

SAS* Tutorial Session - An Introduction to SAS Functions

Dr. Ronald Cody - Robert Wood Johnson Medical School

A. Overview

Most of you have probably already used SAS functions to perform a variety of tasks, such as a LOG function to transform variables and various DATE functions to convert a date into an internal SAS date. We will see in this tutorial that the SAS programming language has a very rich assortment of functions that can greatly simplify some very complex programming problems. Take a moment to browse through this paper to see what SAS functions can do for you.

B. Arithmetic and Mathematical Functions

The functions you are probably most familiar with are ones which perform arithmetic or mathematical calculations. Remember that all SAS functions are recognized as such because the function name is always followed by a set of parentheses. This way, the program can always differentiate between a variable name and a function. Here is a short program which computes a new variable, called LOGLOS which is the natural log of LOS (length of stay). This is a common way to "pull in the tail" of a distribution skewed to the right. We have:

```
DATA FUNC_EG;  
INPUT ID SEX $ LOS HEIGHT WEIGHT;  
LOGLOS = LOG(LOS);  
CARDS;  
etc.
```

The new variable (LOGLOS) will be in the data set FUNC_EG and its values will be the natural (base e) log of LOS. Note that a zero value for LOS will result in a missing value for LOGLOS. When zero's are possible values and you still want a log transformation, it is common to add a small number (usually .5 or 1) to the variable before taking the log.

We will now list some of the more common arithmetic and mathematical functions and their purpose:

Function Name	Action
LOG	Base e log
LOG10	Base 10 log
SIN	Sine of the argument (in radians)
COS	Cosine
TAN	Tangent
ARSIN	Arcsine (inverse sine) of the argument. Result is in radians
ARCOS	Arccosine
ARTAN	Arctangent
FLOOR	Drops off the fractional part of a number
SQRT	Square root

Some functions take more than one argument. For example, the ROUND function has two arguments, separated by commas. The first argument is the number to be rounded and the second argument indicates the roundoff unit. Here are some examples:

```
ROUND (X,1)    Round X to the nearest whole number
ROUND (X,.1)   Round X to the nearest tenth
ROUND (X,100)  Round X to the nearest hundred
```

Another function that takes two arguments is the MOD function. MOD(x,m) will return the value of x modulus m. For example, MOD(5,3) (read this 5 mod 3) will return a 2. At first glance, this seems like an esoteric function of little use in everyday programming. Not true. Here is a program to select every third record of a file:

```
DATA MODEG;
SET OLD;
I + 1;
IF MOD (I,3) = 1;
DROP I;
RUN;
```

Other functions operate on a list of arguments. A good example of this is the MEAN function. If we have a series of variables (say X1 to X5) for each subject and we want the mean of these 5 numbers, we write:

```
MEAN_X = MEAN (OF X1-X5):
```

We may use any variable list following the word OF. An important difference between the MEAN function and the alternative expression:

```
MEAN_X = (X1 + X2 + X3 + X4 + X5)/5;
```

is that the MEAN function returns the mean of the nonmissing values. Thus, if we had a missing value for X5, the function would return the mean of X1, X2, X3, and X4. Our equation for the mean above, would return a missing value if any of the X values were missing.

The SUM, STD, and STDERR functions work the same way as the MEAN function except that a sum, standard deviation, or a standard error is computed instead of a mean.

Two very useful functions are N and NMISS. They return, as you would expect, the number of nonmissing (N) or the number of missing (NMISS) values in a list of variables. Suppose we have recorded 100 scores for each subject and want to compute the mean of these 100 scores. We will allow each subject to be missing 25 or fewer scores and still be included in the calculation. Without the N function, we would have to do a bit of programming. Using the N function, the computation is much simpler:

```

DATA EASYWAY;
INPUT (X1-X100) (2.);
IF N(OF X1-X100) GE 75 THEN
    AVE = MEAN(OF X1-X100);
CARDS;
    etc.

```

The NMISS function is used in a similar fashion.

C. Random Number Functions

SAS has two functions, UNIFORM and RANUNI that generate uniform random numbers in the range from 0 to 1. Random number generators (more properly called pseudo random number generators) require an initial number, called a seed, which they use to calculate the first random number. From then on, each random number is used in some way to generate the next. In either of these functions, a zero seed will cause the function to use a seed derived from the time clock, thus generating a different random series each time it is used. RANUNI can also be seeded with any number of your choosing; UNIFORM requires a 5, 6, or 7 digit odd number as a seed. In either case, if you supply the seed, the function will generate the same series of random numbers each time. A simple example follows in which we use a uniform random number to put a group of subjects in random order:

```

DATA SHUFFLE;
INPUT NAME : $20.;
X = RANUNI(0);
CARDS;
CODY
SMITH
MARTIN
LAVERY
THAYER
RUN;
PROC SORT;
    BY X;
RUN;
PROC PRINT;
    TITLE 'Names in Random Order';
    VAR NAME;
RUN;

```

To generate a series of random numbers from n to m, we need to scale our 0 to 1 random numbers accordingly. To generate a series of random numbers from 1 to 100, we could write:

```
X = 1 + 99 * RANUNI(0);
```

For purposes of statistical modeling, we might also want a series of random numbers chosen from a normal distribution (mean=0, variance=1). The RANNOR function will generate such variables. The allowable seeds for RANNOR follow the same rules as for RANUNI.

D. Time and Date Functions

There are several extremely useful date functions. One, MDY (month, day, year) will convert a month, day, and year value into a SAS date variable. Suppose for example, that we want to know a subject's age as of July 15, 1990 and we know his date of birth. We could use the MDY function to compute the age thus:

```
AGE = (MDY(7,15,90) - DOB)/365.25;
```

Although a more efficient method would be to use a SAS date variable as shown below:

```
AGE = ('15JUL90'D - DOB)/365.25;
```

Another possible use of the MDY function would be when date information is not recorded in one of the standard methods for which SAS has a date format. As long as we can read the month, day, and year into variables, we can then use the MDY function to compute the date. For example:

```
INPUT ID 1-3 MONTH 4-5 DAY 10-11 YEAR 79-80;  
DATE = MDY(MONTH, DAY, YEAR);  
DROP MONTH DAY YEAR;  
FORMAT DATE MMDDYY8.;  
etc.
```

There are several date functions which extract information from a SAS date. For example, the YEAR function returns a four digit year from a SAS date. The MONTH function returns a number from 1 to 12 which represents the month for a given date. There are two functions which return day information. The DAY function returns the day of the month (i.e., a number from 1 to 31) and the WEEKDAY function returns the day of the week (a number from 1 to 7, 1 being Sunday). As an example, suppose we want to see distributions by month and day of the week for hospital admissions. The variable ADMIT is a SAS date variable:

```
PROC FORMAT;  
  VALUE DAYWK 1='SUN' 2='MON' 3='TUE' 4='WED' 5='THU' 6='FRI' 7='SAT';  
  VALUE MONTH 1='JAN' 2='FEB' 3='MAR' 4='APR' 5='MAY' 6='JUN' 7='JUL'  
    8='AUG' 9='SEP' 10='OCT' 11='NOV' 12='DEC';  
DATA HOSP;  
INPUT @1 ADMIT MMDDYY6. etc. ;  
DAY = WEEKDAY(ADMIT);  
MONTH = MONTH(ADMIT);  
FORMAT ADMIT MMDDYY8. DAY DAYWK. MONTH MONTH.;  
CARDS;  
(data lines)  
PROC CHART;  
  VBAR DAY / DISCRETE;  
  VBAR MONTH / DISCRETE;  
RUN;
```

Look for a shortcut method for producing the day of the week or month name in the discussion of the PUT function later in this chapter.

Besides working with date values, SAS has a corresponding set of functions to work with times. For example, we can read a time in hh:mm:ss (hours, minutes, seconds) format using the time8. format. We can then extract hour, minute, or second information from the time variable using the HOUR, MINUTE, or SECOND functions, just the way we used the YEAR, MONTH, and WEEKDAY functions above.

Before we leave the date and time functions, let's discuss two very useful functions, INTCK and INTNX. They may save you pages of SAS coding. INTCK returns the number of intervals between any two dates. Valid, interval choices are: DAY, WEEK, MONTH, QTR, YEAR, HOUR, MINUTE, and SECOND. The syntax of the INTCK function is:

```
INTCK (interval, start, end);
```

Where interval is one of the choices above, placed in single quotes, start is the starting date, and end is the ending date. As an example, suppose we want to know how many quarters, our employees have worked. If START is the SAS variable that holds the starting date, the number of quarters worked would be:

```
NUM_QTR = INTCK('QTR',START,TODAY());
```

(Note: the TODAY function, which has no argument, returns today's date)

Since the algorithms used to compute the number of intervals can be confusing, we recommend reading the section on SAS functions in the appropriate SAS manual.

The INTNX function can be thought of as the inverse of the INTCK function; it returns the date, given an interval, starting date, and the number of intervals elapsed. Suppose we know the date of hire and want to compute the date representing the start of the third quarter. You would use:

```
DATE3RD = INTNX ('QTR',HIRE,3);  
FORMAT DATE3RD MMDDYY8.;
```

If HIRE were 01/01/90, 01/05/90, or 03/30/90, the value of DATE3RD would be 10/01/90. If a person were hired on April 1, 1990, his third quarter date would be 01/01/91.

E. The INPUT and PUT Functions: Converting Numerics to Character and Character to Numeric variables

While the INPUT and PUT functions have many uses, one common application is to convert between numeric and character values.

The PUT function uses the formatted value of a variable to create a new variable. For example, suppose we have recorded the AGE of each subject. We also have a format which groups the ages into four groups:

```

PROC FORMAT;
  VALUE AGEGRP LOW-20='1' 21-40='2' 41-60='3' 61-HIGH='4';
RUN;
DATA PUTEQ;
  INPUT AGE @@;
  AGE4 = PUT (AGE,AGEGRP.);
CARDS;
5 10 15 20 25 30 66 68 99
RUN;

```

In this example, the variable AGE4 will be a character variable with values of '1', '2', '3', or '4'. As another example, suppose we want a variable to contain the three letter day of the week abbreviations (MON, TUE, etc.). One of the SAS built-in formats is WEEKDATEn. returns values such as: WEDNESDAY, SEPTEMBER 12, 1990 (if we use WEEKDATE29.). The format WEEKDATE3. will be the first three letters of the day name (SUN, MON, etc.). To create our character day variable, we can use the PUT function:

```
DAYNAME = PUT (DATE,WEEKDATE3.);
```

There are some really useful tricks that can be accomplished using the PUT function. An example that we used just the other day comes to mind. We had one file that had social security numbers as 9 digit numerics. Another file had the social security numbers coded as 11 digit character strings (123-45-6789). Our job was to merge the two files based on social security number. There are many ways to solve this problem, either pulling out the three numbers between the dashes and recombining them to form a numeric, or converting the 9 digit number to a character string and placing the dashes in the proper places using the appropriate string functions. By far the easiest method is to use the fact that SAS has a built in function, SSN11. which formats 9 digit numerics to 123-45-6789 style social security numbers. Therefore, using the PUT function, we can create a character variable in the form 123-45-6789 like this:

```
SS = PUT (ID,SSN11.);
```

In general, to convert a numeric variable to a character variable, we can use the PUT function, with the appropriate format. If we have a file where group is a numeric variable and we want a character variable instead, we can write:

```
GROUPCHR = PUT (GROUP,1.);
```

We use the INPUT function in a similar manner except that we can "reread" a variable according to a new format. The most common use of the function is to convert character values into numeric values. We will show you a simple example here. In this example, we will read either a group code (character) or a score (numeric). Since we don't know if we will be reading a character or a number, we will read every value as a character and test if it is a valid group code. If not, we will assume it is a score and use the INPUT function to convert the character variable to numeric. Here is the code:

```

DATA FREEFORM;
INPUT TEST $ @@;
RETAIN GROUP;
IF TEST = 'A' OR TEST='B' THEN DO;
    GROUP = TEST;
    DELETE;
    RETURN;
END;
ELSE SCORE = INPUT (TEST,5.);
DROP TEST;
CARDS;
A 45 55 B 87 A 44 23 B 88 99
RUN;
PROC PRINT;
RUN;

```

To help make this example clearer, the data set formed by running this program is shown below:

OBS	GROUP	SCORE
1	A	45
2	A	55
3	B	87
4	A	44
5	A	23
6	B	88
7	B	99

As you can see, the INPUT function opens up a very flexible way of reading data.

F. String Functions

OK you may ask, what's a string? Well, computer people have their own jargon and a string, to a computer programmer, is a string of characters. Thus, ABCDE is a string as is 12345 (read as character data). We will only touch on some of the major string functions here--there are lots of them.

First, it's useful to extract a piece of a string. Suppose we had character ID numbers in a SAS system file. The numbers are 8 characters long and the last two characters of the ID number represent a two digit state code. We want to create a new variable, STATE which contains these two digits. The SUBSTR (pronounced substring) function will do this. The syntax is:

```
char_var = SUBSTR (C, S, L);
```

Where C is a character string, S is the starting position, and L is the length of the substring.

So, if we want to extract the last two digits (positions 7 and 8) we can use:

```
STATE = SUBSTR (ID,7,2);
```

Another very commonly used character function is UPCASE. As the name implies, this function converts the argument to upper case. This can really come in handy when a questionnaire or other source of data was entered in both upper and lower case (by accident). When this happens, we can use the UPCASE function to convert all the strings to upper case. Below is an example of how this could be used:

```
DATA SURVEY;  
INPUT ID 1-3 @5 (QUES1-QUES20)($1.);  
ARRAY QUES[20] QUES1-QUES20;  
DO I = 1 TO 20;  
    QUES[I] = UPCASE(QUES[I]);  
END;  
DROP I;  
etc.
```

The VERIFY function can be used to test if any element of one string is located anywhere in another. A common use for this function is in data verification. Although the VERIFY function is more general, we will show a partial definition below:

```
VERIFY (char_variable, verify_string);
```

This function will return the position in the char_variable that is not present in the verify_string. If all the characters of the char_variable are present in the verify_string, then a 0 is returned. Here are some examples:

```
SCORE='A ' VERIF='ABCDE'  
VERIFY(SCORE,VERIF)=0
```

```
SCORE='X ' VERIF='ABCDE'  
VERIFY(SCORE,VERIF)=1
```

```
SCORE='ABXD' VERIF='ABCDE'  
VERIFY(SCORE,VERIF)=3
```

Suppose that valid values for X were A,B,C,D, or E. A quick way to check for invalid data would be:

```
CHECK = 'ABCDE';  
IF VERIFY(X,CHECK) NE 0 THEN  
    PUT 'ERROR IN RECORD ' _N_ X= ;
```

Notice the blank in the character string assigned to CHECK. If X were a one byte character variable, the blank in CHECK would be unnecessary. If X were longer than one byte, the value would be "left adjusted" and padded on the right with blanks. The VERIFY function, would then return the position of the first blank in the X string if we did not include a blank in the CHECK string.

While we are discussing strings, we should mention the concatenation operation. Although this is not a function, it is a useful string operation and this seems as good as anywhere to tell you about it! In computer jargon, concatenate means to join. So, if we concatenate the string 'ABC' with the string 'XYZ' the result is 'ABCXYZ'.

Pretty clever eh? Things can get a bit sticky if we forget that SAS character variables are padded on the right with blanks to fill out the predefined length of the variable. If this is the case, we can use the TRIM function to remove trailing blanks before we concatenate the strings. The concatenation operator is || (two vertical bars). Suppose we had social security numbers in a file and, instead of the usual - separators, the digit groups were separated by colons, 123:45:6789 for example. One way to read this string and convert it into the more common format would be:

```
DATA CONVERT;
INPUT PART1 $ 1-3 PART2 $ 5-6 PART3 $ 8-11;
SS = PART1 || '-' || PART2 || '-' || PART3;
KEEP SS;
CARDS;
123:45:6789
etc.
```

For those of us who are compulsive programmers, the TRANSLATE function discussed next would be a better way to solve this problem.

The last string function we will discuss is TRANSLATE. This very useful function is used to, as the name implies, translate or convert one set of characters into another. This function may save you writing lots of IF statements. Suppose we have one data set where responses to a questionnaire were coded as A,B,C,D, and E. Another set of data from the same questionnaire was coded 1,2,3,4, and 5 by the data entry person. Assuming that we read the responses as characters, we could write five IF statements to convert the numbers to letters, or use the TRANSLATE function.

The syntax for TRANSLATE is:

```
TRANSLATE (c, to_str, from_str);
```

Where c is the character variable to convert, to_str is the string which we will convert to, and from_str is the string we want to convert from. For example, to translate 1,2,3,4,5 (character data) to A,B,C,D,E we would write:

```
TRANSLATE (variable, 'ABCDE', '12345');
```

In the example below, assume that we have variable names QUES1-QUES50 (\$1. format) and want to convert the 1-5 to A-E:

```
DATA TRANS;
ARRAY QUES[*] $ QUES1-QUES50;
INPUT (QUES1-QUES50) ($1.);
DO I = 1 TO 50;
    QUES[I] = TRANSLATE (QUES[I], 'ABCDE', '12345');
END;
DROP I;
etc.
```

The solution to the social security problem above is:

```
INPUT SS $ 1-11;
SS = TRANSLATE (SS, '-', ':');
```

G. The LAG Function

A lagged value is one from an earlier time. In SAS, we may want to compare a data value from the current observation with a value in the previous observation. We may also want to look back several observations. Without the LAG function, this is a difficult task—without it, it's simple. If we have a variable, X in our data set, the value LAG(X) is the value of X from the previous observation. The value LAGn(X) where n is a number, is the value from the nth previous observation. A common application of the LAG function is to take differences between observations. Suppose each subject was measured twice and each measurement was entered as a separate observation. We want to take the value of X at time 1 and subtract it from the value of X at time 2. We proceed as follows:

Data set ORIG looks like this:

SUBJ	TIME	X
1	1	4
1	2	6
2	1	7
2	2	2
etc.		

To subtract the X at time 1 from the X at time 2, we write:

```
*NOTE: DATA SET ORIG IS SORTED BY SUBJ;  
DATA LAGEG;  
SET ORIG;  
DIFF = X - LAG(X);  
IF TIME = 2 THEN OUTPUT;  
RUN;
```

You could shorten this program even further by using the DIFn function which returns the difference between a value from the current observation and the nth previous observation. The calculation above would be:

```
DIFF = DIF(X);
```

What we have shown you here is only a brief review of SAS functions. Spend some time browsing through the chapter on SAS functions in the SAS Language Guide, from the SAS Institute, now that you have been introduced to the power of SAS functions.

Author's Address: Dr. Ronald Cody
Department of Environmental and Community Medicine
Robert Wood Johnson Medical School
675 Hoes Lane
Piscataway, NJ 08854

Voice Phone: (908) 463-4490

*SAS is a registered trademark of the SAS Institute, Inc., Cary NC.