

Using Functions and Arrays in the SAS® System to Manage and Manipulate Data

Ben Cochran, The Bedford Group, Raleigh, NC

Abstract

Oftentimes, SAS users receive data that is alleged to be cleaned, scrubbed, and ready for an application or placement into a Data Warehouse. Even though the data may be clean, many times further manipulation is needed. This paper examines ways of manipulating data through the use of functions and arrays in the SAS System.

This paper is intended for the beginning and intermediate level SAS user. Functions and arrays are a part of the base SAS System; therefore, no other SAS product will be examined in this paper.

Basic Data Manipulation

Anytime users need to change existing data to meet our application needs, they are manipulating the data. One of the most basic forms of manipulating data is rearranging the order of the observations, or, in other words, sorting it. Sorting data not only rearranges the order of the observations, it allows for **by-group** processing. An example of this is merging two data sets by a common variable.

Another way to manipulate data is through the use of SAS functions.

SAS Functions

A SAS function is a pre-written routine that returns a value. Functions are often categorized by the type of data manipulation performed. These categories include: Data Conversion, Truncation, Arithmetic, Mathematical, Trigonometric, Statistical, Financial, Random Number, Date and Time, and State and Zip.

The general form of a SAS function is:

function-name (argument, argument, ...);

The arguments are surrounded by a set of parentheses. The number of arguments depends on the function. Some functions take a specific number of arguments, some functions have a varying number of arguments, some functions do not have any arguments at all. But there is ALWAYS the set of parentheses that follow the function. Each argument is separated from the others by a comma.

The arguments can be: constants, expressions, variables, or other functions. In other words, functions can be nested.

When using functions to manipulate data several methods can be used to do this. Following are some of the more popular ways to use functions.

Creating Variables with Functions

Many times we need to create values from existing data because what we need for a report or other application is not originally in the data. This is done by using an assignment statement which it has the following syntax:

new_variable = expression ;

New variables can be created by using functions as part of the expression. In this case, the assignment statement would look like this:

x = function-name(argument, argument, . .);

Manipulating Date Variables:

One of the most common examples of creating new variables is found in the manipulation of DATE values. For example, we may be asked to build a drill-down bar chart in SAS/EIS where

the initial view shows a bar for each year. The length of each bar reflects the number of tests performed by a clinic. Clicking on a bar (YEAR) would then yield a display showing a bar for each quarter within the chosen year. Clicking on a bar (QUARTER) would then yield a display showing a bar for each month within the chosen quarter, etc.

In order to accomplish this task, there needs to be a variable for YEAR, QUARTER, and MONTH in the existing data. Upon examining the data, we find that there is NO variable for YEAR, QUARTER, or MONTH. Looking closer, we see that there is a variable named DATE that contains the SAS date of the test. If we know that there are functions available to us that can create the needed variables, we can use them in a DATA step similar to the one below:

```
data new_data;
  set dates;
  year = year(date);
  quarter = qtr(date);
  month = month(date);
run ;
```

In the above example, notice that the variables being created are named YEAR (from the YEAR function), QUARTER (from the QTR function), and MONTH (from the MONTH function). Function names are not reserve words in the SAS language.

Manipulating Other Numeric Variables:

Suppose we want the highest, lowest, and average cholesterol readings for each patient. The data is in a SAS data set named **CHOLESTEROL** and has the following form:

Pat_id	date1	chol1	date2	chol2	date3	chol3
1201	010600	211.1	070200	201.4	112100	191.1
1212	010400	181.7	063000	188.0	120100	191.8
1222	020100	194.8	071100	195.3	121100	196.0
1261	011800	199.7	062800	201.1	120600	211.1
...						

In order to get the desired statistics from this data, write a data step using the **MIN**, **MAX**, and **MEAN** functions. The **MIN** function returns the minimum value from its arguments. The **MAX** function returns the maximum value from its arguments. The **MEAN** function returns the average from its arguments. Notice how these functions are used in the following DATA step:

```
data cholesterol_stats (keep=pat_id high
                        low average) ;
  set cholesterol;
  high = max ( of chol1 - chol3);
  low = min (of chol1 - chol3);
  average = mean (of chol1 - chol3);
run;
```

In the above DATA step, the variables HIGH, LOW and AVERAGE are created for the MAX, MIN, and MEAN functions, respectively. In this example, the arguments are NOT separated by commas. Instead, a variable list notation, the single dash ('-'), is used. Also, notice the use of the keyword 'OF' in the argument list. The SAS system interprets this as the variable list starting with CHOL1 and going through CHOL3. If the word 'OF' were not used in this example, the SAS system would interpret the CHOL1 – CHOL3 as a subtraction equation.

Suppose your data was stored in a SAS data set named **CHOLESTEROL2** and has the following structure.

Pat_id	date	chol_reading
1201	010600	211.1
1201	070200	201.4
1201	112100	191.1
1212	010400	181.7
1212	063000	188.0
1212	120100	191.8
1222	020100	194.8
1222	071100	195.3
1222	121100	196.0
...		

Suppose we are asked to develop an application that can indicate whether a patient's cholesterol reading has gone up or down from the previous reading. In order to get the desired results from this data, it will have to be manipulated with the **LAG** function as in the DATA step below:

```
data compare;
  set cholesterol2;
  by pat_id;
  last_reading = lag(chol_reading);
  if first.pat_id = 1 then last_reading = . ;
  difference = chol_reading - last_reading ;
run;
```

At compile time, the **LAG** function creates a buffer. During the execution of the DATA step, the value of CHOL_READING is stored in this buffer and then assigned to the variable

LAST_READING in the next execution of the DATA step. But, if the first observation from a PAT_ID is being processed, then there is no previous value for CHOL_READING (for that patient) so this DATA step sets the value of LAST_READING to missing. The resulting data set, **COMPARE** can be seen below:

Pat_id	date	chol_reading	last_reading	difference
1201	010600	211.1	.	.
1201	070200	201.4	211.1	-9.7
1201	112100	191.1	201.4	-10.3
1212	010400	181.7	.	.
1212	063000	188.0	181.7	6.3
1212	120100	191.8	188.0	3.8
1224	020100	194.8	.	.
1224	071100	195.3	194.8	.5
1224	121100	196.0	195.3	.7
...				

You can generate **random numbers** from various distributions using random number functions. One of the uses of this type of function is found in applications that generate random samples.

The **PATIENTS** data set contains thousands of observations. The first four observations are:

Pat_id	name	address	city	st
1201	Welch, W. B.	21 East St.	Apex	NC
1212	Coxe, Jan S.	43 King Rd.	Cary	NC
1224	Dow, Sue A.	19 Elm St.	Garner	NC
1261	Moore, Ron	16 Oak Av.	Apex	NC
...				

You want to create a sample of 25 randomly chosen patients. You can do this with the following DATA step which uses the **CEIL** and **RANUNI** functions.

```
data sample_of_25;
  do i = 1 to 25;
    rn = ceil(ranuni(0) * total_obs);
    set patients point=rn nobs=total_obs;
    output;
  end;
  stop;
run;
```

First, look at the **NOBS=** option on the SET statement. At compile time, it reads the descriptor portion of the PATIENTS data set, finds out the number of observations in the data set, and stores that number in the variable TOTAL_OBS.

Both the CEIL and the RANUNI functions are used to create the variable RN. First, the RANUNI function returns a randomly generated number between 0 and 1. Then, that randomly generated number is multiplied by the value of TOTAL_OBS. The result is some number that is between something less than 1 and the number of observations in the PATIENTS data set. This number is then fed into the CEIL function that rounds up to the next highest integer. If the previous number is already an integer, the CEIL function has no effect. The result of this is a whole number whose value is between 1 and the number of observations in the PATIENTS data set. This number is then assigned to the variable called RN.

The POINT= option causes the SET statement to read the RNth number in the PATIENTS data set. Then this observation is OUTPUT to the SAMPLE_OF_25 data set. As this process goes through the DO LOOP 25 times, 25 observations are created for the new data set.

The STOP statement is absolutely necessary to prevent an endless loop. (The DATA step will continue to loop because the end of file is never encountered on PATIENTS data set).

Manipulating Character Variables:

The billing department of the clinic wants to use the PATIENTS data set to create mailing labels for patient bills. To do so, the values of the NAME variable need to be manipulated as follows:

Pat_id	name	→	fullname
1201	Welch, W. B.		Mr. W.B. Welch
1212	Coxe, Jan S.		Ms. Jan S. Coxe
1222	Dow, Sue A.		Ms. Sue A. Dow
1261	Moore, Ron		Mr. Ron Moore
...			

In this data set, the PAT_ID ends in a 1 if the gender of the patient is male, or ends in a 2 if the patient is female. PAT_ID is a character variable with a length of 4. Some of values contain four digits, some contain three. So, we need to examine the last digit, if it is a 1, then the courtesy title is Mr., else if the digit is a 2, then the courtesy title is Ms.

The name field needs to be transformed from lastname, firstname, and middle initial to courtesy title, firstname, middle initial, lastname.

The following data step uses the SUBSTR, LENGTH, SCAN, and TRIM functions and the CONCATENATION operator to manipulate character variables.

```
data mailing_list;
  set patients;
  if substr(pat_id, length(pat_id))='1' then
    title='Mr. ';
  else if substr(pat_id, length(pat_id))='2' then
    title='Ms. ';
  f_name_mi = scan(name, 2, ',');
  last_name = scan(name, 1, ',');
  full_name = title!!trim(f_name_mi)!!' '!!
    last_name;
run;
```

In this example,

- the **LENGTH** function finds the length of the character string stored in the PAT_ID variable. It returns a number, in this case, a three or a four.
- The **SUBSTR** function has three arguments and creates a substring from its first argument, in this case, PAT_ID. The second argument marks the beginning of the substring that will be created from the first argument. In this case, the substring starts on the last character of PAT_ID. The third argument is optional, and it indicates how long the substring will be. The default is to go to the end of argument 1. In other words, the substring is going to start with the last character of argument 1 and go for a length of one. It will pick the last character from PAT_ID whether it has three or four digits. The value returned by the SUBSTR function is NOT assigned to any variable, but is used to compare with the quoted value, a '1' or a '2'. If it is a '1' then the value of TITLE will be 'Mr. ', if it is a '2', then the value of TITLE will be 'Ms. '.
- The **SCAN** function scans a character string, looking for the nth 'word' separated by a delimiter. A word is a text string between two delimiters. The SCAN function has three arguments. The first is the text to be scanned, the second is the word, and the third specifies the delimiter. In this DATA step the SCAN function is first used to create the variable F_NAME_MI. It scans the variable NAME, looking for the second word separated by a comma. The SCAN function is next used to create the variable LAST_NAME. It scans the variable NAME

looking for the first word separated by a comma.

- The **CONCATENATION** operator (!!) is used to create FULL_NAME. It has the effect of 'gluing' together character strings. First, TITLE is glued to the **TRIM** of F_NAME_MI. The TRIM function trims all trailing blanks from the end of a string. Next, a single blank is 'glued' to FULL_NAME. This separates F_NAME_MI from LAST_NAME by a single blank. Finally, LAST_NAME is joined to FULL_NAME.

Variable Conversion

In many applications we must convert one data type to another. The first example illustrates the conversion from character values to numeric values. Recall the **CHOLESTEROL** data set.

Pat_id	date1	chol1	date2	chol2	date3	chol3
1201	010600	211.1	070200	201.4	112100	191.1
1212	010400	181.7	063000	188.0	120100	191.8
1222	020100	194.8	071100	195.3	121100	196.0
1261	011800	199.7	062800	201.1	120600	211.1
...						

One of the clinical physicians wants to examine the length of time (in days) between testing. When the PATIENTS data set was created, all the date variables were created as character variables. They need to be converted to numeric so that the number of days between DATE1 and DATE2 can be calculated. The **INPUT** function is used to do such a conversion.

The INPUT function has 2 arguments. The first is the variable to be converted, the second is an informat that states HOW the variable will be converted.

```
data numeric_dates;
  set cholesterol;
  drop chol1 - chol3 date3;
  num_date1=input(date1, mmddyy6.);
  num_date2=input(date2, mmddyy6.);
  span1 = num_date2 - num_date1;
run;
```

In this example, the **INPUT** function first reads the DATE1 variable with the MMDDYY6. informat and stored the numeric results in the variable NUM_DATE1. NUM_DATE2 is

created in a similar manner. View the **NUMERIC_DATES** data set below:

Pat_id	date1	date2	num_ date1	num_ date2	span
1201	010600	070200	14615	14793	178
1212	010400	063000	14613	14791	178
1222	020100	071100	14641	14802	161
1261	011800	062800	14627	14789	162
...					

The next example illustrates numeric to character conversion. Recall the **COLESTEROL2** data set.

Pat_id	date	chol_reading
1201	010600	211.1
1201	070200	201.4
1201	112100	191.1
1212	010400	181.7
1212	063000	188.0
1212	120100	191.8
1222	020100	194.8
1222	071100	195.3
1222	121100	196.0
...		

When this data set was created, **PAT_ID** was created as a numeric variable. Remember its values are either a 3 or 4 digit number. As a numeric variable in a SAS data set, it takes up eight bytes of storage. If we convert it to a character variable, it takes up only four bytes of storage. The following DATA step illustrates how to do this conversion.

```
data character_id(drop=x_id);
  set cholesterol2(rename=(pat_id=x_id));
  pat_id = put(x_id, $4.);
run;
```

The **PAT_ID** variable in the **CHOLESTEROL2** data set is renamed to **X_ID**. The **PUT** function reads the value of **X_ID** and then writes it to a new **PAT_ID** using the \$4. format. The **PUT** function always creates a character variable. The new data set, **CHARACTER_ID** looks just like **CHOLESTEROL2**. The only difference is that **PAT_ID** is a character variable in **CHARACTER_ID** while **PAT_ID** is a numerical variable in **COLESTEROL2**.

Data Transformation

Sometimes the data that you are working with is in the shape of the **CHOLESTEROL** data set (one observation per **PAT_ID**).

Pat_id	date1	chol1	date2	chol2	date3	chol3
1201	010600	211.1	070200	201.4	112100	191.1
1212	010400	181.7	063000	188.0	120100	191.8
1222	020100	194.8	071100	195.3	121100	196.0
1261	011800	199.7	062800	201.1	120600	211.1
...						

The application that you are developing needs data in the 'shape' of the **CHOLESTEROL2** data set (one observation per cholesterol reading).

Pat_id	date	chol_reading
1201	010600	211.1
1201	070200	201.4
1201	112100	191.1
1212	010400	181.7
1212	063000	188.0
1212	120100	191.8
1222	020100	194.8
1222	071100	195.3
1222	121100	196.0
...		

The following DATA step will transpose the data from one observation per **PAT_ID** to one observation per cholesterol reading.

```
data transpose1(keep=pat_id date chol_reading);
  set cholesterol;
  array dates {3} date1 date2 date3;
  array chols {3} chol1 chol2 chol3;
  do i = 1 to 3;
    date = dates{i};
    chol_reading = chols{i};
    output;
  end;
run;
```

In this example, the array statements collect three variables under each array. When the **DO** loop iterates the first time, the value of the first variable in the **DATES** array (the variable **DATE1**) is assigned to the variable **DATE**. Next, the value of the first variable in the **CHOLS** array (the variable **CHOL1**) is assigned to the variable **CHOL_READING**. Then the output statement writes out the first observation.

When the DO loop iterates the second time, the value of the second variable in the DATES array (the variable DATE2) is assigned to the variable DATE. Next, the value of the second variable in the CHOLS array (the variable CHOL2) is assigned to the variable CHOL_READING. Then the output statement writes out the second observation.

When the DO loop iterates the third time, the value of the third variable in the DATES array (the variable DATE3) is assigned to the variable DATE. Next, the value of the third variable in the CHOLS array (the variable CHOL3) is assigned to the variable CHOL_READING. Then the output statement writes out the third observation. Notice that the OUTPUT statement executes three times for every single time the SET statement executes.

To transpose the data from one observation per cholesterol reading to one observation per PAT_ID use the following DATA step.

```
data transpose2(keep=pat_id date chol_reading);  
  array dates {3} $ date1 date2 date3;  
  array chols {3} chol1 chol2 chol3;  
  do i = 1 to 3;  
    set cholesterol2;  
    dates{i} = date;  
    chols{i} = chol_reading;  
  end;  
  output;  
run;
```

First of all, notice in this example that the SET statement executes three times for every single time the OUTPUT statement executes. In other words, for every three observations read, only one is written.

Next, notice the '\$' in the ARRAY statement. This is an example of an ARRAY statement that creates the variables DATE1, DATE2, and DATE3. In this DATA step, the ARRAY statement appears before the SET statement, meaning that without the '\$' on the ARRAY statement, the date variables would be created as numeric. In the CHOLESTEROL2 data set, all the date variables are character. The '\$' just keeps SAS from doing an automatic numeric to character conversion.

Conclusion

The SAS System is very robust and has well over a hundred functions and other ways to manipulate data. Only a few of them are presented in this paper. For a full discussion of all the functions available, a user would have to consult the literature published by SAS Institute, Inc. The best place to start would be the SAS Language Reference Guide, or the on-line documentation available for Version 8 of the SAS System.

The author can be reached at the following location:

Ben Cochran
The Bedford Group
3216 Bedford Avenue
Raleigh, NC 27607
919.831.1191

bedford.group@mindspring.com

The Bedford Group is a SAS Institute Quality Partner and SAS Certified Professional for Version 8. They specialize in SAS consulting and training.

Trademark Information

SAS is a registered trademark of SAS Institute Inc. in the USA and other countries. The ® indicates USA registration.