRST.ID is equal to 1 the LAG value is set to missing and in subsequent observations all the corresponding lagged values are shifted.

The LAST.MEMBERID is assigned the value of 1 for the observation with last occurrence and all the previous observation values are 0. The automatic variable _ERROR_ is 0 when the DATA step is executed with no errors and _N_ counts the number of iterations.

The below DATA _NULL_ step example explains how program data vector works when the LAG function is used.

```sas
data _null_
    set Table1;
    by MemberID;
    if not first. MemberID then lag_qty = lag(qty);
    put _all_;
run;
```

The LAG should set the value to missing for the FIRST.MEMBERID (where FIRST.MEMBERID = 1) and assign the correct lagged values on the subsequent observations. However, we notice that the lagged value for observation 2 is missing when it should be 10 and observation 6 is incorrect as shown (highlighted red cells) in Figure 2.

Notice that _ERROR_ = 0 indicating there is no error although we are not getting accurate results. Therefore, it is important to remember that when we conditionally use the LAG function, we can receive unexpected results without receiving any warning (_ERROR_ = 1).
<table>
<thead>
<tr>
<th>MEMBERID</th>
<th>DrugFill_date</th>
<th>DrugName</th>
<th>Qty</th>
<th>First.MEMBERID</th>
<th>Last.MEMBERID</th>
<th>LAG_QTY</th>
<th><em>Error</em></th>
<th><em>N</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>01/15/2010</td>
<td>ABILIFY</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1111</td>
<td>03/16/2010</td>
<td>ABILIFY</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1111</td>
<td>03/30/2010</td>
<td>ABILIFY</td>
<td>40</td>
<td>0</td>
<td>1</td>
<td>30</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2222</td>
<td>01/04/2010</td>
<td>GLEEVEC</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>2222</td>
<td>02/10/2010</td>
<td>GLEEVEC</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>40</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>2222</td>
<td>02/15/2010</td>
<td>GLEEVEC</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>2222</td>
<td>07/01/2010</td>
<td>GLEEVEC</td>
<td>40</td>
<td>0</td>
<td>1</td>
<td>30</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

The below DATA _NULL_ step output the correct results as shown in Figure 2.

```sql
data _null_;  
  set Table1;  
  by MemberId;  
    lag_qty = lag(qty);  
    if first.MemberId then lag_qty = .;  
  put _all_;  
run;
```

The following example displays how to generate multiple lagged values. Here LAG_QTY ask for the first value of the LAG. LAG_n would yield the_nth_ LAG.

```sql
data Example4;  
  set Table1;  
  by MemberId DrugFill_Date;  
    lag1 = lag1(qty);  
    lag2 = lag2(qty);  
    lag3 = lag3(qty);  
    lag4 = lag4(qty);  
    lag5 = lag5(qty);  
    lag6 = lag6(qty);  
    lag7 = lag7(qty);  
    lag8 = lag8(qty);  
run;
```
DOES LAG WORKS IN PROC SQL?

As we already by now know, the LAG function works in the DATA step. Can we imply the same function in PROC SQL? Let us look at the following SQL query which normally works in ORACLE. The following SQL code is added to examine the outcome results in PROC SQL and if we look at the SAS log output below, the error showing 'the LAG function is not supported in PROC SQL and it is only valid within the DATA step.'

```sql
proc sql;
  select DrugName, qty,
       lag(qty,1) as lag_qty
  from  Table1;
quit;
```

So far, we have seen the capability of the LAG function to look back and work with previous values, but the next question comes to mind, specifically is there any function that can look forward through the data set observations and work with that field?. The answer to this question is "yes" because the LEAD function is available in ORACLE.

When we try to use the LEAD function in a PROC SQL statement, we get the following result:

```sql
proc sql;
  select DrugName, qty,
       lead(qty,1) as lag_qty
  from  Table1;
quit;
```

Unlike the LAG function, the LEAD function does not work either in the DATA step or in the PROC SQL statement because the LEAD function would need to read forward to observation rows yet to be processed. SAS reads observations one at a time and then moves to the next observation. SAS only processes a record once the entire observation has been read, and there is no way to alter that process to look forward to the next observation. The reason the LEAD function works in Oracle is because Oracle processes the data and allocates memory differently than SAS.

However, we can achieve similar results as we would get using the LEAD function in Oracle by using some creative SAS code. In Example 5 below we merge the same data set twice and rename the QTY variable to LEAD_QTY to avoid over writing the value of the QTY field that exists in the original data set. Remember to first sort the data appropriately since the BY statement does not work. Notice that the value of LEAD_QTY in observation 1 is the same value of QTY in observation 2, basically the “LEAD” value that you would get if you used the LEAD function in Oracle.
**Example 5**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Member</th>
<th>Drug Fill Date</th>
<th>Drug Name</th>
<th>Qty</th>
<th>lead_qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>ABILIFY</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>03/16/2010</td>
<td>ABILIFY</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>03/30/2010</td>
<td>ABILIFY</td>
<td>40</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>2222</td>
<td>01/04/2010</td>
<td>GLEEVEC</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>2222</td>
<td>02/10/2010</td>
<td>GLEEVEC</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>2222</td>
<td>02/15/2010</td>
<td>GLEEVEC</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>2222</td>
<td>07/01/2010</td>
<td>GLEEVEC</td>
<td>40</td>
<td>.</td>
</tr>
</tbody>
</table>

### LAG WITH PROC EXPAND:

The PROC EXPAND procedure, which is only available with SAS/ETS (Econometric Time Series), has the LEAD function capability.

This procedure code and results are demonstrated below:

```sas
proc expand data = Table1 out = Example6 method=none;
by MemberId;
convert Qty = lead_qty / transformout= (lead);
convert Qty = lag_qty / transformout= (lag);
run;
```

### Example 6

<table>
<thead>
<tr>
<th>Obs</th>
<th>Member ID</th>
<th>Drug Fill Date</th>
<th>Drug Name</th>
<th>Qty</th>
<th>lag_qty</th>
<th>lag2_qty</th>
<th>lead_qty</th>
<th>lead2_qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>.</td>
<td>.</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>ABILIFY</td>
<td>20</td>
<td>10</td>
<td>.</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>03/16/2010</td>
<td>ABILIFY</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>40</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>03/30/2010</td>
<td>ABILIFY</td>
<td>40</td>
<td>30</td>
<td>20</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>2222</td>
<td>01/04/2010</td>
<td>GLEEVEC</td>
<td>10</td>
<td>.</td>
<td>20</td>
<td>30</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>2222</td>
<td>02/10/2010</td>
<td>GLEEVEC</td>
<td>20</td>
<td>10</td>
<td>.</td>
<td>30</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>2222</td>
<td>02/15/2010</td>
<td>GLEEVEC</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>40</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>2222</td>
<td>07/01/2010</td>
<td>GLEEVEC</td>
<td>40</td>
<td>30</td>
<td>20</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

**PROC EXPAND** is designed to obtain LAG and LEAD values by using the CONVERT statement. We can use the TRANSFORMOUT option to perform data transformation before and after the interpolating function is fit. Notice that this method can also take care of the missing values for each member. We can use as many CONVERT statements as we wish to create LAGn and LEADn from data.

### LAG WITH INTCK FUNCTION:

Suppose we have pharmacy data shown in Table 2 containing member prescriptions of various drugs with different drug fill dates. We want to output the members who are prescribed multiple drugs and we also want to flag the members who receive more than one drug with in 30 days period.

First we eliminate members who were prescribed only one prescription. Next we flag the members with dual prescriptions within 30 days.
Let us look at the following program and we will explain how it works. Remember to always sort the data by MemberID before manipulating the data in the next DATA step.

```
data Table2;
  input  @1 MemberID $4.
        @6 DrugFill_Date mmddyy10.
        @17 DrugName $7.
        @25 qty 2.;
  format DrugFill_Date mmddyy10.;
datalines;
1111 01/01/2010 ABILIFY 10
1111 01/15/2010 AMBIEN 20
1111 03/16/2010 ALORA  05
2222 01/01/2010 ASPRIN  01
3333 01/04/2010 TYLENOL 01
4444 02/10/2010 ABILIFY 10
4444 02/15/2010 AMOXIL  20
run;
```

**Table 2** Showing the details of single and multiple drugs for each member with different fill dates

<table>
<thead>
<tr>
<th>Obs</th>
<th>Member ID</th>
<th>DrugFill Date</th>
<th>Drug Name</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>AMBIEN</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>03/16/2010</td>
<td>ALORA</td>
<td>05</td>
</tr>
<tr>
<td>4</td>
<td>2222</td>
<td>01/01/2010</td>
<td>ASPRIN</td>
<td>01</td>
</tr>
<tr>
<td>5</td>
<td>3333</td>
<td>01/04/2010</td>
<td>TYLENOL</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>4444</td>
<td>02/10/2010</td>
<td>ABILIFY</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>4444</td>
<td>02/15/2010</td>
<td>AMOXIL</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>4444</td>
<td>07/01/2010</td>
<td>DOXIL</td>
<td>20</td>
</tr>
</tbody>
</table>

Let us see how our code works to process our data as required. First we want to eliminate the members with one prescription by flagging those observations where FIRST.MEMBERID and LAST.MEMEBRID is equal to 1. Output all other members that have more than one prescription in a separate data set. Here we take advantage of the INTCK function. As we know the INTCK function is a popular and powerful SAS function utilized to calculate the time interval in units of time (days, weeks, months, years, etc.) between two SAS DATE, TIME or DATETIME variables. Initially we apply the LAG function for MEMBERID and DRUGFILL_DATE after we sort them BY-group statement. Notice that we calculate the date difference for each MEMBERID and LAG_MEMBERID, the INTCK function will calculate the number of days between DRUGFILL_DATE and LAG_DRUGFILL_DATE and then we can flag the members with more than one drug within the 30 days time period.

```
data singlePrescriptions DualPrescriptions;
  set Table2;
  by MemberID DrugFill_Date;
  *----Output single prescriptions----*;
  if first.MemberID and last.MemberID then output singlePrescriptions;
  else
    do;
      lag_memberid = lag(MemberID);
      lag_drugFill_date = lag(DrugFill_Date);
      if lag_memberid = MemberID then
        do;
          numdays= intck('DAY',lag_drugFill_date, DrugFill_Date);
          if numdays < 30 then star = '**';
        end;
  *----Output dual prescriptions with '*' flag----*;
```
output DualPrescriptions;
end;
run;

**Dual Prescriptions with in 30 Days Supply**

<table>
<thead>
<tr>
<th>Obs</th>
<th>MemberID</th>
<th>DrugFill_Date</th>
<th>DrugName</th>
<th>qty</th>
<th>lag_drug_memberid</th>
<th>lag_drug_date</th>
<th>numdays</th>
<th>star</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>1111</td>
<td>18263</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>AMBIEN</td>
<td>20</td>
<td>1111</td>
<td>18277</td>
<td>14</td>
<td>*</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>02/16/2010</td>
<td>ALORA</td>
<td>5</td>
<td>1111</td>
<td>18337</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4444</td>
<td>02/16/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>4444</td>
<td>18383</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4444</td>
<td>02/15/2010</td>
<td>AMOXIL</td>
<td>20</td>
<td>4444</td>
<td>18393</td>
<td>5</td>
<td>*</td>
</tr>
<tr>
<td>6</td>
<td>4444</td>
<td>07/01/2010</td>
<td>DOXIL</td>
<td>20</td>
<td>4444</td>
<td>18300</td>
<td>136</td>
<td></td>
</tr>
</tbody>
</table>

**LAG WITH DIF FUNCTION:**

The DIF function works the same way as the LAG function, the only difference being LAG simply assigns previous values, but DIF assigns the difference between the current value and the previous value of a variable. The following data set contains member information and drug details with different fill dates. We want to increase the drug price if a member uses the same drug for more than twelve months. Here we used the DIF function to calculate the number of days and the price difference every time the member refills the same medication. The LAG function is used to compute the increased price difference.

```r
data Table3;
input @1 MemberID $4.
    @6 DrugFill_Date mmddyy10.
    @17 DrugName $7.
    @24 price 8.
    format DrugFill_Date mmddyy10. price 8.;
datalines;
1111 01/01/2007 ABILIFY 30
1111 01/15/2008 ABILIFY 60
1111 10/16/2009 ABILIFY 80
1111 07/01/2010 ABILIFY 90
2222 01/04/2007 GLEEVEC 30
2222 02/10/2008 GLEEVEC 40
2222 12/15/2009 GLEEVEC 60
2222 07/01/2010 GLEEVEC 90
;
run;
```

**Table 3** Showing the member information with price details for each drug with different drug fill dates.
data Example7;
  set Table3;
  by MemberID DrugFill_Date;
    diff_drugFill_date = dif(DrugFill_Date);
    lag_price = lag(price);
    dif_price = dif(price);
    if first.MemberID then do;
      diff_drugFill_date = .;
      lag_price = .;
      dif_price = .;
    end;
    if diff_drugFill_date > 365 then do;
      percent_increase = round((dif_price/lag_price)*100);
    end;
run;

Notice that the LAG_PRICE and DIF_PRICE are set to missing for each member before calculating price increase values.

<table>
<thead>
<tr>
<th>Obs</th>
<th>MemberID</th>
<th>DrugFill_Date</th>
<th>DrugName</th>
<th>price</th>
<th>diff_drugFill_date</th>
<th>lag_price</th>
<th>dif_price</th>
<th>price_increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2007</td>
<td>ABILIFY</td>
<td>30</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2008</td>
<td>ABILIFY</td>
<td>60</td>
<td>37.5</td>
<td>30</td>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>10/15/2009</td>
<td>ABILIFY</td>
<td>80</td>
<td>64.0</td>
<td>60</td>
<td>20</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>07/01/2010</td>
<td>ABILIFY</td>
<td>90</td>
<td>250</td>
<td>80</td>
<td>10</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>2222</td>
<td>01/04/2007</td>
<td>GLEEVEC</td>
<td>30</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2222</td>
<td>02/10/2008</td>
<td>GLEEVEC</td>
<td>40</td>
<td>402</td>
<td>30</td>
<td>20</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>2222</td>
<td>12/15/2009</td>
<td>GLEEVEC</td>
<td>60</td>
<td>674</td>
<td>40</td>
<td>20</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>2222</td>
<td>07/01/2010</td>
<td>GLEEVEC</td>
<td>90</td>
<td>190</td>
<td>60</td>
<td>30</td>
<td>.</td>
</tr>
</tbody>
</table>

LAG WITH RETAIN STATEMENT:

RETAIN is one of the most useful tools for capturing information from the previous observation. We can also use the RETAIN statement to compute across observations. There are situations where every programmer needs to fill empty observations with corresponding values as shown in Table 4. Suppose we want to calculate the average PRICE value with reference to drug quantity every time member purchase drugs from the pharmacy.

data Table4;
  input @1 MemberID $4.
    @6 DrugFill_Date mmddyy10.
    @17 DrugName $7.
    @25 qty 2.
    @28 price dollar6.2 ;
  format DrugFill_Date mmddyy10. price Dollar6.2 ;
datalines;
1111 01/01/2010 ABILIFY 10 $6.25
1111 01/15/2010 ABILIFY 20
1111 03/16/2010 ABILIFY 30
1111 01/01/2010 ABILIFY 40
2222 01/04/2010 GLEEVEC 10 $5.55
2222 02/10/2010 GLEEVEC 20
2222 02/15/2010 GLEEVEC 30
2222 07/01/2010 GLEEVEC 40 ;
run;
Table 4 Showing the member information of price details for each drug with different quantities.

<table>
<thead>
<tr>
<th>Obs</th>
<th>MemberID</th>
<th>DrugFill_Date</th>
<th>Drug_Name</th>
<th>qty</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>$6.25</td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>ABILIFY</td>
<td>20</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>03/15/2010</td>
<td>ABILIFY</td>
<td>30</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>40</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>2222</td>
<td>01/04/2010</td>
<td>GLEEVEC</td>
<td>10</td>
<td>$5.55</td>
</tr>
<tr>
<td>6</td>
<td>2222</td>
<td>02/10/2010</td>
<td>GLEEVEC</td>
<td>20</td>
<td>.</td>
</tr>
<tr>
<td>7</td>
<td>2222</td>
<td>02/15/2010</td>
<td>GLEEVEC</td>
<td>30</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>2222</td>
<td>07/01/2010</td>
<td>GLEEVEC</td>
<td>40</td>
<td>.</td>
</tr>
</tbody>
</table>

We can use the above code to fill the empty price values for each member by using the RETAIN statement to hold the previous value of a variable. As you can see in Example 8 output, the PRICE variable is filled with corresponding values. We can then use the LAG function to calculate the average PRICE.

```
data Example8(rename=(Old_price=price));
  set Table4;
  by MemberID ;
  if first.MemberID then old_price=price;
  retain old_price;
  if price EQ . then old_price=old_price;
  else old_price=price;
  lag_qty = lag(qty);
  if first.MemberID then lag_qty = .;
  else avgprice = mean(old_price*lag_qty);
run;
```

We can use the above code to fill the empty price values for each member by using the RETAIN statement to hold the previous value of a variable. As you can see in Example 8 output, the PRICE variable is filled with corresponding values. We can then use the LAG function to calculate the average PRICE.

<table>
<thead>
<tr>
<th>Obs</th>
<th>MemberID</th>
<th>DrugFill_Date</th>
<th>Drug_Name</th>
<th>qty</th>
<th>price</th>
<th>old_price</th>
<th>lag_qty</th>
<th>avgprice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>10</td>
<td>$6.25</td>
<td>6.25</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1111</td>
<td>01/15/2010</td>
<td>ABILIFY</td>
<td>20</td>
<td>.</td>
<td>.</td>
<td>6.25</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>1111</td>
<td>03/15/2010</td>
<td>ABILIFY</td>
<td>30</td>
<td>.</td>
<td>.</td>
<td>6.25</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>01/01/2010</td>
<td>ABILIFY</td>
<td>40</td>
<td>.</td>
<td>.</td>
<td>6.25</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>2222</td>
<td>01/04/2010</td>
<td>GLEEVEC</td>
<td>10</td>
<td>$5.55</td>
<td>5.55</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>2222</td>
<td>02/10/2010</td>
<td>GLEEVEC</td>
<td>20</td>
<td>.</td>
<td>.</td>
<td>5.55</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>2222</td>
<td>02/15/2010</td>
<td>GLEEVEC</td>
<td>30</td>
<td>.</td>
<td>.</td>
<td>5.55</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>2222</td>
<td>07/01/2010</td>
<td>GLEEVEC</td>
<td>40</td>
<td>.</td>
<td>.</td>
<td>5.55</td>
<td>30</td>
</tr>
</tbody>
</table>

CONCLUSION

We have illustrated the basics of how to utilize the LAG function appropriately. There are several other SAS papers devoted to explaining the pitfalls of the LAG function when it is used conditionally. In order to avoid unexpected results, a better understanding of the concepts are required to ensure we meet our objective.

The illustrated code examples we have provided should be useful to programmers when they look for an alternate approach with other related functions. We hope this paper may be useful for the base SAS user who is lagging behind the LAG function.
REFERENCES


Schreier, Howard (2007). Conditional Lags Don’t Have to be Treacherous NESUG, Coders’ Corner


ACKNOWLEDGMENTS

We would like to acknowledge Mr. Shimels Afework, MBA, the Senior Director of PerformRx. PerformRx, LLC was formed in 1999 as a division of the AmeriHealth Mercy Family of Companies. We provide specialized pharmacy benefit management services (PBM) through proactively managing escalating pharmacy costs while focusing on clinical and financial results.

We also acknowledge the SAS Research and Development team for their Technical support

CONTACT INFORMATION:

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

Name     Anjan Matlapudi
          Senior SAS Programmer, Pharmacy Informatics and Finance
Address   PerformRx, The Next Generation PBM
          200 Stevens Drive
          Philadelphia, PA 9113
Work Phone:         (215)937-7252
Fax:    (215)863-5100
E-mail:                   anjan.matlapudi@performrx.com
                          anjanmat@gmail.com

Name     J. Daniel Knapp, MBA
          Senior Manager, Pharmacy Informatics and Finance
Address   PerformRx, The Next Generation PBM
          200 Stevens Drive
          Philadelphia, PA 19113
Work Phone:          (215-937-7251
Fax:     (215)863-5100
E-mail:                    Daniel.Knapp@performrx.com

SAS® is a registered trademark or trademark of SAS® Institute, Inc. in the USA and other countries. Oracle is a registered trademark of the Oracle Corporation. ® indicates USA registration.