A Survey of Some of the Most Useful SAS® Functions

Ron Cody, Camp Verde, Texas

ABSTRACT

SAS™ Functions provide amazing power to your DATA step programming. Some of these functions are essential—some of them save you writing volumes of unnecessary code. This paper covers some of the most useful SAS functions. Some of these functions may be new to you and they will change the way you program and approach common programming tasks.

INTRODUCTION

The majority of the functions described in this paper work with character data. There are functions that search for strings, others that can find and replace strings or join strings together. Still others that can measure the spelling distance between two strings (useful for "fuzzy" matching). Some of the newest and most amazing functions are not functions at all, but call routines. Did you know that you can sort values **within** an observation? Did you know that not only can you identify the largest or smallest value in a list of variables, but you can identify the second or third or n[th] largest of smallest value? If this introduction has caught your attention, read on!

HOW SAS STORES CHARACTER VALUES

Before we discuss functions that operate on character values, it is important to be aware of how SAS stores character values. To help that discussion, you need to understand two important character functions: LENGTHN and LENGTHC.

LENGTHN AND LENGTHC

These two functions return information about the length of character values. LENGTHN returns the length of its argument not counting trailing blanks. LENGTHC returns the storage length of a character variable. You may be familiar with an older SAS function called LENGTH. LENGTH and LENGTHN return the same value except when the argument is a missing value. In that case, LENGTH returns a 1 and LENGTHN returns a 0. There are several new functions that look like old functions except that there is an "n" added to the end of the name. The "n" stands for "null string." In SAS 9, the concept of a string of zero length was introduced. In most cases, if you see a new function (such as TRIMN) that looks like one you already know (TRIM), use the newer one ending with the "n".

Take a look at the following program:

Program 1

```
data chars1;
   length String $ 7;
   String = 'abc';
   Storage_length = Lengthc(string);
   Length = lengthn(String);
   Display = ":" || String || ":";
   put Storage_length= /
       Length= /
       Display=;
run;
```

Figure 1: Output from Program 1

```
Storage_length=7
Length=3
Display=:abc    :
```

Remember, the storage length of a SAS character variable is set at compile time. Since the LENGTH statement comes before the assignment statement for String, SAS assigns a length of 7 for String. The LENGTHN function returns a 3 since this is the length of String, not counting the trailing blanks. Finally, by concatenating a colon on each side of String, it is easy to see that this value contains 4 trailing blanks.

If you move the LENGTH statement further down in the program like this:

Program 2

```
data chars2;
   String = 'abc';
   length String $ 7;
   Storage_length = lengthc(String);
   Length = lengthn(String);
   Display = ":" || String || ":";
   put Storage_length= /
       Length= /
       Display=;
run;
```

You obtain the following:

Figure 2: Output from Program 2

```
Storage_length=3
Length=3
Display=:abc:
```

Notice that the LENGTH statement is ignored. Since String = 'abc' appears before the LENGTH statement, the length of String has already been set. As a good rule-of-thumb, run PROC CONTENTS on all of your data sets and check the storage length of all your character variables. Don't be surprised if you see some character variables with lengths of 200, the default length for many of the SAS character functions—that is, the length that SAS assigns to a variable if you do not implicitly indicate the length in a LENGTH statement or some other way.

THE MISSING FUNCTION AND THE CALL MISSING ROUTINE

In the "old days" you could check for a missing value in a DATA step like this:

```
*Old way;
if Age = . then . . .
If Char = ' ' then . . .


*New way;
if missing(Age) then . . .
if missing(Char) then . . .
```

The argument to the missing function can either be character or numeric and the function returns a value of true if the argument is a missing value and false otherwise. I highly recommend that you use this function in any program where you need to test for a missing value. You will find that the programs read so much better.

If you need to set one or more character or numeric variables to a missing value, you can do it the old way like this:

```
array x[10] x1-x10;
array chars[5] a b c;
do i = 1 to 10;
   x[i] = .;
end;
do I = 1 to 3;
   chars[i] = ' ';
end;
drop i;
```

or you can save yourself a lot of effort by using the call missing routine like this:

```
call missing(of x1-x10, a, b, c);
```

THE INPUT FUNCTION

This is one function that I had difficulty understanding when I was learning SAS (so many years ago). An easy way to think about the INPUT function is to ask yourself "What does an INPUT statement do?" It takes a text value, usually from a file and reads it according to a supplied INFORMAT. Well, the INPUT function does a similar thing. It takes a text value (the first argument to the function) and "reads" it as if it were reading data from a file, according to the INFORMAT that you supply as the second argument. Perhaps the next program will make this clear:

Program 3

```
data _null_;
    c_date = "9/15/2004";
    c_num = "123";
    Sas_Date = input(c_date,mmddyy10.);
    Number = input(c_num,10.);
    put SAS_Date= Number=;
run;
```

You have two character variables in this program (c_date and c_num). By using the INPUT function you created a true SAS date (numeric) on the first value and performed a character to numeric conversion on the second. Notice that the informat used to convert c_num is 10. This is not a problem. Unlike reading text from a file, the INFORMAT you supply cannot read past the end of the character value. After you run this program the value of SAS_Date and Number are:

Figure 3: Output from Program 3

```
SAS_Date = 16329
Number = 123
```

THE PUT FUNCTION

The companion function to the INPUT function is the PUT function. Again, think of what a PUT statement does—it takes a SAS value (character or numeric), applies a FORMAT and writes out the result, usually to a file. The PUT function takes a SAS value (the first argument) and "writes" out the formatted result (using format supplied as the second argument) to a SAS character variable. One use of the PUT function is for numeric to character conversion. Another use is to use user-written formats to create new variables. Here is an example:

Program 4

```
data _null_;
    SAS_Date = 1;
    Number = 1234;
    SS_num = 123456789;
    Char_Date = put(SAS_Date,mmddyy10.);
    Money = put(Number,dollar8.2);
    SS_char = put(ss,ssn.);
    put Char_date= Money= SS_char=;
run;
```

This example takes three numeric values (a SAS date, a number, and a social security number) and creates three character variables. After you run this program, the values of the character variables are:

Figure 4: Output from Program 4

```
Char_Date = "1/2/1960"
Money = "$1,234.00"
SS_Char = "123-45-6789"
```

The next program shows how you can use a format to group ages into categories. This is somewhat easier than writing a series of IF-THEN-ELSE statements. Here is the program:

Program 5

```
proc format;
```

```
        value agegrp 0-20='0 to 20'
                      21-40='21 to 40'
                      41-high='41+';
run;
data PutEx;
    input Age @@;
    AgeGroup = put(Age,agegrp.);
datalines;
15 25 60
;
```

The new variable AgeGroup is now a character variable with the formatted values for Age. The storage length of this new variable is the longest formatted value. Below, you can see the values of Age and AgeGroup.

Figure 5: Output from Program 5

```
Age     AgeGroup
 15     0 to 20
 25     21 to 40
 60     41+
```

FIND AND FINDC FUNCTIONS

The FIND function takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument. If the substring is found, the function returns its position. If it is not found, the function returns a 0. There are two optional arguments to the FIND function—modifiers and starting position. The most useful modifier is the 'i' modifier. This says to ignore case. The starting position defines at what position the search begins. If the starting value is a negative number, the search starts at the absolute value of the starting position and the search proceeds from right to left. By the way, you can enter these two optional arguments in any order! Why is this possible? Modifiers are always character values and starting positions are always numeric values. How clever! If you only want either a modifier or a starting position, enter that as the third argument. Here is an example:

Program 6

```
data locate;
    input String $10.;
    First = find(String,'xyz','i');
    First_c = findc(String,'xyz','i');
    /* i means ignore case */
datalines;
abczyx1xyz
1234567890
abcz1y2x39
XYZabcxyz
;
```

This example uses the 'i' modifier for both functions. By using this modifier, you save yourself the trouble of having to change the case of one or more strings before you start your search.

Figure 6: Output from Program 6

```
String          First     First_c
abczyx1xyz         8           4
1234567890         0           0
abcx1y2z39         0           4
XYZabcxyz          1           1
```

In the first observation, the substring 'xyz' is not found until the eighth position in String. Because the FINDC function is looking for an 'x' or a 'y' or a 'z', it returns a 4 in the first observation because of the 'z' in the fourth position. Notice

4

that when there are no matches as in observation 2, the functions return a 0.

THE COMPRESS FUNCTION

This is certainly one of my favorite SAS functions. It is also the only function I know that changed from version 8 to 9. How is that possible? Don't programs written in version 8 have to work exactly the same way in version 9? The answer is "yes" but the change in version 9 was to add an optional third argument to the function.

The three arguments to the COMPRESS function are:

```
Compress(String, characters-to-remove, optional-modifiers)
```

where

String is the value from which you want to remove characters (unless you use a 'k' modifier)

Characters-to-remove is a list of characters you want to remove from *String*. If you only give the COMPRESS function one argument, it removed spaces from *String*,

Optional-modifiers allow you to specify character classes such as:

- a   upper- and lowercase letters
- d   numerals (digits)
- i   ignores case
- k   keeps listed characters instead of removing them
- s   space (blank, tabs, lf, cr) to the list
- p   punctuation

I believe that the 'k' (keep) modifier is what makes this function really powerful. The 'k' option tells the function that the list of characters to remove or the other modifiers are now a list of characters that you want to keep—throw away all the others. It is usually better to specify what you want to keep than what you want to throw away. For example, if you use the two modifiers 'k' and 'd', the function will keep all the digits in String and throw away all the rest. This is especially useful when your string contains non-printable characters.

Here is an example:

Program 7

```
data phone;
   input Phone $15.;
   Phone1 = compress(Phone);
   Phone2 = compress(Phone,'(-) ');
   Phone3 = compress(Phone,,'kd');
datalines;
(908)235-4490
(201) 555-77 99
```

The COMPRESS function will remove spaces from Phone1, because you only used one argument. For Phone2, you specified open and closed parentheses, a dash and a space. And for Phone3, you specified the two modifiers 'k' and 'd'. Notice the two commas. They are necessary to tell SAS that the 'kd' are modifiers (third argument) and not a list of character to remove (second argument). Notice in the listing below, that Phone2 and Phone3 are the same. However, had there been any extraneous characters in Phone, Phone2 would still contain those characters.

Figure 7: Output from Program 7

```
     Phone          Phone1         Phone2        Phone3
(908)235-4490    (908)235-4490   9082354490    9082354490
(201) 555-77 99 (201)555-7799    2015557799    2015557799
```

Here is another very useful example showing how the COMPRESS function can be used to extract the digits from values that contain other non-digit characters, such as units. Take a look:

Program 8

```
data Units;
   input @1 Wt $10.;
   Wt_Lbs =
       input(compress(Wt,,'kd'),8.);
   if findc(Wt,'K','i') then
       Wt_Lbs = 2.2*Wt_Lbs;
datalines;
155lbs
90Kgs.
;
```

You see that the input data contains units such as lbs. or Kgs. This is a fairly common problem. Using the COMPRESS function makes for a very simple and elegant solution. You start by keeping only the digits in the original value and you use the INPUT function to do the character to numeric conversion. Now you need to test if the original value contained an upper- or lowercase 'k'. If so, you need to convert kilograms to pounds. The FINDC function, with the 'i' modifier makes this a snap.

Figure 8: Output from Program 8

```
Listing of Data Set Units
  Wt      Wt_Lbs
155lbs      155
90Kgs.      198
```

THE SUBSTR FUNCTION

If you need to extract a substring from a string, the SUBSTR function is the way to go. By the way, there is also a SUBSTRN function that works very much like the SUBSTR function with a few additional features. I don't believe these features are needed very often, so I chose to describe the slightly simpler SUBSTR function for this paper.

The first argument to this function is the input string. The second argument is the starting position of where you want to extract your substring, and the third, optional argument, is the length of the substring. If you omit the third argument, the function extracts a substring from the input string up to the last non-blank character. That is, it ignores the trailing blanks in the input string (this feature can be quite useful).

Before we go on to the example, it is very important to understand something called "default length." For example, in the next program, if you did not include a LENGTH statement, SAS would still need to assign a length to State. For this function, the default length is equal to the length of the first argument to the function. This makes sense since you cannot extract a substring from a string that is longer than the string itself. Many other SAS character functions have a default length of 200. Be sure to know which functions have this property or include a LENGTH statement for character variables that you create with assignment statements in a DATA step. It is just fine to include a LENGTH statement when it isn't needed. It does no harm.

Here is a simple example using the SUBSTR function:

Program 9

```
data pieces_parts;
   input Id $9.;
   length State $ 2;
   State = substr(Id,3,2);
   Num = input(substr(Id,5),4.);
datalines;
XYNY123
XYNJ1234
;
```

Here you want to extract the state code (starting in position 3 for a length of 2) and the digit part of the ID starting in position 5. Notice that you omit the third argument in the digit extraction. This is useful since some digits are 3 characters long and some are 4. You use the INPUT function to perform the character to numeric conversion in this example.

Figure 9: Output from Program 9

```
Listing of Data Set PIECES_PARTS
   Id        State     Num
XYNY123       NY       123
XYNJ1234      NJ       1234
```

THE SUBSTR FUNCTION USED ON THE LEFT-HAND SIDE OF THE EQUAL SIGN

Way back when I was learning SAS (probably before your time), this was called the SUBSTR pseudo function. That name was too scary and SAS has renamed it the SUBSTR function used on the left-hand side of the equal sign. To my knowledge, this is the only SAS function allowed to the left of the equal sign. Here's what it does:

It allows you to replace characters in an existing string with new characters. This sounds complicated, but you will see in the following program, that it is actually straight forward. This next program uses the SUBSTR function (on the left-hand side of the equal sign) to mask the first 5 characters in an account number. Here is the code:

Program 10

```
data bank;
   input Id Account : $9. @@;
   Account2 = Account;
   substr(Account2,1,5) = '*****';
datalines;
001 123456789 002 049384756 003 119384757
;
```

First you assign the value of Account to another variable (Account2) so that you don't destroy the original value. Next, you replace the characters in Account2, starting from position 1 for a length of 5 with five asterisks. Here is the listing:

Figure 10: Output from Program 10

```
Id    Account2
 1    *****6789
 2    *****4756
 3    *****4757
```

THE SCAN FUNCTION

You use the SCAN function to parse (take apart) a string. The first argument to the SCAN function is the string you want to parse. The second argument specifies which "word" you want to extract. The third (optional) argument is a list of delimiters. The reason I put "word' in quotes is that SAS defines a word as anything separated by a delimiter. The default list of delimiters is quite long and it is slightly different between ASCII and EBCDIC encoding. Therefore, it is probably a good idea to supply a third argument and explicitly specify your delimiters.

A very useful feature of this function is that you can use a negative value for which word you want. This causes the scan to go from right to left. This is particularly useful when you have names in the form: First, Middle, Last or just First and Last. If you use a -1 for the word, you always get the last name. Here is an example:

Program 11

```
data first_last;
   length Last_Name $ 15;
   input @1  Name  $20.;
   Last_Name = scan(Name,-1,' ');
datalines;
Jeff W. Snoker
Raymond Albert
Alfred E. Newman
Steven J. Foster
Jose Romerez
;
```

Some names contain a middle initial, some do not. By using the -1 as the second argument to the function, you always the last name.

Figure 11: Output from Program 11

```
                       Last_
Name                   Name
Jeff W. Snoker         Snoker
Raymond Albert         Albert
Alfred E. Newman       Newman
Steven J. Foster       Foster
Jose Romerez           Romerez
```

UPCASE, LOWCASE AND PROPCASE FUNCTIONS

These three functions change the case of the argument. UPCASE and LOWCASE are pretty obvious. PROPCASE (stands for proper case) capitalizes the first character in each "word" and sets the remaining letters to lowercase. Again, you see "word" in quotes. The default delimiter is a space so the PROPCASE function will capitalize each word. You can specify a list of delimiters as the optional second argument to this function. I recommend specifying both a blank and a single quote as delimiters. Then the function will correctly capitalize such names as D'Angelo. Here is an example:

Program 12

```
data case;
   input Name $15.;
   Upper = upcase(Name);
   Lower = lowcase(Name);
   Proper = propcase(Name," '");
datalines;
gEOrge SMITH
D'Angelo
;
```

Notice that in order to specify a single quote as a delimiter, you need to place the list of delimiters in double quotes.

Figure 12: Output from Program 12

```
Name           Upper         Lower         Proper
gEOrge SMITH   GEORGE SMITH  george smith  George Smith
D'Angelo       D'ANGELO      d'angelo      D'Angelo
```

THE TRANWRD FUCTION

This function performs a find and replace operation on a given string. The three arguments to this function are the input string, the find string, and the replace string. If the replace string is longer than the find string, you may want to use a LENGTH statement for the created variable to avoid having your value truncated. Probably the most common use of the TRANWRD function is address standardization, as demonstrated in the next program:

Program 13

```
data convert;
   input @1 address $20. ;
   *** Convert Street, Avenue and
   Boulevard to their abbreviations;
   Address = tranwrd(Address,'Street','St.');
   Address = tranwrd(Address,'Avenue','Ave.');
   Address = tranwrd(Address,'Road','Rd.');
datalines;
89 Lazy Brook Road
```

```
123 River Rd.
12 Main Street
;
```

In each of the three lines using the TRANWRD function, you are replacing the words Street, Avenue, and Road with their abbreviations.

Figure 13: Output from Program 13

```
Listing of Data Set CONVERT
Obs    Address
 1     89 Lazy Brook Rd.
 2     123 River Rd.
 3     12 Main St.
```

THE SPEDIS FUNCTION

The SEDIS function is one of the most useful functions for inexact matching (also known as fuzzy matching). This function computes the "spelling distance' between two strings. Did you ever misspell a word in Microsoft Word? Well, I never have, but people tell me that if you do misspell a word, Word will underline the misspelt word in red. You can then right-click the mouse to bring up a list of possible correctly spelt words. The SPEDIS function uses a similar algorithm, If the two strings (arguments one and two) match exactly, the function returns a 0. For each category of spelling mistake, the function assigns penalty points. For example, if you get the first letter wrong, you incur a large penalty. If you place two letters in the wrong order (ie vesus ei for example), you get a fairly small number of penalty points. When the function has checked for each category of errors, it divides the total penalty points by the length of the first string. This makes sense. Suppose you get one letter wrong in a three letter word compared to one letter wrong in a ten letter word. The former is certainly a bigger mistake and deserves a larger value for the spelling distance.

What value is consider large for spelling distance? If you allow a very large value for spelling distance in trying to match names from two files, for example, you may be joining observations that do not really belong together. If you only allow very small spelling distances between names, you may not combine two names that actually belong together. To get a feel for what values result from different spelling errors, take a look at the program shown next:

Program 14

```
data compare;
   length String1 String2 $ 15;
   input String1 String2;
   Points = spedis(String1,String2);
datalines;
same same
same sam
first xirst
last lasx
receipt reciept
;
```

Figure 14: Output from Program 14

```
String1    String2    Points
same       same          0
same       sam           8
first      xirst        40
last       lasx         25
receipt    reciept       7
```

You may also consider using the SOUNDEX function to match names in two files. However, I have found that SOUNDEX tends to match names that are quite dissimilar.

TRIMN AND STRIP FUNCTIONS

The TRIMN function removes trailing blanks and the STRIP function removes leading and trailing blanks. TRIMN is similar to the older TRIM function except for how these two functions deal with missing values. When you have a missing value, the TRIM function returns a single blank while the TRIMN function returns a string of zero length. Both of these functions are useful when you are concatenating strings using the concatenation operator, as demonstrated in the following program:

Program 15

```
data _null_;
   length Concat $ 8;
   One = '   ABC   ';
   Two = 'XYZ';
   One_two = ':' || One || Two || ':';
   Trim = ':' || trimn(One) || Two || ':';
   Strip = ':' || strip(One) || strip(Two) || ':';
   Concat = cats(':',One,Two,':');
   put one_two= / Trim= / Strip= /
       Concat=;
run;
```

Figure 15: Output from Program 15

```
One_two=:   ABC   XYZ:
Trim=:   ABCXYZ:
Strip=:ABCXYZ:
Concat=:ABCXYZ:
```

You can see that when you concatenate the two strings (One and Two) without using either of these functions, the result maintains those blanks. Notice that the Trim variable has no blanks between the 'ABC' and 'XYZ' and the Strip variable has no blanks at all. Finally, you can see that it is much easier to use the CATS function to remove leading and trailing blanks and then concatenate the strings.

NOTALPHA, NOTDIGIT AND NOTALNUM FUNCTIONS

I have only listed three of what I call the "not" functions. There are more and you can consult the SAS OnLine Doc or the reference listed at the end of this paper if you are interested. Each of these functions returns the first position in a string that is not a letter (alpha), digit (digit), or alphameric (letter or digit), respectively. Since all of the "not" functions search every position in a string, including trailing blanks, you may choose to strip (or trim) the strings first.

There is an optional second argument to these functions—a starting position from where you start the search. If you enter a negative value for the starting position, the search starts at the absolute value of the position and proceeds from right to left. If any of the functions does not find a non-alpha, non-digit, etc. it returns a 0.

This function is one of the powerhouse functions for data cleaning of character data. You may have a rule about a character value—that it is only consists of digits, or letters, etc.

Here is an example:

Program 16

```
data data_cleaning;
   input String $20.;
   Not_alpha = notalpha(strip(String));
   Not_digit = notdigit(strip(String));
   Not_alnum = notalnum(strip(String));
datalines;
abcdefg
1234567
abc123
1234abcd
;
```

Figure 16: Output from Program 16

```
              Not_     Not_     Not_
  String     alpha    digit    alnum
abcdefg        0        1        0
1234567        1        0        0
abc123         4        1        0
1234abcd       1        5        0
```

CATS AND CATX FUNCTIONS

These two functions concatenate strings. The CATS (I pronounce this Cat – S) function first strips leading and trailing blanks from each of the string before joining them. CATX also strips leading and trailing blanks from strings but inserts delimiters (the first argument to the CATX function) between each of the strings.

One very important point to remember about these functions is that the storage length of the result, if not previously defined, is 200. The default length when you use the concatenation operator (|| or !!) is the sum of the lengths of the strings to be joined.

If you have a list of variables in the form Base1-Basen, you must use the keyword 'OF' before the list. Finally, and this is really cool, values in the lists may be character or numeric. If some of the arguments are numeric, SAS will treat the numbers as if they were characters and will not add any conversion messages to the SAS Log.

The following example shows how these functions perform the stripping operation and how CATX inserts delimiters:

Program 17

```
data join_up;
   length Cats $ 6 Catx $ 13;
   String1 = 'ABC   ';
   String2 = '   XYZ   ';
   String3 = '12345';
   Cats = cats(String1,string2);
   Catx = catx('-',of String1-String3);
run;
```

Figure 17: Output from Program 17

```
Cats = 'ABCXYZ'
Catx = 'ABC-XYZ-12345'
```

COUNT AND COUNTC FUNCTIONS

SAS has two counting functions, COUNT and COUNTC. There difference between them is much the same as the difference between FIND and FINDC. COUNT counts the number of times a substring appears in a string, COUNTC counts the number of times individual characters appear in a string. These functions take the same arguments as the FIND and FINDC functions. The first argument is a string you want to search, the second argument is either a substring (COUNT) or a list of characters (COUNTC). Finally, you can include optional modifiers as the third argument, the 'i' modifier (ignore case) being the most useful. The following program demonstrates these two functions:

Program 18

```
data Dracula; /* Get it Count Dracula */
   input String $20.;
   Count_abc = count(String,'abc');
   Countc_abc = countc(String,'abc');
   count_abc_i = count(String,'abc','i');
datalines;
xxabcxABCxxbbbb
cbacba
;
```

11

Figure 18: Output from Program 18

```
                     Count_   Countc_    Count_
String                abc       abc      abc_i
xxabcxABCxxbbbb         1        7          2
cbacba                 0        6          0
```

AN INTERESTING COMBINATION OF COUNTC AND THE CATS FUNCTIONS

There is a very interesting and powerful way to combine the COUNTC and CATS functions. Suppose you have a survey and you record a Y' or an 'N' for each response. Suppose you want to count the number of Y's (either upper- or lowercase). You might immediately think of putting each of the survey variables in an array, looping through each question, and incrementing a counter every time you encountered a 'Y' value. Take a look at the ingenious way of combining the COUNTC and CATS functions to accomplish this goal. The first time I saw this combination of functions was in an email from my friend Mike Zdeb. I believe he "invented" it, but I'm not certain. Here is the program:

Program 19

```
data Survey;
   input (Q1-Q5)($1.);
   Num = countc(cats(of Q1-Q5),'y','i');
datalines;
yynnY
nnnnn
;
```

The CATS function concatenates all of the survey responses into a single string and the COUNTC function then counts how many Y's (ignore case) there are in the string. I really love this program!

Figure 19: Output from Program 19

```
Listing of Survey
Q1     Q2     Q3     Q4     Q5     Num
y      y      n      n      Y       3
n      n      n      n      n       0
```

SOME DATE FUNCTIONS (MDY, MONTH, WEEKDAY, DAY, YEAR, AND YRDIF)

This section describes some of the most common (and useful) date functions. The MDY function returns a SAS date given a month, day, and year value (the three arguments to the function). The WEEKDAY, DAY, MONTH and YEAR functions all take a SAS date as their argument and return the day of the week (1=Sunday, 2=Monday, etc.), the day of the month (a number from 1 to 31), the month (a number from 1 to 12) and the year respectively.

The YRDIF function computes the number of years between two dates. The first two arguments are the first date and the second date. An optional third argument allows you to specify the number of days in a month and the number of days in a year. For example, for certain financial calculations (such as bond interest), you might specify '30/360' that asks for 30 day months and 360 day years. The YRDIF function had a slight problem computing the difference in years when one or both of the dates fell on a leap year. This problem was fixed in version 9.3. If you are running a version of SAS prior to 9.3, you need to specify 'ACT/ACT' as the third argument to the YRDIF function. Note that the calculation may be off by one day when leap years are involved. This author still believes this is better than subtracting the two dates and dividing by 365.25, the way we computed ages prior to the YRDIF function.

The next program demonstrates all of these date functions:

Program 20

```
data DateExamples;
   input (Date1 Date2)(:mmddyy10.) M D Y;
   SAS_Date = MDY(M,D,Y);
   WeekDay = weekday(Date1);
   MonthDay = day(Date1);
   Year = year(Date1);
```

12

```
   Age = yrdif(Date1,Date2);
   format Date: mmddyy10.;
datalines;
10/21/1955 10/21/2012 6 15 2011
;
```

Figure 20: Output from Program 20

```
            Week    Month
SAS_Date    Day     Day     Year    Age
  18793     10      21      1955    57
```

THE ARRAY FUNCTION

There are times when you define an array and it is not convenient to count the number of elements in the array. A good example of this is when you define an array to be all the numeric or character variables in a data set. If the data set contains lots of variables, you may not want to count them. The DIM function takes as its argument the name of an array and it returns the number of elements in that array. Note that when you define the arrays, you use an asterisk in place of the number of array elements.

In the program that follows, two arrays are defined using the keywords _numeric_ and _character_. These refer to all the numeric or character variables defined at that point in the DATA step. In this program, you want to convert all numerical values of 999 to a SAS missing value and you want to convert all character values to proper case

Program 21

```
data convert;
   input (A B C)($) x1-x3 y z;
   array nums[*] _numeric_;
   array chars[*] _character_;
   do i = 1 to dim(nums);
      if nums[i]=999 then nums[i]=.;
   end;
   do i = 1 to dim(chars);
      chars[i] = propcase(chars[i]," '");
   end;
   drop i;
   datalines;
RON jOhN mary 1 2 999 3 999
;
```

Defining arrays this way can save you lots of time and programming effort.

Figure 21: Output from Program 21

```
A      B      C      x1    x2    x3    y     z
Ron    John   Mary   1     2     .     3     .
```

THE N, NMISS, SUM, AND MEAN FUNCTIONS

These functions are part of what SAS calls the descriptive statistics functions. The SUM and MEAN functions compute a sum or mean. Remember that these functions ignore missing values (this is not the same as treating missing values a zeros).

The N function returns the number of non-missing values in a list of values; the NMISS function returns the number of missing values in a list of values.

The next program illustrates a very useful combination of the N and NMISS functions with the MEAN function. You compute means only if there are a certain number of non-missing (or missing) values in your list of values. Here is the program that does just that:

Program 22

```
data descriptive;
```

```
   input x1-x5;
   Sum = sum(of x1-x5);
   if n(of x1-x5) ge 4 then
      Mean1 = mean(of x1-x5);
   if nmiss(of x1-x5) le 3 then
      Mean2 = mean(of x1-x5);
datalines;
1 2 . 3 4
. . . 8 9
;
```

In this program, you compute Mean1 only if there are 4 or more non-missing values. You compute Mean2 only if there are 3 or fewer missing values.

Figure 22: Output from Program 22

```
Sum    Mean1    Mean2
 10     2.5      2.5
 17      .       8.5
```

THE SMALLEST AND LARGEST FUNCTIONS

It is very easy to find the largest (MAX function) or smallest value (MIN function) in a list of values. However, it used to be quite difficult to determine the second largest or smallest number or the third etc. You can use the SMALLEST and LARGEST functions to find the nth smallest or largest number in a list of values. The first argument to each of these functions specifies if you want the smallest, second smallest, etc. (or largest) in a list of values. For example, SMALLEST(1,of x1-x10) is the same as MIN(of x1-x10). SMALLEST(2, of x1-x10) returns the second smallest value. Note that both of these functions ignore missing values. The following program demonstrates these two functions:

Program 23

```
data descriptive;
   input x1-x5;
   S1 = smallest(1,of x1-x5);
   S2 = smallest(2,of x1-x5);
   L1 = largest(1,of x1-x5);
   L2 = largest(2,of x1-x5);
datalines;
7 2 . 6 4
10 . 2 8 9
;
```

Figure 23: Output from Program 23

x1	x2	x3	x4	x5	S1	S2	L1	L2
7	2	.	6	4	2	4	7	6
10	.	2	8	9	2	8	10	9

THE LAG FUNCTION

Since SAS operates on one observation at a time, you need a way to obtain values from previous observations. For example, you might want to compute a difference from one observation to the next. The LAG function provides this capability. The LAG function returns the value of its argument the last time the function executed. This is important! If you execute the LAG function conditionally (i.e. following an IF statement), and the condition is not true, the next time you execute the function, you will not obtain a value from the previous observation—you will obtain a value from the last time the IF statement was true.

There are a family of LAG functions: LAG2 returns a value from two previous arguments, LAG3 from three times, and so forth. If you execute the LAG functions for every iteration of the DATA step, they will return values from a previous observation, two back, three back and so forth.

One common use of the LAG function is to compute differences between observations. Another is to compute a moving average, the subject of the next program:

Program 24

```
data Moving;
   input X @@;
   Moving = mean(X,lag(x),lag2(x));
datalines;
50 40 55 20 70 50
;
```

In this program, you compute the mean of the current value and the two previous values.

Figure 24: Output from Program 24

```
X      Moving
50     50.0000
40     45.0000
55     48.3333
20     38.3333
70     48.3333
50     46.6667
```

THE CALL SORTN ROUTINE

You can sort within an observation using the CALL SORTN routine. You supply the call routine with a list of variables. After the call, the values of all the variables have changed and they are now in ascending order. Take a look at the following program to see how this works. Note that there is also a CALL SORTC routine for character values. When you use that routine, all of the character values have to be the same length and the result is an alphabetical sort.

The program below reads in 5 scores. After the call, the values of Score1 to Score5 have changed. You also want to compute the mean of the 3 highest scores. You will find lots of uses for this amazing call routine.

Program 25

```
data Scores;
   input Score1-Score5;
   call sortn(of Score1-Score5);
   Top3 = mean(of Score3-Score5);
datalines;
80 70 90 10 80
;
```

In the output below, you see that the original scores were 80, 70, 90, 10, and 80 but they have traded places so that Score1 is the lowest, Score2, the next lowest, and so forth.

Figure 25: Output from Program 25

```
Score1    Score2    Score3    Score4    Score5    Top3
  10        70        80        80        90      83.333
```

CONCLUSION

This paper covered some of the most useful functions in SAS. I understand that it may be a bit overwhelming, but I just couldn't leave out some of my favorites. I think you can see how indispensable SAS functions (and call routines) are to DATA step programming.

REFERENCES

Cody, Ron, 2010, *SAS Functions by Example*, Second edition, SAS Press, Cary, NC., SAS OnLine Doc., SAS Institute, Cary, NC.

ABOUT THE AUTHOR

Dr. Ron Cody was a Professor at the Robert Wood Johnson Medical School in New Jersey for 26 years and is now a private consultant and writer. He has been a SAS user since the late 70's and is the author of Applied Statistics and the SAS® Programming Language (fifth edition), published by Prentice Hall. He has also authored the following books with SAS Press: *Learning SAS by Example: A Programmer's Guide*, *SAS Functions by Example*, 2nd edition, *Cody's Data Cleaning Techniques*, 2nd edition, *Longitudinal Data and SAS: A Programmer's Guide* and *The SAS Workbook*. Ron is currently working on a new book, tentatively titled: 101 *Common SAS Programming Tasks and How to Solve Them*. Ron has presented invited papers for numerous local, regional, and national SAS conferences.

CONTACT INFORMATION

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

Name: Ron Cody
Address: PO Box 5049
City, State ZIP: Camp Verde, TX 78010
E-mail: ron.cody@gmail.com

Check out my author page at: support.sas.com/cody