Paper 9540-2016

How to Create Data-Driven Lists

Kate Burnett-Isaacs, Statistics Canada

ABSTRACT

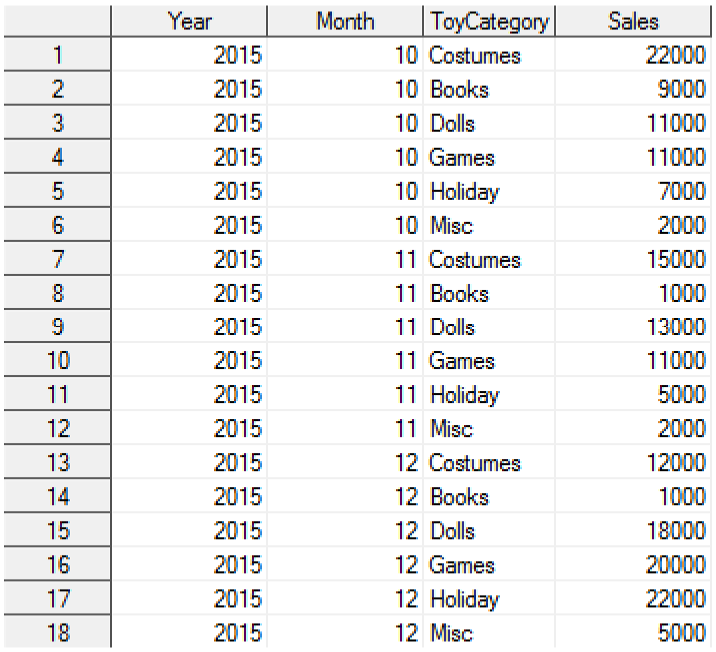
As SAS® programmers we often want our code or program logic to be driven by the data at hand, rather than be directed by us. Such dynamic code enables the data to drive the logic or sequence of execution. This type of programming can be very useful when creating lists of observations, variables, or data sets from ever-changing data. Whether these lists are used to verify the data at hand or are to be used in later steps of the program, dynamic code can write our lists once and ensure that the values change in tandem with our data. This Quick Tip paper will present the concepts of creating data-driven lists for observations, variables, and data sets, the code needed to execute these tasks, and examples to better explain the process and results of the programs we will create.

INTRODUCTION

Generating data-driven lists of observations, variables or datasets are useful techniques to have in our SAS dynamic programming toolbox. We need a way to effectively create lists and subsequently use them when our data is ever-changing. We can make these actions more efficient with the use of macro programming. This type of programming enables our code and processing to be flexible with varying input data. This paper will demonstrate how to create data-driven lists of observations, variables, and datasets and touch upon useful accompanying tools like counting the items in our lists and the %PUT statement.

Macro programming is critical to creating data-driven processes. Macros allow for code to be generalized and execution logic to vary with the data provided. This Quick Tip assumes the reader has a basic understanding of macro programming and language. The macro concepts used in this paper include: the timing of macro references and execution in relation to DATA steps or procedures; defining macros and macro variables; calling and executing macro code; and some basic conditional macro programming. A useful resource for more information on macro programming is *Carpenter's Complete Guide to the SAS Macro Language* (2004).

To help understand the processes discussed in this paper and to verify the results of our code, we will be using a dataset on toy sales, presented in Figure 1: Toy Sales for Quarter 4, 2015. This dataset contains year and month of sale for Quarter 4, 2015, the category of toys sold (*ToyCategory*) and the sales revenue for each category during each month:



	Year	Month	ToyCategory	Sales
1	2015	10	Costumes	22000
2	2015	10	Books	9000
3	2015	10	Dolls	11000
4	2015	10	Games	11000
5	2015	10	Holiday	7000
6	2015	10	Misc	2000
7	2015	11	Costumes	15000
8	2015	11	Books	1000
9	2015	11	Dolls	13000
10	2015	11	Games	11000
11	2015	11	Holiday	5000
12	2015	11	Misc	2000
13	2015	12	Costumes	12000
14	2015	12	Books	1000
15	2015	12	Dolls	18000
16	2015	12	Games	20000
17	2015	12	Holiday	22000
18	2015	12	Misc	5000

Figure 1: Toy Sales for Quarter 4, 2015

DATA-DRIVEN LIST OF OBSERVATIONS

It is important to understand what observations we are dealing with in our programming, particularly if the values and number of those observations are data-dependent. To create a list of observations, we need to pull the values of the observations from the dataset in question and put them in a macro variable. We will use the PROC SQL procedure, specifically a SELECT statement, to do this. Using the data from ToySales found in Figure 1: Toy Sales for Quarter 4, 2015 as an example; let's put the distinct categories of toys found in this dataset all into one list. The code required for this action is as follows:

```
%macro listobs;
  proc sql noprint;
   select distinct ToyCategory
         into :listofcategories separated by ', '
               from ToySales;
  quit;

  %put My list of observations is: &listofcategories;
  %mend;
%listobs;
```

There are three key components to this code that allow us to generate our lists of observations. The first is the DISTINCT argument, which is used to select only the unique values found in the variable ToyCategory. If we wanted all observations, even duplicates, in our list, we would not use this option.

Secondly, we use the INTO clause to assign the value produced in the SELECT statement to the macro variable *listofcatogorie*s. In this case, each value in the list is separated by a comma and a space. The value that is found within the single quotation marks in the SEPARATED BY argument is the literal value that will separate the list items. Therefore pay attention to any characters or spacing you want to display between your observations and make sure the exact match is in the single quotation marks after the SEPARATED BY argument.

A helpful tool to verify that the above code has generated our desired list of observations, is the %PUT statement. This statement displays the resolved values of the macro variable, *listofcategories,* to the user log. This is a useful tip to make sure that our macro variable has been defined correctly and resolves to our desired value. Our macro variable *listofcategories* resolves to:

```
My list of observations is: Books, Costumes, Dolls, Games, Holiday, Misc
```

We now have made a list of unique toy category observations that can change based on the data at hand.

A useful tool to accompany the creation of a list of observations is determining the number of observations in that list and putting that value into a macro variable. We can then use this macro variable as our end value in %DO loop to process all the list items. The procedure is similar to creating the list in the first place in that we will use PROC SQL and the SELECT statement. This time around, we want to count the number of observations we are putting into our list, so not surprisingly, we use the COUNT function in our SELECT statement. We can either count the number of variables altogether in our data set or we can count the number of unique observations by using the DISTINCT argument. We will then put the number of observations in our dataset INTO a macro variable, like so:

```
%macro numobs;
      proc SQL noprint;
                  SELECT count(ToyCategory) INTO :numbers
                        FROM WORK.ToySales;
      quit;
      %put The number of observations in our list is &numbers.;

      proc SQL noprint;
                  SELECT count(distinct(ToyCategory)) INTO :dist_numbers
                        FROM WORK.ToySales;
      quit;
      %put The number of unique observations in our list is &dist_numbers.;
%mend;
%numobs;
```

With the use of the %PUT statement we can examine what the macro variables *numbers* and *unique_numbers* resolve to:

```
The number of observations in our list is 18
The number of unique observations in our list is 6
```

We now have end values to use in a %DO loop to process our list items.

DATA-DRIVEN LIST OF VARIABLES

A dynamic list of variables is easy to establish and can significantly improve the efficiency of our coding. Like in the case of creating a list of observations, we will use PROC SQL and the SELECT statement to pull the list of variables in our dataset and put them into a macro variable. The main difference here is that instead of pulling these variables directly from our dataset, we pull them from the SAS dictionary table called *Columns*. SAS dictionary tables are read only tables that contain information about the current SAS session (Eberhardt and Brill, 2006). There are many dictionary tables that contain information on topics ranging from macros to indexes to session options. As you have already guessed, the dictionary table called *Columns* contains metadata about all variables (or columns) available in every dataset in every library within our current SAS session. *Dictionary* is an automatically generated SAS library, and we access it the same way as any other library in our SELECT statement, like:

```
  proc sql;
      Select *
      From dictionary.Columns
  quit;
```

The variable or column in the dictionary table *Columns* that contains the variable names found in our datasets is called ‘Name’. So, this is the variable we want to pick in our SELECT statement. Let’s make a list of the variables found in ToySales and put them it a macro variable called *listvars*:

```
%macro listvar;
    proc sql noprint;
           select Name into :listvars separated by ' '
                  from dictionary.Columns
                         where (libname = 'WORK') AND (memname = 'TOYSALES');
    quit;
%put My dataset has &numvars variables and they are: &listvars.;
%mend;
%listvar;
```

It is very important to use a WHERE clause in this process. If we did not include the WHERE clause, then we would get all the variables that exist in the active SAS session (and it’s a lot!). In order to pull the variables used only in our dataset called ToySales, we need to identify the library (*libname*) our dataset is in and the dictionary table member (which is our dataset in question) name (*memname*). In our case, our library name is WORK and our member name is TOYSALES:

```
where (libname = 'WORK') AND (memname = 'TOYSALES');
```

The values found in the variables *libname* and *memname* are in capitals, so it is important that these values are capitalized by the programmer or by using the UPCASE function, such as:

```
where (libname = upcase('work')) AND (memname = upcase('ToySales'));
```

The UPCASE function provides more flexibility in the code because the dataset names can also be data-driven by using a macro variable.

Thanks to the %PUT statement, we can see that our list of variables is as follows:

```
My dataset has the following variables: Year Month ToyCategory Sales
```

Note that I have not separated these variables in my list by a comma. This allows me to use my list of variables from the ToySales dataset in subsequent DATA step statements such as FORMAT, LENGTH and KEEP statements.

To enhance our repertoire of list building, we can put the number of variables in our list into a macro variable. Like in the observations case, we want to use the COUNT function in our SELECT statement to count the number of different names we have in our dictionary table *Columns*. Since we can’t have duplicate variable names in our dataset, there is no need to use the DISTINCT argument. The code to generate a macro variable of the number of variables we have in our ToySales dataset is:

```
%macro numvar;
    proc sql noprint;
                  select count(name) into : numvars
                         from dictionary.Columns
                         WHERE (libname = 'WORK') AND (memname = 'TOYSALES');
    quit;
%mend;
%numvar;
```

The macro variable *numvars* resolves to the number 4, which we can then use in a %DO loop to process all of our variables in our list!

DATA-DRIVEN LIST OF DATASETS

Dictionary tables also contain information about the datasets available in the current session of SAS. Therefore, if we wanted to put all the datasets of a certain library into a list we would again use PROC

SQL and the SELECT statement and pull our dataset names (*memname*) from the dictionary table called *Members* and put it into a macro variable called *listtable*:

```
%macro tablenames;
    proc sql noprint;
          select memname into :listtable separated by ' '
                from dictionary.Members
                      where (libname = 'WORK') ;
    quit;
    %put My dataset has the following tables: &listtable.;
%mend;
%tablenames;
```

The same concepts of creating lists of observations or variables applies here, including the SEPARATED BY argument and WHERE clause. The macro variable *listtable* returns the following list:

```
My dataset has the following tables: SASMACR TOYSALES
```

The downside of using this code is that it returns ALL datasets in the WORK library, including the SAS macro catalog (which contains the compiled macros from the current SAS session).

If we wanted to find out how many datasets we have in our WORK library, we can use the COUNT function in our SELECT statement to determine the number of *memname* in the dictionary table *Members*:

```
%macro tablenumbers;
    proc sql noprint;
          select count(memname) into : numdataset
                from dictionary.Members
    quit;
%mend;
%tablenumbers;
```

But again, this is going to count ALL datasets in the WORK library.

So what if we want only certain datasets in our list? If we need a data-driven list of datasets then we need to create datasets dynamically. One way to do this is to use macro %DO loops, where a new dataset is created in each iteration of the loop. Consider the following example where we create a separate dataset for each month in our ToySales dataset:

```
%do i=10 %to 12;
    data Sales_month&i.;
          set  ToySales;
                where month=&i.;
                drop month;
                rename Sales=Sales_&i.;
run;
```

[Note: I have hard-coded the months in question to maintain focus on the task at hand, but these too can be data-driven.]

In this %DO loop, we have created three datasets and we want to put them in a list without hard-coding the numerical suffixes 10, 11 or 12 (because these values could change depending on the time period found in our dataset). To accomplish this we must work with the %DO loop to add the dataset we just made to a list of previously made datasets.

The most difficult aspect of this task is to how to start. We want to create a list and use it again, which makes a macro variable the logical place to start. We cannot just set a macro variable to be the dataset created in each iteration, because it will get overwritten with each pass. So, we need a way to keep the first dataset we make (Sales_10) while we add subsequent datasets.

It is important to note that macro code is resolved prior to the execution of the DATA step. With this in mind, we need to define a null macro variable called *datasetlist* prior to the %DO loop. Then, within the %DO loop, we redefine the macro variable *datasetlist* to add on subsequent datasets made with each iteration of the %DO loop. The full macro is written as follows:

```
%macro datasetlist;
%let datasetlist=  ;

%do i=10 %to 12;
%put &datasetlist.;

    data Sales_month&i.;
          set  ToySales;
                where month=&i.;
                drop month;
                rename Sales=Sales_&i.;
    run;
    %let datasetlist= &datasetlist. Sales_month&i.;
%end;
%put &datasetlist.;
%mend;
%datasetlist;
```

To better understand how this list of our datasets is generated, let's look at one iteration of the %DO loop at a time. In the first loop, *datasetlist* will resolve to a blank value prior to the creation of Sales_10. Once the DATA step has been executed, *datasetlist* is redefine to add Sales_10 to the initial blank value. Then we move on to the next iteration of the %DO loop. Each iteration looks as follows:

FIRST ITERATION (`I=10`):

Prior to DATA step:

```
datasetlist = ;

data Sales_month10;
          set ToySales;
                where month=10;
                drop month;
                rename Sales=Sales_10;
run;
```

After DATA step:

```
%let datasetlist = Sales_month10;
```

SECOND ITERATION (`I=11`):

Prior to DATA step:

```
datasetlist = Sales_month10;

data Sales_month11;

    Set ToySales;
          where month=11;
          drop month;
          rename Sales=Sales_11;
run;
```

After DATA step:

```
%let datasetlist = Sales_month10 Sales_month11;
```

THIRD ITERATION (I=12):

Prior to DATA step:

```
  datasetlist =Sales_month10 Sales_month11;
  data Sales_month12;
      Set ToySales;
             where month=12;
             drop month;
             rename Sales=Sales_12;
  run;
```

After DATA step:

```
  %let datasetlist = Sales_month10 Sales_month11 Sales_month12;
```

Now we have our list of datasets we can use in subsequent data processing such as SET or MERGE DATA step statements:

```
  data AllSales;
      merge &datasetlist.;
  run;
```

which resolves to:

```
  data AllSales;
      merge Sales_month10 Sales_month11 Sales_month12;
  run;
```

To determine the number of datasets using the above method, we have to get a little creative. We will want to work with the %DO loop to count how many datasets we are creating. Like how we needed to create an initial blank dataset, we would need to create a start value for the macro variable (*j*) we will use to count the number of datasets we are creating. In this case, our start value will be zero:

```
%let j=0;
```

Then, we would loop through each iteration and add a value of 1 to our macro variable *j* using the %EVAL function (since macro values are treated as text, we must use the %EVAL function to evaluate any arithmetic expressions using macro variables):

```
%do i=10 %to 12;
      %let j=%eval(&j.+1);
%end;
```

We can count the number of datasets in our list while we create the list:

```
%macro datasetlist;
%let datasetlist=  ;
%let j=0;

%do i=10 %to 12;
      data Sales_month&i.;
             set  ToySales;
                    where month=&i.;
                    drop month;
                    rename Sales=Sales_&i.;
      run;
      %let datasetlist= &datasetlist. Sales_month&i.;
      %let j=%eval(&j.+1);
      %put &datasetlist. and &j.;
%end;
%put I have &j datasets and they are &datasetlist.;

%mend;
%datasetlist;
```

The %PUT statement resolves to:

```
I have 3 datasets and they are Sales_month10 Sales_month11 Sales_month12
```

CONCLUSION

We have now explored how to make data-driven lists, how to verify these list with the %PUT statement and how to count the number of items we have in our lists. By using key concepts of macro programming, such as defining, calling and executing macros and macro variables, we can dynamically program and then use our lists. Using procedures and the SELECT statement, we can create lists of observations and variables. Creating a list of datasets requires a little out of the box thinking, but is really quite easy once you get the hang of it! You now have gained an addition to your dynamic programming toolbox and can create data-driven lists with ease.

REFERENCES

Carpenter, Art. 2004. *Carpenter's Complete Guide to the SAS Macro Language*, 2nd ed. Cary, NC: SAS Institute, Inc.

Eberhardt, Peter and Brill, Ilene (2006). *How Do I Look it Up If I Cannot Spell It: An Introduction to SAS Dictionary Tables.* SUGI 31 Proceedings. Cary, NC: SAS Institute, Inc.

ACKNOWLEDGMENTS

I would like to thank Andrew Clapson from MD Financial Manage in Ottawa, Canada for his support and review of this paper.

RECOMMENDED READING

- *Carpenter's Complete Guide to the SAS Macro Language*

CONTACT INFORMATION

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

Kate Burnett-Isaacs
Statistics Canada
Kate.Burnett-Isaacs@canada.ca