Paper 029-2013

"How Do I . . .?" There is more than one way to solve that problem; Why continuing to learn is so important

Arthur L. Carpenter
CA Occidental Consultants

ABSTRACT

In the SAS® forums questions are often posted that start with "How do I . . . ?". Generally there are multiple solutions to the posted problem, and often these solutions vary from simple to complex. In many of the responses the simple solution is often inefficient and also reflects a somewhat naïve understanding of the SAS language. This would not be so very bad except sometimes the responder thinks that their response is the best solution, or perhaps worse, the only solution. Worse yet, when there is a range of solutions, the 'right answer' that the original poster selects often reflects the simplest solution that the original poster understands. In both cases these folks have stopped learning and have stopped expanding their understanding of the language.

The examples in this presentation will illustrate the progression of solutions from the simple (simplistic) to the sophisticated for a number of 'How do I . . . ?' questions, and through the discussion of the individual techniques, we will learn how and why it is so very important to continue to learn.

INTRODUCTION

Within the SAS community it is often stated that for SAS there is almost always more than one solution to even the most difficult programming problem. Although SAS is a complex programming language, there are many simple solutions – often to even difficult problems. Sometimes these simple solutions are 'just right', but sometimes they are too simplistic and tend to be inefficient. Fortunately there are often several techniques that can be applied in any given situation, and it is not unusual that the range of solutions varies from simple to complex.

When programmers speak of efficiency they often refer to the speed of processing. The efficiency of the computer, however, should not be the only consideration. The programmer who will be maintaining the code must also be taken into consideration. A complex coding solution that takes hours to develop and maintain, but saves a few computational seconds may be less efficient than a simpler solution that is quick to develop and maintain. As the programmer gains technical prowess, the more complex solutions become quicker to program and simpler to maintain. The beginning programmer will only be able to develop and maintain the simple solutions, however, the advanced programmer has the ability to choose among solutions that the beginning programmer cannot even envision.

This is why we must force ourselves to continue to learn new and more complex elements of the language. As we grow in sophistication so too can our programs. As you look through the solutions to the problems presented in this paper, ask yourself how you would code the solution. It is likely that your solution will be different than any presented in this paper. One of the wonderful aspects of SAS is its flexibility to allow multiple solutions to virtually every problem.

THE CLASSIC TABLE LOOKUP

How do I assign a value to a variable based on a code in another variable?

The process of determining and assigning values to a variable based on the value in another variable is known as a lookup. There are many more techniques and types of lookups than the three primary ones shown here, however these approaches should illustrate the differences in the techniques and why it is so very important to understand them.

In the examples shown here we have gone to a farm and counted various kinds of animals. The number of animals observed and the animal's code have been recorded in the data set COUNTS. Based on the animal's code, we need to add the name of the animal to the data set.

IF-THEN/ELSE

The simplest form of table lookup uses IF-THEN processing. A series of IF statements without any corresponding ELSE statements is the slowest form of lookup. Simply adding an ELSE to all but the first IF statement, can make a big difference.

```
data counts;
input code count;
datalines;
1 23
4 15
2 3
5 27
3 5
run;

* Assign animal names with
* IF-THEN/ELSE processing;
data ifthen;
   set counts;
   if code=1 then animal='Chicken';
   else if code=2 then animal='Horse'; ❶
   else if code=3 then animal='Pig';
   else if code=4 then animal='Sheep';
   else animal='Unknown';
   run;
```

❶ Notice the ELSE for each of the IF statements (except for the first one). A slight performance enhancement can be realized if the most commonly occurring codes are arranged higher in the list.

When there are only a very few items to 'look up', as in this example, this form of lookup may be sufficient. However as the number of items increases, the coding itself can be burdensome. Three or four items is about my limit for a series of IF-THEN/ELSE statements.

Performance wise this form of lookup is roughly equivalent to a SELECT statement with a series of WHEN statements in the DATA step, and the CASE statement in a SQL step.

Merges and Joins

Another very common form of lookup is seen when the item to be 'looked up' is already in a data table. Here the data set CODES contains the association between code and animal.

```
data codes;
input code animal $;
datalines;
4 Sheep
1 chicken
3 Pig
2 Horse
run;
```

There are multiple ways to add the variable ANIMAL from CODES onto the data set COUNTS. The JOIN in SQL and the MERGE in the DATA step can produce very similar results, often with radically different efficiencies. In both cases there are multiple ways to structure the coding of the steps.

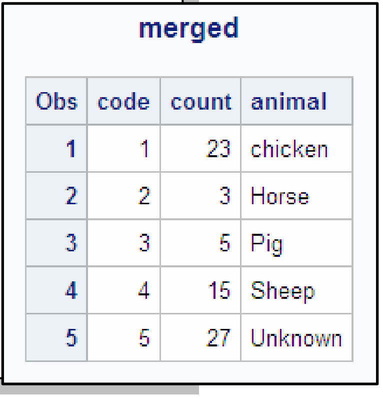
The SQL language reads the incoming data into memory and then performs operations such as joins. The DATA step on the other hand processes at the observation level. When the incoming table fits in memory, the join will usually be faster. This is especially true when you consider that the DATA step MERGE requires sorted data sets, and when the time to sort the data is included in the overall total compared to the SQL join.

The DATA step MERGE statement lists the data sets that are to be joined. Since the BY statement is required to perform a match merge, the two incoming data sets must first be sorted.

```
* Assign animal names using a merge;
proc sort data=counts;
   by code;
   run;
proc sort data=codes;
   by code;
   run;
data merged;
   merge counts(in=incounts)
         codes(in=incodes);
   by code;
   if incounts; ❷
   if not incodes then
              animal='Unknown'; ❸
   run;
```

❷ Only observations that appear in the COUNTS data set are to be included in the new data set (WORK.MERGED).

❸ When codes appear in the COUNTS data set, but not in the data set of codes, the value of ANIMAL is set to 'Unknown'.



merged

Obs	code	count	animal
1	1	23	chicken
2	2	3	Horse
3	3	5	Pig
4	4	15	Sheep
5	5	27	Unknown

In a SQL step, the LEFT JOIN most closely mimics the DATA step's MERGE.

```
proc sql noprint;
  create table joined as
    select c.code, c.count, ❼
          coalesce(a.animal,'Unknown') as animal ❻
      from counts as c left join codes as a  ❺
        on c.code=a.code; ❹
  quit;
```

❹ Similar to the WHERE, the ON phrase is used to select those rows which have the same value of CODE from each of the two data sets.
❺ The two data sets COUNTS and CODES are joined using a LEFT JOIN. Notice that the table COUNTS is listed first – as it is in the MERGE statement above.
❻ Missing values of ANIMAL are replaced with 'Unknown' through the use of the COALESCE function.
❼The variables to be included in the new data set (WORK.JOINED) are listed.

User Defined Formats

Lookups executed through the implementation of user defined formats will be faster than the IF-THEN/ELSE, MERGE, and JOIN techniques shown above. For a number of reasons this technique should probably be the user's first choice whenever there are more than just a few items to lookup.

Since formats can be permanently stored in libraries, a change in the format can be used to change assignments in numerous programs while making a change in only one place. The user defined format should be sufficiently fast for most users as long as the number of items in the format does not exceed 20 to 40 thousand.

The key to this technique is to create a user defined format. For this simple example, where we have only a few items to lookup, we could explicitly write the PROC FORMAT step. However as the number of items increases, writing the VALUE statement (or INVALUE statement for INFORMATS), becomes impractical.

```
proc format;
   value animals
      1 ='chicken'
      2 ='Horse'
      3 ='Pig'
      4 ='Sheep'
      other='Unknown';
   run;
```

Fortunately it is very often the case that the format can be based on an existing data set which already contains the pairings. For the barnyard animal example the pairs are found in the WORK.CODES data set. When this data set exists, it can be converted into a data set that PROC FORMAT can use to generate the format. PROC FORMAT uses the CNTLIN= option to identify this 'control' data set.

A control data set is built with specific variables. This data set is then passed to PROC FORMAT where the format is created based on the *instructions* contained in the data sets observations. As a minimum the control data set must, at the very least, have the variables FMTNAME, START, and LABEL. Over 20 additional variables are available, and these are used to implement the many options available to PROC FORMAT.

```
data control(keep=fmtname start label hlo); ❽
   set codes(rename=(code=start animal=label))
       end=eof;
   retain fmtname 'animals'; ❾
   output control;
   if eof then do;
      hlo='o'; ❿
      label='Unknown';
      output control;
   end;
   run;
```

❽ The variables needed in this control data set appear on the KEEP= data set option.

- FMTNAME is used to name the format.
- START contains the value to be mapped (the code).
- LABEL is the value being mapped to, in this case, the animal name.
- HLO allows for additional options, such as; HIGH, LOW, and OTHER

❾ The name of the format which is to be created is stored in FMTNAME.
❿ The HLO variable allows us to assign 'Unknown' to any code that is not otherwise assigned to an animal.

Once the control data set has been created, it is passed to PROC FORMAT using the CNTLIN= option.

```
proc format cntlin=control; ①
   run;
data fromfmt;
   set counts;
   animal = put(code,animals.); ②
   run;
```

① The control data set is converted into a format through the use of PROC FORMAT's CNTLIN= option.
② The PUT function (or the INPUT function in the case of INFORMATS) is used to perform the actual lookup.

Advanced LOOKUP Techniques

There are numerous other lookup techniques that have increasing complexity, and for large data sets, potentially huge performance gains. For most users in most situations these other techniques add program complexity without a corresponding performance gain. However when the number of items to lookup exceeds the efficient capacity of the FORMAT, they are available.

Some of the more advanced lookup techniques include:

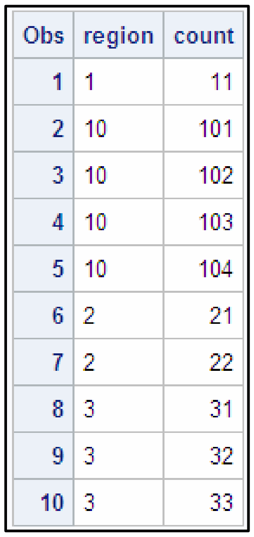
- Double SET statement merges
- The use of data set indexes
- Key indexing
- The use of DATA step hash objects

These techniques are outside the scope of this paper, but have been discussed extensively including the 2001 paper "Table Lookups: From IF-THEN to Key-Indexing", http://www2.sas.com/proceedings/sugi26/p158-26.pdf.

CHANGING THE ORDER OF DISPLAYED VALUES

\How do I change the order of displayed values?

The $2 variable REGION takes on the values of '1', '2', . . . ,'10'. When the values of this variable are displayed, the order of the values are '1', '10', '2', . . ., '9'. How can I get them to be displayed in the 'correct' order?



Obs	region	count
1	1	11
2	10	101
3	10	102
4	10	103
5	10	104
6	2	21
7	2	22
8	3	31
9	3	32
10	3	33

First of course is that they are in the correct order, as a character variable '10' sorts before '2' alphabetically. Consequently using PROC SORT is not going to be a helpful solution. The sort order is also known as the *internal* order.

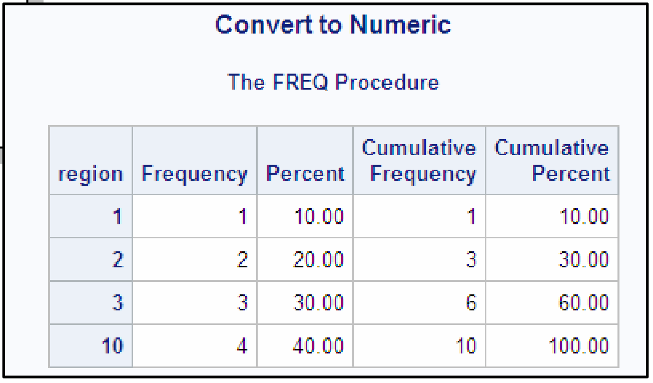
We see this ordering in the PROC PRINT of the sorted data set on the right as well as when the data is used in other procedures such as PROC FREQ, which is used in the examples that follow.

Convert to Numeric

The problem would go away if REGION was a numeric variable. The conversion of character to numeric values is best handled through the INPUT function in a DATA step.

```
data Nregion(keep=region count);
   set regions(rename=(region=creg));
   region=input(creg,2.); ❶
   run;
title1 'Convert to Numeric';
proc Freq data=nregion;
   table region;
   run;
```

❶ The INPUT function converts the character value in CREG to numeric. The result is stored in the numeric variable REGION, which can be used in PROC FREQ.



Convert to Numeric

The FREQ Procedure

region	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	1	10.00	1	10.00
2	2	20.00	3	30.00
3	3	30.00	6	60.00
10	4	40.00	10	100.00

Changing the Stored Value

Rather than create a new variable we might consider modifying the existing variable. Placing a blank in front of the values (other than '10') would cause them to sort differently.

```
data cregion;
   set regions;
   region=right(region); ❷
   run;
```

❷ The RIGHT function effectively inserts a blank by right justifying the value in the available two spaces.

Change the Stored Value

The FREQ Procedure

region	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	1	10.00	1	10.00
2	2	20.00	3	30.00
3	3	30.00	6	60.00
10	4	40.00	10	100.00

Using a User Defined Format

Each of the previous two techniques require an extra pass of the data. This is not a problem when the data sets are small, however if they are large, more efficient solutions are required. One possibility is the use of a user defined format.

```
proc format;
   value $rightreg ❸
      '1'  =  ' 1'
      '2'  =  ' 2'
      '3'  =  ' 3'
      '10' =  '10';
   run;
title1 'User Defined Format';
proc Freq data=regions
          order=formatted; ❹
   table region;
   format region $rightreg.; ❺
   run;
```

User Defined Format

The FREQ Procedure

region	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	1	10.00	1	10.00
2	2	20.00	3	30.00
3	3	30.00	6	60.00
10	4	40.00	10	100.00

❸ The format $RIGHTREG. is used to shift the values to the right.
❹ PROC FREQ automatically uses formatted classification variables (REGION) to form the groups, but by default the unformatted (INTERNAL) values determine the order. The ORDER= option can be used to control the display order, which in this case is based on the formatted values.
❺ The FORMAT statement is used to create the association with the format and the variable REGION.

This technique will be faster than either of the first two as an extra pass of the data is not required.

When there are more than just a few regions, you could call the RIGHT function directly from the format. The function appears as the label in square brackets and without quotes. The format $RTFUNC would be used in the same way as $RIGHTREG above ❺.

```
proc format;
   value $rtfunc
      other=[right()];
   run;
```

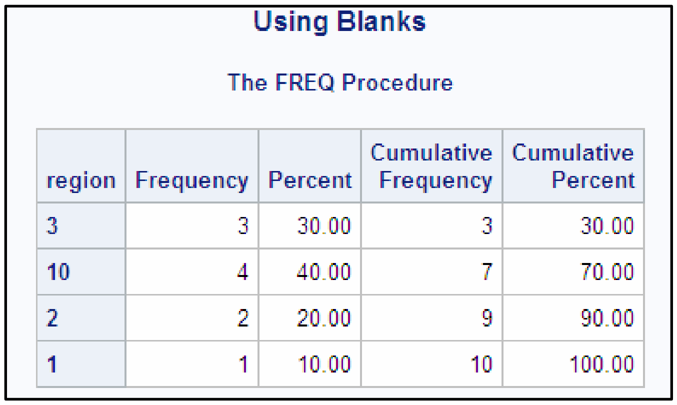
But what if I want region 3 to be first followed by region 10 then 2?

Trick the Format

Since blanks sort first, blanks can be inserted in the label portion of the format. Most ODS styles will left justify character strings after the order has been determined, consequently we can take advantage of this by adding as many leading blanks on the format label as are needed.

```
proc format;
   value $regorder
      '3'  =  '   3'
      '10' =  '  10' ❻
      '2'  =  ' 2'
      '1'  =  '1';
   run;
title1 'Using Blanks';
proc Freq data=regions
          order=formatted;
   table region;
   format region $regorder.;
   run;
```

❻ Three leading blanks will sort before two and so on.


Using Blanks

The FREQ Procedure

region	Frequency	Percent	Cumulative Frequency	Cumulative Percent
3	3	30.00	3	30.00
10	4	40.00	7	70.00
2	2	20.00	9	90.00
1	1	10.00	10	100.00

Using NOTSORTED

When neither the internal nor the formatted order is desired, and you are using the MEANS, SUMMARY, TABULATE, or REPORT procedures, you can combine the use of the NOTSORTED option on the PROC FORMAT statement along with preloading the format.

In this example we want the same region order as shown above, however we are not padding the label with leading blanks.

```
proc format;
   value $regorder (notsorted) ❼
      '3'  =  '3'
      '10' =  '10'
      '2'  =  '2'
      '1'  =  '1';
   run;
```

```
proc means data=regions nway noprint;
   class region / preloadfmt order=data; ❽
   output out=cnts n=number sum=totalcnt;
   format region $regorder.; ❾
   run;
```

```
proc freq data=cnts order=data;
   table region;
   weight number; ❿
   run;
```

❼ The NOTSORTED option on the VALUE statement preserves the internal order of the items as they are listed in the format definition. This option can effect efficiencies if the number of items in the format is large, but that will not be an issue for this format.
❽ Preloaded formats cannot be used with PROC FREQ, however they are very practical for the MEANS / SUMMARY procedure (see Carpenter 2012 for more on preloaded formats). The ORDER=DATA option allows the format's NOTSORTED order to be surfaced.
❾ The format is associated with the classification variable in the usual way.
❿ If w need to see the order preserved in the output from PROC FREQ, which does not support preloaded formats, we can use the data set generated by the MEANS procedure as input to FREQ. The WEIGHT statement forces the correct percentages to be calculated. Notice the use of the ORDER=DATA option, in this procedure it preserves the incoming data order, which was created by the MEANS procedure.

COMPARISON ACROSS COLUMNS

How do I determine if the value of one variable is in a list of other variables?

Comparing the values of any two columns is very straight forward, but when the number of columns to compare becomes large the solution is not so obvious. In the examples that follow we need to determine if the value in the variable VALUE is also in one or more of the other variables in the data set. If it is found the variable INLIST is set to 'yes', otherwise it is set to have a value of 'no'.

In these examples there are only three variables to which we want to compare: TRT1, TRT2, and TRT3; but we may in actuality need to allow for many more variables than just three.

Using IF-THEN/ELSE Processing

The simplest solution is to use IF-THEN/ELSE processing to compare the variable VALUE to the other variables. Certainly this code solution is straightforward; however it is not very expandable. Imagine rewriting the IF statement for 100 different TRT variables instead of just three!

```
data IFcheck;
   set values;
   if value=trt1 or
      value=trt2 or
      value=trt3 then InList='yes';
   else inlist='no';
   run;
```

Essentially the same solution can be seen in the following SQL step. Again the code will become difficult to maintain as the number of comparison variables increases.

```
proc sql;
create table SQLWhen as
   select *,
          case
             when (value=trt1 or value=trt2 or value=trt3 )then 'yes'
             else 'no'
          end
            as InList format=$3.
      from values;
quit;
```

Using SQL and the WHICHC Function

```
proc sql;
create table SQLcheck as
   select *,
     case
        when whichc(value,trt1,trt2,trt3 )then 'yes'
        else 'no'
     end
          as InList format=$3.
       from values;
quit;
```

A variation on the previous SQL step utilizes the WHICHC function. This function compares the first argument to the remaining arguments and returns the number of the first match, and a 0 if no matches are found.

Using the IN Operator

The IN operator will also compare an item to a list of values, however it cannot compare a variable’s value to a list of variables as is done here. Syntax such as is shown here will result in an error. This solution, even if it worked, would have the same limitation as the first solutions in that each variable to be compared would need to be listed separately.

```
data INCheck;
   set values;
   * This will not work !!!!;
   if value in(trt1,trt2,trt3) then InList='yes';
   else inlist='no';
   run;
```

Although the IN operator cannot be used with a list of variables, it can be used to access the elements of an array – which is essentially the same thing. This is the first solution for this problem that we have looked at in this section that will work for any number of variables that start with the letters TRT.

```
data INCheck;
   set values;
   array trtval {*} trt:; ❶
   if value in trtval ❷ then InList='yes';
   else inlist='no';
   run;
```

❶ The array TRTVAL is declared. Its dimension is not specified and the list of variables includes all variables that start with the letters TRT.
❷ Rather than list the variables only the array name is used. The comparison will be made for each element of the array.

Using WHICHC in the DATA Step

This solution varies only slightly from the previous one. Instead of the IN operator the WHICHC function is used in a fashion similar to how it is used in the SQL step shown above.

```
data WhichCheck;
   set values;
   array trtval {*} trt:; ❸
   if whichc(value,of trtval{*}) then InList='yes'; ❹
   else inlist='no';
   run;
```

❸Again an array is set up which will contain all the comparison variables.

❹ The second argument of the WHICHC function is the array. Notice that the syntax is different in this function call as compared to how the array was called when used with the IN operator ❷.

Discussion Thread

A question very similar to this one was posted on LinkedIn. The SQL solution using the WHICHC function was suggested by Patrick Mather.
http://www.linkedin.com/groupItem?view=&srchtype=discussedNews&gid=70702&item=212076970&type=member&trk=eml-anet_dig-b_pd-ttl-cn&ut=2QQE3h2AWA4lE1

COUNTING OBSERVATIONS

How do I determine how many observations are in my data set?

The number of observations in a data set can be important for a number of reasons. One of the most common is to determine whether or not the data set is empty (0 observations). In each of the examples in this section the macro variable &DSOBS is created to hold the number of observations in the data set WORK.REGIONS.

Counting Observations with the DATA Step

When a DATA step is executed there is an implied loop that executes for each incoming observation. The automatic temporary variable _N_ counts the iterations of this implied loop; and by inference in this simple DATA step, the number of observations on the incoming data set. The last observation is detected by the END= option which sets the temporary variable EOF to 1. The SYMPUTX routine is then used to load the largest value of _N_ into the macro variable &DSOBS.

```
data _null_;
   set regions end=eof;
   if eof then call symputx('dsobs',_n_);
   run;
%put There are &dsobs observations.;
```

Compared to the techniques that follow, this is a very inefficient way to obtain the number of observations. Not only do we have the overhead of the DATA step, but worse yet, all the incoming observations must be read and counted. At least we are using the _NULL_ to avoid the creation of a new data set.

Using SQL to Count Observations

Like in the previous example the observations can be counted in an SQL step. The previous DATA _NULL_ step is roughly equivalent to this SQL step. The automatic macro variable &SQLOBS counts the number of observations and this value is written to the macro variable &DSOBS. Although no data table is created, like the previous DATA step all observations must be read and processed.

```
proc sql noprint;
create table _null_ as
   select *
     from regions;
   quit;
%let dsobs=&sqlobs;
%put number of obs is &dsobs;
```

A similar alternative to using the macro variable &SQLOBS is the COUNT function. In this example the COUNT function is used to count all the observations in the incoming data set. The result of the COUNT function is then written into the macro variable &DSOBS. The SEPARATED BY clause is not necessary here, however by including it, the value stored in the macro variable is left justified and trimmed.

```
proc sql noprint;
   select count(*)
      into :dsobs separated by ' '
         from regions;
   quit;
%put number of obs is &dsobs;
```

Like the DATA step approach shown above, this SQL step is inefficient in that all the observations of WORK.REGIONS must be read in order for them to be counted.

Using CONTENTS to Access the Metadata

Since the number of observations in a data set is always stored as a part of that data set's metadata, it is not necessary to physically count the observations. Each of the methods shown below takes advantage of this metadata one way or another. These approaches are not equivalent in either programming complexity or processing efficiency.

```
proc contents data=work.regions
              out=cont noprint;
   run;
data _null_;
   set cont(keep=nobs);
   call symputx('dsobs',nobs);
   stop;
   run;
%put From Contents count is &dsobs;
```

Since PROC CONTENTS specifically works with metadata, we can use it to abstract the value of interest. Using the OUT= option results in a data set with one observation per variable in the data set. Some variables such as NOBS will be a constant across all observations. We can read this data set and place the value of the variable NOBS into the macro variable &DSOBS using the SYMPUTX routine. Notice the use of the STOP statement so that we only read one of the observations.

Using the NOBS= SET Statement Option

In the DATA step, the SET statement option NOBS= retrieves the number of observations stored in the metadata. In this example the NOBS= option is used to place the number of observations in the temporary variable (N_OBS) during the compilation phase of the DATA step. Then during the execution phase this value is transferred to the macro variable &DSOBS by the SYMPUTX routine. Notice the use of the stop statement to prevent the reading of any observations from the data set WORK.REGIONS.

```
data _null_;
   call symputx('dsobs',n_obs);
   stop;
   set regions nobs=n_obs;
   run;
%put There are &dsobs observations.;
```

Indirect Access of the Meta Data

Potions of the metadata of SAS data sets are also available through a series views in SASHELP and SQL DICTIONARY tables that are automatically created by SAS. Of particular interest for this example are SASHELP.VTABLE and DICTIONARY.TABLES.

The views in the SASHELP library are accessed as you would any SAS data set. SASHELP.VTABLE contains one observation for each data set known to SAS. Here the specific data set of interest has been selected in a WHERE clause and the value, which contains the number of observations and is stored in the variable NOBS, is written to the macro variable &DSOBS.

```
data _null_;
   set sashelp.vtable(
      where=(libname='WORK' &
             memname='REGIONS'));
   call symputx('dsobs',nobs);
   run;
```

Since a view is created when requested, and since there may be a great many data sets known to SAS, the creation of this view may be noticeably time consuming. When this view is requested the entire table is created *before* the WHERE clause is applied, consequently selecting only one row from the view is unlikely to improve the performance of this technique.

When using SQL, you may access both the SASHELP views as in the previous step as well as a series of DICTIONARY tables only available in a SQL step. Like views, the DICTIONARY tables are always current, but they tend to load faster than the SASHELP views.

Similar to SASHELP.VTABLE the table DICTIONARY.TABLES has one row per data set known to SAS. It also contains the variable NOBS, which contains the number of observations. In this SQL step the value of the variable NOBS is written to the macro variable &DSOBS. The WHERE clause has been specified to select only the table of interest.

```
proc sql noprint;
select nobs
   into :dsobs
      from dictionary.tables
         where libname='WORK' &
               memname='REGIONS';
quit;
```

In each of the previous two examples we have wanted information from a specific data set, but have created either a SASHELP view or a DICTIONARY table with many rows, all but one of which were discarded.

Reading the Metadata Directly

In terms of processing speed the fastest way to retrieve the number of observations is to use the macro language to read the value directly. The following macro function returns the number of observations in the data set named in its one parameter ❶.

```
%macro obscnt(dsn=); ❶
%local dsid obs; ❷
%let dsid = %sysfunc(open(&dsn)); ❸
%if &dsid %then
        %let obs = %sysfunc(attrn(&dsid,nlobs)); ❹
%else %let obs = .;
%let dsid = %sysfunc(close(&dsid)); ❺
&obs ❻
%mend obscnt;
%let dsobs = %obscnt(dsn=work.regions); ❼
```

❶ The name of the data set of interest is held in the parameter &DSN.
❷ This is a macro function, therefore all macro variables created must be placed in the local symbol table.
❸ The OPEN function allows access to the data set. This function returns a non-zero value when the named data set exists and is available for use. Stored in the macro variable &DSID, this value is used in the ATTRN and CLOSE functions.
❹ The ATTRN function returns numeric attributes from the metadata. The first argument is the data set id number (&DSID) and the second is the attribute to be retrieved. NLOBS returns the number of non deleted observations.
❺ Good practices require that the data set be closed after use. This allows the data set to be used by others as well.
❻ The value to be returned is written by the macro.
❼ The %OBSCNT macro returns the number of observations, and the resulting %LET statement is:

```
%let dsobs = 10;
```

Counting Observations Thread

A similar discussion can be found on a LinkedIn thread:
http://www.linkedin.com/groupItem?view=&srchtype=discussedNews&gid=70702&item=209843149&type=member&trk=eml-anet_dig-b_pd-ttl-cn&ut=3vaF17OB5xbRE1

RETURN THE LAST 10 OBSERVATIONS

How do I create a data set containing only the last 10 observations from my incoming data set?

When we need to read a selected subset of the incoming data set, we need to establish a criterion for the selection. In this series of examples the criterion is to select only the last 10 observations from the incoming data set. The problem of course is to know which observations are in the last 10.

Some of the solutions shown here require the programmer to know the total number of observations in the data set. Several solutions to this separate problem were discussed in the previous sections of this paper, and that portion of the techniques shown below will not restate them.

Reading all Observations and Counting Them

One of the first solutions is to read all observations, number them and then only output the last 10. This requires two passes of the data. This solution is only included here so that I can discourage its use. In the first pass an observation counter (CNT) is added to the data set and the total number of observations is then saved in a macro variable. Note that the macro variable has been resaved as each observation is counted. Then in a second pass of the data a determination is made on the acceptability of each observation. We have had to read every observation twice. For large data sets this would be very inefficient.

```
data nums;
   set sightings;
   cnt+1;
   call symputx('count',cnt);
   run;
data nums10(drop=cnt);
   set nums;
   if cnt > (&count-10);
   run;
```

If we know how to determine the number of observations without counting them, we can solve the problem with a single pass of the data.

The DATA step solution shown here utilizes the NOBS= option on the SET statement. The total number of observations is stored in OBSCNT. Each observation is read and the temporary variable _N_ is automatically incremented. When _N_ is within the last 10 observations the current observation is written to the new data set. There are several DATA step and SQL step variations on this solution.

```
data lastten;
   set sightings nobs=obscnt;
   if obscnt-_n_ lt 10 then output lastten;
   run;
```

Because all observations are read, this solution is clearly still less than optimal, especially as the size of the incoming data set increases. Rather than reading all observations and throwing away what we do not want, we would like a solution that reads only the observations of interest.

Using FIRSTOBS

The FIRSTOBS system option is used to declare the number of the first observation to read on a sequential read. If we set `options firstobs=10;` all subsequent data reads will start at observation 10. This is of course dangerous if we forget to reset it back to 1. Fortunately there is a equivalent data set option that we do not need to reset. We can take advantage of the FIRSTOBS= data set option if we can calculate the number of the observation to at which to start the read. This is an easy calculation if we know the total number of observations in the incoming data set.

In this solution the macro function %OBSCNT, which was discussed previously, is used to return the total number of observations. If we assume that the incoming data set (SIGHTINGS) has 459 observations, the macro call is replaced with a number, say 459, and the macro function %EVAL forces the subtraction to take place within the macro processor. This is important as the result (450) will be available when the SET statement is compiled. It is as if we written :

```
data firstobs;
   set sightings
          (firstobs=%eval(%obscnt(dsn=work.sightings)-9));
   run;
```

```
     set sightings(firstobs=450);
```

A very nice side benefit of using this form of calculation in the data set option is that it is not constrained to the DATA step. The same option would be applied virtually anywhere the incoming data set is named.

```
proc print data=sightings(firstobs=%eval(%obscnt(dsn=work.sightings)-9));
   run;
```

If the possibility exists that there might be fewer than 10 observations in the data set additional checking would be necessary. This could be accomplished through the use of the MAX function.

```
proc print data=sightings(firstobs=
                 %sysfunc(max(%eval(%obscnt(dsn=work.sightings)-9),1));
   run;
```

Using the POINT= Option

Normally the SET statement causes the observations to be read in sequential order. However it is possible to read the observations in any order, and this is accomplished through the use of the POINT= option on the SET statement. The POINT= option names a temporary variable which contains the number of the next observation that is to be read.

By setting up a non-sequential read of the incoming data set we are no longer constrained to starting the read at the first observation.

```
data pointer;
   do ptr = obscnt-9 to obscnt; ❸
      set sightings point=ptr ❷
                   nobs=obscnt; ❶
      output pointer;
   end;
   stop;❹
   run;
```

❶ The NOBS= option stores the total number of observations in the temporary variable OBSCNT.
❷ The POINT= option identifies the variable PTR as one that will contain the value of the next observation which is to be read.
❸ The index of this DO loop will range so as to cover the last 10 observations. Only these observations will be read.
❹ Whenever the SET statement appears inside of a DO loop (this is known as a DOW loop), there is the possibility of creating an infinite loop. This is most likely when the POINT= option is involved. The STOP statement terminates the DATA step (by now we have read the 10 observations) and prevents an infinite loop.

This is an efficient solution; however it does require the programmer to understand the concepts of SET statement options and the DOW loop.

Working with SAS Dates

This last "How do I…" question comes directly from a SAS Forums thread (https://communities.sas.com/thread/41182). The two primary responses are both to the point, but because they take such different approaches they illustrate how varied our programming responses can be.

How do I remove leading zeros in days and months when using the MMDDYY format?

The date value is to be displayed in the MMDDYY form, however if either the day or month values is a single digit the leading zero is not to be displayed. By default the MMDDYY format will display leading zeros for both months and days. What we want is shown below:

```
data sample;
input dt MMDDYY10. ;
format dt mmddyy10.;
datalines ;
11/03/2012
04/11/2012
10/20/2012
;
run;
```

```
MM/DD/YY.     desired
11/03/2012 → 11/3/2012
04/11/2012 → 4/11/2012
```

Create a new variable

One solution is to create a character variable with the zeros removed. This is a fairly simple process and it can be accomplished in several ways. The solution shown here (suggested by @Tom) takes the three numeric portions of the date and converts them to character using the CATX function.

```
data sample1;
set sample;
dt1=catx('/',month(dt),day(dt),year(dt)); ❶
run;
```

❶The MONTH and DAY functions return numeric values (without leading zeros and these are the values that are converted and inserted into the string.

Notice that like the INPUT function, CATX performs the numeric to character conversion without writing a conversion note to the LOG.

Create a new format

Rather than create a new variable as was done in the previous solution, we can also create a format that will accomplish the same thing. In this solution (suggested by @Peter Crawford), date directives are used in a PICTURE format.

```
proc format;
   picture mydate ❷
      low-high = '%m/%d/%Y' ❸
                 (datatype= date) ; ❹
   run;

data sample2;
   set sample;
   dt2=put(dt,mydate.);❺
   run;
```

❷ Create the picture format MYDATE.
❸ The case sensitive %m, %d, and %Y date directives return the numeric month, day, and year values respectively, without leading zeros.
❹ The DATATYPE option tells the format how to interpret the incoming value.
❺ Here the PUT function is used to create a second variable, however the format could have been used in a PROC step without creating the second variable.

HOW DO I CONTINUE TO LEARN?

Devote 20 to 30 Minutes a Day to Learning

Set aside some time everyday for learning something new. This is not easy to do, there are too many demands for our time, but this is guaranteed to save you time in the long run. If the boss asks you what you are doing, tell her: “Art told me to do this.”

Read User Papers

Thousands of papers have been written for various SAS user group conferences. Written by the people who use SAS, these papers tend to concentrate on the useful and helpful. Almost 24,000 SAS related papers have been collected and indexed by Lex Jansen. His web site (http://lexjansen.com/) has become a primary entry portal for finding SAS papers.

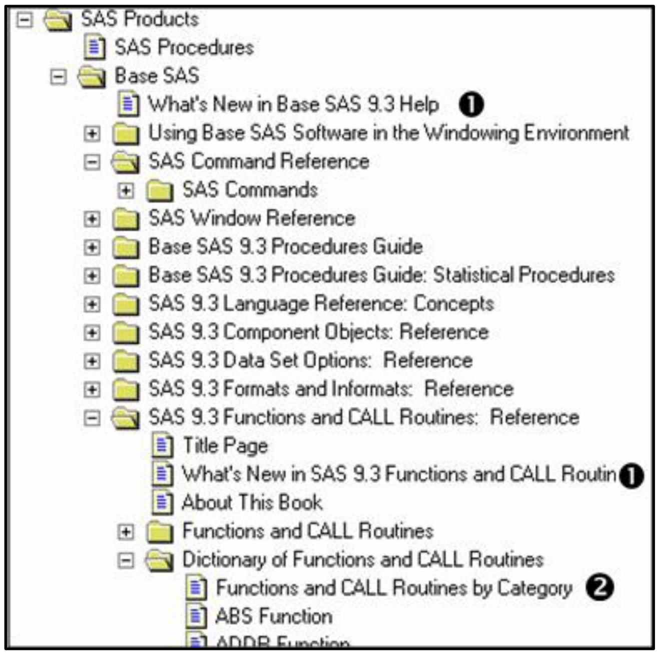
The first SAS papers were published in 1976. Many are of course now dated, but many others like the one on the “GENERAL LINEAR MODELS PROCEDURE” by Dr. J. Goodnight (http://www.sascommunity.org/sugi/SUGI76/Sugi-76-02%20Goodnight.pdf) are still valuable today.

Look for User Written (SAS Press) Books

Unlike the documentation that tells us about the language, books written by the users of the language tend to tell us how to use it, and perhaps more importantly, why to apply the various techniques. SAS Press (http://support.sas.com/publishing), formerly known as Books by Users, specializes in these books. There are now hundreds of books, and they have been written on virtually every SAS topic.

Study the Documentation

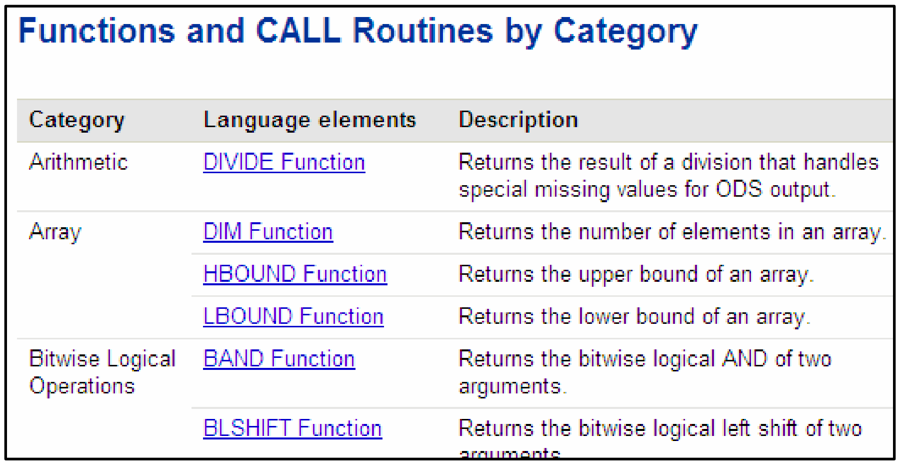
Don't just *use* the documentation to 'look something up', *study it*. Use it as a learning tool.


SAS Products
SAS Procedures
Base SAS
What's New in Base SAS 9.3 Help ❶
Using Base SAS Software in the Windowing Environment
SAS Command Reference
SAS Commands
SAS Window Reference
Base SAS 9.3 Procedures Guide
Base SAS 9.3 Procedures Guide: Statistical Procedures
SAS 9.3 Language Reference: Concepts
SAS 9.3 Component Objects: Reference
SAS 9.3 Data Set Options: Reference
SAS 9.3 Formats and Informats: Reference
SAS 9.3 Functions and CALL Routines: Reference
Title Page
What's New in SAS 9.3 Functions and CALL Routin❶
About This Book
Functions and CALL Routines
Dictionary of Functions and CALL Routines
Functions and CALL Routines by Category ❷
ABS Function

❶If you have been using SAS for awhile be sure to read the "What's new" section. This is where you will find out about new options and new capabilities of existing tools.

When we have been using a tool for awhile, we tend not to look it up in the documentation, why would we? But the fact is that even older elements evolve and often they receive new options. The COMPRESS option originally only removed blanks, now it can remove any number of characters and in a number of different ways.

❷The SAS language has a great many options, statements, formats, informats, and functions. There are too many to learn. How do you even determine which of the over 450 functions you need to learn more about? Whenever a category has a long list of items, the documentation has a "BY Category" selection. This is a very good place to start to learn about what is available. The box to the left shows the first few functions in the "Functions and CALL Routines by Category". If you were to read only the one or two sentence description of 20 or so functions a day, it would take you about a month to get through all the functions. Many of these functions you will never use, but how will you know about what is available if you do not at least skim through the brief descriptions. In my consulting I cannot tell you how many times a client has shown me some cool code when they could have used an existing function.


Functions and CALL Routines by Category

Category	Language elements	Description
Arithmetic	DIVIDE Function	Returns the result of a division that handles special missing values for ODS output.
Array	DIM Function	Returns the number of elements in an array.
	HBOUND Function	Returns the upper bound of an array.
	LBOUND Function	Returns the lower bound of an array.
Bitwise Logical Operations	BAND Function	Returns the bitwise logical AND of two arguments.
	BLSHIFT Function	Returns the bitwise logical left shift of two

Lurk in the SAS Forums

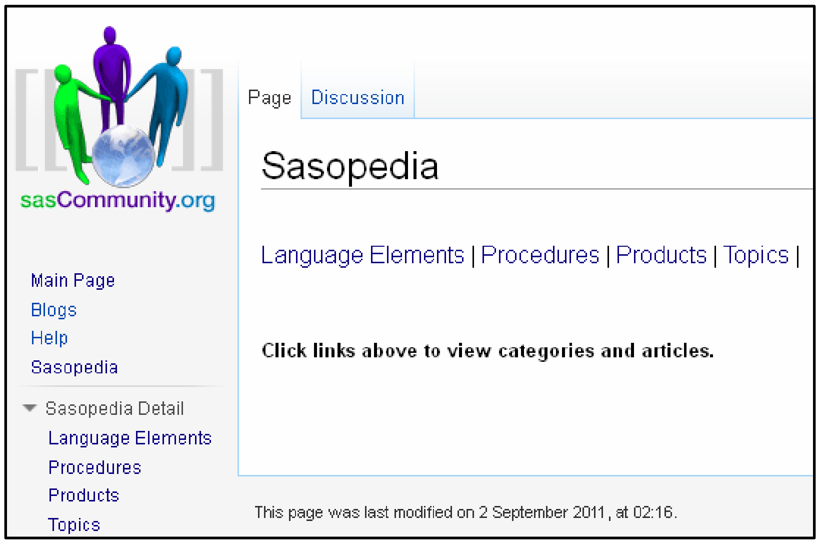
SAS Technical Support is wonderful, but it is not an interactive forum. The SAS forums (https://communities.sas.com/community/support-communities) allow users to post questions and some of the best SAS programmers in the world provide answers. I regularly learn new techniques and approaches to solving problems by reading their input. You need to establish a SAS profile to contribute (this is a good idea anyway as it allows you to sign up for all sorts of information about SAS), but anyone can browse the site and read responses. The idea for this paper came from reading multiple solutions to a problem posted in one of the forums.

Explore sasCommunity.org

sasCommunity.org is a wiki site established and run by the SAS Global Users Group. Because it is user supplied content much of its information is very practical in nature. The site features a 'Tip of the Day', which tends to offer very 'How to' oriented tips on various aspects of SAS.

Also very popular is the Sasopedia section. You can access thousands of user written articles by examining:

Language Elements	Things like statements, options, functions
Procedures	Procedure name
Products	SAS products and modules
Topics	General topics like the Macro Language



Sasopedia

Language Elements | Procedures | Products | Topics |

Click links above to view categories and articles.

This page was last modified on 2 September 2011, at 02:16.

Two papers that will help you get started with this site include:
Carpenter, Art and Don Henderson," Taking Full Advantage of sasCommunity.org: Your SAS® Site"
http://support.sas.com/resources/papers/proceedings12/157-2012.pdf

Carpenter, Art, "sasCommunity.org - Your SAS® Site: What it is and How to Get Started"
http://www.sascommunity.org/wiki/images/f/f2/83_sasCommunitySuperDemo.pdf

Read Blogs

There are a great many blogs about SAS; perhaps too many to keep track of them all. Fortunately there are a couple of easy to reach blog consolidators. If you regularly go to sasCommunity.org (and I hope you will after reading this paper) take a look at sasCommunity's blog planet (http://www.sascommunity.org/planet/). SAS Institute also has a collection of blog pages that you are likely to find interesting (http://blogs.sas.com/content/).

Find a few blogs that you like and read them.

Take Classes and Attend Conferences

Never stop learning. Instructor led classes are offered by SAS Institute, as well as, by a number of independent SAS trainers. Most user conferences offer classes as a part of the conference, and these tend to be especially good deals. The papers presented at conferences can yield wonderful insight into how others have approached and solved problems.

Find out more about user conferences in your area by visiting support.sas.com (http://support.sas.com/usergroups/).

SUMMARY

As SAS programmers we regularly encounter problems that require coding solutions. Fortunately for us the SAS language is both powerful and flexible. As a result of this flexibility we have the ability to solve most problems in more than one way, and the various solutions will not all have the same efficiency (for the computer or for the programmer).

As we continue to learn about the complexities of the SAS language, we should be constantly looking for new ways to solve old problems. We need to be open to exploring learning opportunities that push the boundaries of our knowledge. We must continually fight the urge to think that "I know it all" or perhaps worse "I know enough".

ABOUT THE AUTHOR

Art Carpenter's publications list includes five books, and numerous papers and posters presented at SUGI, SAS Global Forum, and other user group conferences. Art has been using SAS® since 1977 and has served in various leadership positions in local, regional, national, and international user groups. He is a SAS Certified Advanced Professional programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

sas Certified Clinical Trials Programmer Using SAS®9

sas® Certified Advanced Programmer *for* SAS®9

REFERENCES

Similar examples of some of the more advanced techniques discussed in this paper can befound in the book Carpenter's Guide to Innovative SAS® Techniques by Art Carpenter (SAS Press, 2012).

Carpenter, Art, 2001, "Table Lookups: From IF-THEN to Key-Indexing", http://www2.sas.com/proceedings/sugi26/p158-26.pdf

Carpenter, Arthur L, 2012, "Programming With CLASS: Keeping Your Options Open". Published in the conference proceedings for PharmaSUG 2012, WUSS 2012, MWSUG 2012. http://www.pharmasug.org/proceedings/2012/TA/PharmaSUG-2012-TA10.pdf

TRADEMARK INFORMATION