

Advanced DATA Step Look-Back and Look-Ahead Techniques

Joshua M. Horstman, Nested Loop Consulting

ABSTRACT

The DATA step is a powerful and versatile tool for manipulating SAS data sets. Because it is built around the concept of reading one record at a time into the Program Data Vector, DATA step programming logic generally has access to only one record at a time. However, there are many situations in which it is advantageous to be able to look back at values from prior records or to look ahead at values from later records. This presentation introduces several advanced DATA step programming techniques that allow the programmer to work with values from multiple records concurrently. Approaches discussed include the double SET statement, the self-merge, the LAG function, and more. Examples are used to explain each method and demonstrate its use.

INTRODUCTION

The SAS DATA step is an implied loop. The logic encoded into a DATA step executes iteratively, once for each input record. In a typical DATA step, values are loaded into the program data vector (PDV), processing occurs, and then sometimes one or more output records are created in an output dataset. This framework is suitable for a tremendous variety of data processing tasks. However, there are often tasks that require processing values from multiple records concurrently.

Fortunately, there are tools available within the DATA step to facilitate these tasks. This paper discussed several DATA step techniques that allow the SAS programmer to look ahead to a later record or to look back to an earlier record. These methods are presented through a series of examples. Additional discussion of these methods can be found in section 3.1 of Carpenter (2012).

LOOKING BACK WITH THE LAG FUNCTION

One of the easiest ways to utilize a value from a record that was processed prior to the record currently in the program data vector is to use the LAG family of functions. This family consists of 99 sequentially numbered functions LAG1, LAG2, LAG3, ... LAG99.

These functions are used to store values in a queue and then later return those values that were previously stored. The numerical suffix on the name of the function specifies the length of the queue that will be created. In other words, LAG n creates a queue that can store n values. The LAG1 function can also be invoked using the alias LAG.

The LAG functions accept a single argument. When the function is called, the value of that argument is placed on the queue and the value in the last position of the queue is returned by the function. The queue is initialized with missing values, so the first n calls to LAG n will return a missing value.

Each instance of a LAG function within the code for a DATA step will have its own separate queue. By using the LAG functions to pass the values of variables from the program data vector into a queue, one can effectively save these values for use when processing a subsequent observation.

The example below demonstrates this technique. This DATA step uses the LAG function to compute the number of days since the previous visit for subjects in a research study.

```

data labvisits(keep=subjid visit lagvisit interval lagdate lbdtd);
  set labdates;
  by subjid; ❶

  lagvisit= lag(visit); ❷
  lagdate = lag(lbdtd); ❷

  if not first.subjid then do; ❸
    interval = lbdtd - lagdate; ❹
    if not missing(interval) then output;
  end;
  format lagdate mmddyy10.;
run;

```

Because a BY statement ❶ is used, the dataset must be sorted by subject (SUBJID) prior to this DATA step. Further, the data must be sorted by visit number (VISIT) within subject for our computation to proceed correctly.

The LAG function ❷ is used to place the current date (LBDT) into a queue of length 1 and visit number (VISIT) into a separate queue of length 1. During the first iteration of the DATA step, both calls to the LAG function return missing values. This is acceptable because it would be illogical to attempt to determine the number of days since the previous visit for the first visit in the dataset.

In fact, it does not make sense to perform this computation at the first visit for each subject, since the prior record in the dataset is for a different subject. To ensure this does not happen, the derivation ❹ is placed within a block of conditional logic ❸. However, it is important that the LAG function be placed outside the conditional logic. It must be called for each record to place the desired values in their respective queues.

LOOKING AHEAD USING A MERGE

Ordinarily, a DATA step merge is used to combine two or more different datasets. However, by merging a dataset with itself, it is possible to bring values from different records together into the program data vector. The key to this technique is having the MERGE statement reading from different points within the same dataset. In the following DATA step, a MERGE statement is used to determine the number of days until the next visit for subjects in a research study.

```

options mergenoby=nowarn; ❶

data nextvisit(keep=subjid visit lbdtd days2nextvisit);
  merge labdates(keep = subjid visit lbdtd)
        labdates(firstobs = 2 ❷
                  keep = subjid lbdtd
                  rename = (subjid=nextsubj lbdtd=nextdt)); ❸
  Days2NextVisit = ifn(subjid=nextsubj, nextdt-lbdtd, ., .); ❹
run;

```

The MERGE is performed without a BY statement because we do not wish to match up observations based on the value of any variable. Rather, this is a one-to-one merge in which records are matched up based on their position within the dataset. This means the dataset needs to be sorted by subject and visit prior to the execution of this DATA step. To suppress any log warnings about the missing BY statement, the MERGENOBY system option is set to NOWARN ❶.

The input dataset is included twice on the MERGE statement. However, the use of the FIRSTOBS= data set option ❷ on the second reference to the input dataset offsets the read operation by one record. The first time this MERGE statement executes, it will read both the first and second records, with the incoming variables from the second record renamed ❸ to avoid overwriting the values read from the first record.

The MERGE statement maintains separate pointers into each input dataset, even if they happen to be the same dataset. On the second iteration of this DATA step, the MERGE statement will read the second record from the first input dataset and the third record from the second input dataset.

Thus, for each record, the PDV contains both the date of the current visit and that of the subsequent visit, if any. The number of days to the next visit is then calculated using simple subtraction after first confirming that both values pertain to the same subject ④.

LOOKING AHEAD USING A DOUBLE SET STATEMENT

Another approach for looking ahead to subsequent records utilizes two SET statements. This technique is similar to the MERGE method described above but offers the programmer greater control over the read process. Once again, the input dataset must be sorted by subject and visit before this DATA step begins execution.

```
data nextvisit(keep=subjid visit lbdtd days2nextvisit);  
  set labdates(keep=subjid visit lbdtd) end=lastlab; ①  
  if not lastlab then do; ②  
    set labdates(firstobs = 2 ③  
                  keep      = subjid lbdtd  
                  rename    = (subjid=nextsubj lbdtd=nextdtd));  
    Days2NextVisit = ifn(subjid=nextsubj,nextdtd-lbdtd,.,.); ④  
  end;  
run;
```

The first SET statement ① is used to read the “current” observation, i.e. the record for which the new variable (number of days to next visit) is being derived. The END= option on the SET statement creates a temporary variable called LASTLAB. This variable will be 0 until the associated SET statement has read the last observation from the LABDATES dataset. At that point, it will be assigned a value of 1.

Until we reach that last observation, the conditional logic ② will allow the second SET statement to execute. Due to the FIRSTOBS= dataset option ③, the second SET statement begins reading with the second observation in the dataset. Despite sharing the same input dataset, each SET statement maintains its own pointer into that dataset. Incoming variables read by the second SET statement are renamed so they do not collide with the same variables being read using the first SET statement.

The assignment statement which derives the number of days to the next visit is also placed within the same block of conditional logic. Thus, it will not be executed on the last iteration of the DATA step when there are no further visits. Additional logic ④ ensures this derivation is not performed for the last visit for each subject.

LOOKING BACK USING A DOUBLE SET STATEMENT

Earlier we used the LAG family of functions to retrieve values from an earlier observation. This approach is quite handy for looking back a specific number of observations. However, when the number of observations to look back is unknown, it can often be more convenient to use a double SET statement.

In the prior example, the second SET statement was always reading one record ahead of the primary SET statement. In this example, the second SET statement will be behind the primary SET statement, allowing us to look back to prior records.

The goal of this example is to find all lab visits that fall between the first and second potassium reading that is greater than or equal to 4.2. In a single pass of the data, we cannot identify the second peak (if it exists) until the records of interest have already been processed. We could employ separate DATA steps to perform multiple passes of the data. However, the use of two SET statements allows for multiple passes within a single DATA step.

```

data BetweenPeaks(keep=subjid visit lbd t potassium);
  set labdates(keep=subjid lbd t potassium); ❶
  by subjid lbd t; ❷
  retain firstloc .
      found ' '; ❸
  obscnt+1; ❹
  if first.subjid then do; ❺
    found=' ';
    firstloc=.;
  end;
  if found=' ' and potassium ge 4.2 then do; ❻
    if firstloc=. then firstloc=obsnt; ❼
    else do;
      found='x'; ❽
      do point=firstloc to obsnt; ❾
        set labdates(keep= subjid visit lbd t potassium)
          point=point; ❿
        output betweenpeaks;
      end;
    end;
  end;
run;

```

The primary SET statement ❶ reads one record at a time from the beginning of the dataset as usual. Because we are using a BY statement ❷, the dataset must be already sorted by the BY variables. The RETAIN statement ❸ is used to initialize some bookkeeping variables that will retain their values across iterations of the DATA step. In addition, a counter variable called OBSCNT is established ❹ to track the current record position as we move through the dataset.

Whenever the current record being processed represents the first record for a subject, we re-initialize our bookkeeping variables ❺. The variable FIRSTLOC will record the location of the first potassium peak for the current subject while the FOUND flag tracks whether we have encountered the second peak.

If a potassium value meeting the threshold criteria is encountered and we've not already found the second peak ❻, then we have found either the first peak or the second peak. If it is the first peak, we store the location in FIRSTLOC.

If it is not the first peak, then the second peak has been encountered and the appropriate flag is set ❽. Now that the locations of both the first and second peaks are known, the second SET statement is used to go back and read each of those records and write them to the output dataset. This is done by using a DO loop ❾ in conjunction with the POINT= option ❿ to direct the SET statement to read only the specific records between the first and second potassium peaks.

LOOKING BACK USING A FIFO STACK

Another way to look back to values from prior records involves using a first-in-first-out (FIFO) stack. A stack is simply a collection of values that rotate through according to specific entrance and exit rules. This approach can be especially helpful for computing certain types of statistics such as moving averages which require processing across a series of values.

The following example calculates a three-day moving average of potassium levels for each subject. The stack is implemented within the context of the DATA step by using an array.

```

data Average(keep=subjid visit lbdn potassium Avg3day);
  set labdates;
  by subjid;

  retain visitcnt .; ❷
  array stack {0:2} _temporary_; ❶
  if first.subjid then do;
    call missing(of stack{*}); ❸
    visitcnt=0;
  end;
  visitcnt+1;
  index = mod(visitcnt,3); ❹
  stack{index} = potassium;
  avg3day = mean(of stack{*});
run;

```

Calculating a three-day moving average requires always having access to the current value and the two prior values. These are stored in an array of three elements ❶. We also establish a counter variable ❷ to track how many visits have been read for the current subject. When we begin reading records for a subject, the stack is initialized with missing values and the visit counter is reset ❸.

We calculate an array index where the current potassium value should be stored by using the MOD function to cycle through the elements of the array ❹. This ensures that the current value always replaces the oldest value (first-in-first-out). Computing the moving average is a simple matter of averaging the array elements ❺. Note that the MEAN function ignores missing values. The moving average will still be computed for the first two records per subject but will be based on fewer than 3 values.

CONCLUSION

Learning how to “look ahead” and “look back” in the DATA step allows the SAS programmer to tackle virtually any data processing task that might be needed. To gain a greater level of control over the operation of the DATA step, the programmer must thoroughly master important DATA step statements such as SET and MERGE, their associated options, and other DATA step tools such as the LAG functions and arrays.

REFERENCES

Carpenter, Art. 2012. *Carpenter’s Guide to Innovative SAS® Techniques*. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

The examples in this paper are adapted with permission from *Carpenter’s Guide to Innovate SAS® Techniques* by Art Carpenter. A statistical programmer, consultant, trainer, and author for decades, Art became known and respected throughout the community of SAS users. He produced several books, dozens of training courses, and numerous conference papers. While Art is now enjoying a well-earned retirement, he has graciously allowed me to continue bringing his training materials to new audiences.

CONTACT INFORMATION

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

Joshua M. Horstman
 Nested Loop Consulting LLC
josh@nestedloopconsulting.com