An Introduction to SAS® Character Functions
Ronald Cody, Ed.D.

Introduction

SAS® software is especially rich in its assortment of functions that deal with character data. This class of functions is sometimes called STRING functions. With over 30 new character functions in SAS® 9, the power of SAS to manipulate character data is even more impressive.

Some of the functions we will discuss are: LENGTH, SUBSTR, COMPBL, COMPRESS, VERIFY, INPUT, PUT, TRANWRD, SCAN, TRIM, UPCASE, LOWCASE, || (concatenation), INDEX, INDEXC, AND SPEDIS. Some of the new and exciting SAS® 9 functions that we will cover are the "ANY' and "NOT" functions, the concatenation functions (and call routines), COMPARE, INDEXW, LENGTHC, PROPCASE, STRIP, COUNT, and COUNTC.

How Lengths of Character Variables are Set in a SAS Data Step

Before we actually discuss these functions, we need to understand how SAS software assigns storage lengths to character variables. It is important to remember two things: 1) The storage length of a character variable is set at compile time. and 2) this length is determined by the first appearance of a character variable in a DATA step. There are several ways to check the storage length of character variables in your SAS data set. One way is to run PROC CONTENTS. Another is to use the SAS Explorer window and select "view columns." If you are using SAS® 9 and above, the new function LENGTHC can be used to determine the storage length of a character variable. Look at the following program:

```
data chars1;
   file print;
   string = 'abc';
   length string $ 7; /* Does this do anything? */
   storage_length = lengthc(string);
   display = ":" || string || ":";
   put storage_length=;
   put display=;
run;
```

What is the storage length of STRING? Following the rules, the length is set by the assignment statement string = 'abc' which results is a storage length of 3. The LENGTH statement is ignored (however in SAS® 9, an informative note is written in the SAS log). The LENGTHC function shows the storage length of STRING to be 3 (as would output from PROC CONTENTS or the "view columns" from the SAS Explorer). The || operator is the concatenation operator which joins strings together. By concatenating a colon on each side of the variable STRING, you can see if there are any leading or trailing blanks in the value. Look at the SAS output below:

```
storage_length=3
display=:abc:
```

What if we move the LENGTH statement before the assignment statement?

```
data chars2;
```

```
   file print;
   length string $ 7; /* Does this do anything? */
   string = 'abc';
   storage_length = lengthc(string);
   display = ":" || string || ":";
   put storage_length=;
   put display=;
run;
```

Let's look at the output again:

```
storage_length=7
display=:abc    :
```

Notice that the storage length of STRING is now 7. The DISPLAY variable clearly shows the actual value of STRING is 'abc' followed by 4 blanks.

Converting Multiple Blanks to a Single Blank

This example will demonstrate how to convert multiple blanks to a single blank. Suppose you have some names and addresses in a file. Some of the data entry clerks placed extra spaces between the first and last names and in the address fields. You would like to store all names and addresses with single blanks. Here is an example of how this is done:

```
data multiple;
   input #1 @1  name    $20.
         #2 @1  address $30.
         #3 @1  city    $15.
            @20 state    $2.
            @25 zip      $5.;
   name = compbl(name);
   address = compbl(address);
   city = compbl(city);
datalines;
Ron Cody
89 Lazy Brook Road
Flemington         NJ   08822
Bill     Brown
28   Cathy   Street
North   City       NY   11518
;
title "Listing of Data Set MULTIPLE";
proc print data=multiple noobs;
   id name;
   var address city state zip;
run;
```

Here is the listing:

```
Listing of Data Set MULTIPLE

   name            address             city       state     zip

Ron Cody      89 Lazy Brook Road    Flemington     NJ      08822
Bill Brown    28 Cathy Street       North City     NY      11518
```

This seemingly difficult task is accomplished in a single line using the COMPBL function. It COMPresses successive blanks to a single blank. How useful!

How to Remove Characters from a String

A more general problem is to remove selected characters from a string. For example, suppose you want to remove blanks, parentheses, and dashes from a phone number that has been stored as a character value. Here comes the COMPRESS function to the rescue! The COMPRESS function can remove any number of specified characters from a character variable. The program below uses the COMPRESS function twice. The first time, to remove blanks from the string; the second to remove blanks plus the other above mentioned characters. Here is the code:

```
data phone;
   input phone $ 1-15;
   phone1 = compress(phone);
   phone2 = compress(phone,'(-) ');
datalines;
(908)235-4490
(201) 555-77 99
;
title "Listing of Data Set PHONE";
proc print data=phone noobs;
run;
```

Here is the listing:

```
Listing of Data Set PHONE

     phone            phone1          phone2

(908)235-4490      (908)235-4490    9082354490
(201) 555-77 99     (201)555-7799    2015557799
```

The variable PHONE1 has just blanks removed. Notice that the COMPRESS function does not have a second argument here. When it is omitted, the COMPRESS function removes only blanks. For the variable PHONE2, the second argument of the COMPRESS function contains a list of the characters to remove: left parenthesis, blank, right parenthesis, and dash. This string is placed in single or double quotes. Remember, when you specify a list of characters to remove, blanks are no longer included unless you explicitly include a blank in the list.

Character Data Verification

A common task in data processing is to validate data. For example, you may want to be sure that only certain values are present in a character variable. In the example below, only the values 'A', 'B', 'C', 'D', and 'E' are valid data values. A very easy way to test if there are any invalid characters present is shown next:

```
data verify;
   input @1  id $3.
         @5  answer $5.;
   position = verify(answer,'abcde');
datalines;
001 acbed
002 abxde
003 12cce
004 abc e
;
title "Listing of Data Set VERIFY";
proc print data=verify noobs;
run;
```

The workhorse of this example is the VERIFY function. It is a bit complicated. It inspects every character in the first argument and, if it finds any value not in the verify string (the second argument), it will return the position of the first offending value. If all the values of the string are located in the verify string, a value of 0 is returned. To help clarify this, look at the listing below:

```
 Listing of Data Set VERIFY

id     answer    position

001    acbed         0
002    abxde         3
003    12cce         1
004    abc e         4
```

One thing to be careful of when using the VERIFY function (and many of the other character functions) is trailing blanks. For example, look at the following:

```
data trailing;
   length string $ 10;
   string = 'abc';
   pos = verify(string,'abcde');
run;
```

The value of POS is 4, the position of the first trailing blank. One way to avoid the trailing blank problem is to remove the trailing blanks before using the verify function. The TRIM function does this for us. We change the assignment statement to read:

```
pos = verify(trim(string),'abcde');
```

and the result is a 0. Another approach would be to include a blank in the verify string. However, this would not detect blanks in the middle of the string either.

Substring Example

We mentioned in the Introduction that a substring is a part a longer string (although it can actually be the same length but this would not be too useful). In this example, you have ID codes which contain in the first two positions, a state abbreviation. Furthermore, positions 7-9 contain a numeric code. You want to create two new variables; one containing the two digit state codes and the other, a numeric variable constructed from the three numerals in positions 7,8, and 9. Here goes:

```
data pieces_parts;
   input id $ 1-9;
   length state $ 2;
   state = substr(id,1,2);
   num = input(substr(id,7,3),3.);
datalines;
NYXXXX123
NJ1234567
;
title "Listing of Data Set PIECES_PARTS";
proc print data= pieces_parts noobs;
run;
```

Creating the state code is easy. We use the SUBSTR function. The first argument is the variable from which we want to extract the substring, the second argument is the starting position of the substring, and the last argument is the length of the substring (not the ending position as you might guess). Also note the use of the LENGTH statement to set the length of STATE to 2 bytes. Without a LENGTH statement, the length of STATE would be the same as the length of ID. Why? Remember that character variable lengths are set at compile time. The starting position and length parameters are constants here, but they could have been computed or read in from an external file. So, without a LENGTH statement, what is SAS to do? What is the longest substring you can extract from a string of length n? The answer is n and that is what SAS uses as the default length of the result.

Extracting the three digit number code is more complicated. First we use the SUBSTR function to pull out the three numerals (numerals are character representations of numbers). However, the result of a SUBSTR function is always a character value. To convert the character value to a number, we use the INPUT function. The INPUT function takes the first argument and "reads" it as if it were coming from a file, according to the informat listed as the second argument. So, for the first observation, the SUBSTR function would return the string '123' and the INPUT function would convert this to the number 123. As a point of interest, you may use a longer informat as the second argument without any problems. For example, the INPUT statement could have been written as:

```
input (substr(id,7,3),8.);
```

and everything would have worked out fine. This fact is useful in situations where you do not know the length of the string ahead of time.

Using the SUBSTR Function on the Left-Hand Side of the Equal Sign

There is a particularly useful and somewhat obscure use of the SUBSTR function that we would like to discuss next. You can use this function to place characters in specific locations within a string by placing the SUBSTR function on the left hand side of the equal sign (in the older manuals I think this was called a SUBSTR pesudo function).

Suppose you have some systolic blood pressures (SBP) and diastolic blood pressures (DBP) in a SAS data set. You want to print out these values and star high values with an asterisk. Here is a program that uses the SUBSTR function on the left of the equals sign to do that:

```
data pressure;
   input sbp dbp @@;
   length sbp_chk dbp_chk $ 4;
   sbp_chk = put(sbp,3.);
   dbp_chk = put(dbp,3.);
   if sbp gt 160 then
      substr(sbp_chk,4,1) = '*';
   if dbp gt 90 then
      substr(dbp_chk,4,1) = '*';
datalines;
120 80 180 92 200 110
;
title "Listing of Data Set PRESSURE";
proc print data=pressure noobs;
run;
```

We first need to set the lengths of SBP_CHK and DBP_CHK to 4 (three spaces for the value plus one for the possible asterisk). Next, we use a PUT function to perform a numeric to character conversion. The PUT function is, in some ways, similar to the INPUT function. It "writes out" the value of the first argument, according to the FORMAT specified in the second argument. By "write out" we actually mean assign the value to the variable on the left of the equal sign. The SUBSTR function then places an asterisk in the fourth position when a value of SBP is greater than 160 or a value of DBP is greater than 90, as you can see in the output below:

```
Listing of Data Set PRESSURE

sbp    dbp    sbp_chk    dbp_chk

120     80     120         80
180     92     180*        92*
200    110     200*       110*
```

Unpacking a String

To save disk storage, you may want to store several single digit numbers in a longer character string. For example, storing five numbers as numeric variables with the default 8 bytes each would take up 40 bytes of disk storage per observation. Even reducing this to 3 bytes each would result in 15 bytes of storage. If, instead, you store the five digits as a single character value, you need only 5 bytes.

This is fine, but at some point, you may need to get the numbers back out for computation purposes. Here is a nice way to do this:

```
data pack;
   input string $ 1-5;
datalines;
12345
8 642
;
data unpack;
   set pack;
   array x[5];
   do j = 1 to 5;
      x[j] = input(substr(string,j,1),1.);
   end;
   drop j;
run;

title "Listing of Data Set UNPACK";
proc print data=unpack noobs;
run;
```

We first created an array to hold the five numbers, X1 to X5. Don't be alarmed if you don't see any variables listed on the ARRAY statement. ARRAY X[5]; is equivalent to ARRAY X[5] X1-X5; We use a DO loop to cycle through each of the 5 starting positions corresponding to the five numbers we want. As we mentioned before, since the result of the SUBSTR function is a character value, we need to use the INPUT function to perform the character to numeric conversion.

Parsing a String

Parsing a string means to take it apart based on some rules. In the example to follow, five separate character values were placed together on a line with either a space, a comma, a semi-colon, a period, or an explanation mark between them. You would like to extract the five values and assign them to five character variables. Without the SCAN function this would be hard; with it, it's easy:

```
data parse;
   input long_str $ 1-80;
   array pieces[5] $ 10
      piece1-piece5;
   do i = 1 to 5;
      pieces[i] = scan(long_str,i,',.! ');
   end;
   drop long_str i;
datalines;
this line,contains!five.words
abcdefghijkl xxx yyy
;
title "Listing of Data Set PARSE";
proc print data=parse noobs;
run;
```

The function:

```
SCAN(char_var,n,'list-of-delimiters');
```

returns the *n*th "word" from the char_var, where a "word" is defined as anything between two delimiters. If there are fewer than *n* words in the character variable, the SCAN function will return a blank. If *n* is negative, the scan will proceed from right to left. If *n* is greater than the number of words in the string, a missing value is returned.

By placing the SCAN function in a DO loop, we can pick out the *n*th word in the string.

Using the SCAN Function to Extract a Last Name

Here is an interesting example that uses the SCAN function to extract the last name from a character variable that contains first and last name as well as a possible middle name or initial. In this example, you want to create a list in alphabetical order by last name. First the program, then the explanation:

```
data first_last;
   input @1  name  $20.
         @21 phone $13.;
   ***extract the last name from name;
   last_name = scan(name,-1,' '); /* scans from the right */
datalines;
Jeff W. Snoker       (908)782-4382
Raymond Albert       (732)235-4444
Alfred Edward Newman (800)123-4321
Steven J. Foster     (201)567-9876
Jose Romerez         (516)593-2377
;
title "Names and Phone Numbers in Alphabetical Order (by Last Name)";
proc report data=first_last nowd;
   columns name phone last_name;
   define last_name / order noprint width=20;
   define name      / display 'Name' left width=20;
   define phone     / display 'Phone Number' width=13 format=$13.;
run;
```

It is easy to extract the last name by using a –1 as the second argument of the SCAN function. Remember, a negative value for this arguments results in a scan from right to left. Output from the REPORT procedure is show below:

```
Names and Phone Numbers in Alphabetical Order (by Last Name)

  Name                 Phone Number
  Raymond Albert       (732)235-444
  Steven J. Foster     (201)567-987
  Alfred Edward Newman (800)123-432
  Jose Romerez         (516)593-237
  Jeff W. Snoker       (908)782-438
```

Locating the Position of One String Within Another String

Two somewhat similar functions, INDEX and INDEXC can be used to locate a string, or one of several strings within a longer string. For example, if you have a string 'ABCDEFG' and want the location of the letters DEF (starting position 4), the following INDEX function could be used:

```
INDEX('ABCDEFG','DEF');
```

The first argument is the argument you want to search, the second argument is the string you are searching for. This would return a value of 4, the starting position of the string 'DEF'. If you want to know the starting position of any one of several characters, the INDEXC function can be used. As an example, if you wanted the starting position of any of the letters 'G', 'C', 'B', or 'F' in the string 'ABCDEFG', you would code:

```
INDEXC('ABCDEFG','GCBF');
```

or

```
INDEXC('ABCDEFG','G','C','B','F');
```

The function would return a value of 2, the position of the 'B', the first letter found in the first argument. Here is a short program which demonstrate these two functions: If the search fails, both functions return a zero.

```
data locate;
   input string $ 1-10;
   first = index(string,'xyz');
   first_c = indexc(string,'x','y','z');
datalines;
abcxyz1234
1234567890
abcx1y2z39
abczzzxyz3
;
title "Listing of Data Set LOCATE";
proc print data=locate noobs;
run;
```

FIRST and FIRST_C for each of the 4 observations are:

```
obs  first     first_c
 1     4          4
 2     0          0
 3     0          4
 4     7          4
```

Changing Lower Case to Upper Case and Vice Versa

The two companion functions UPCASE and LOWCASE do just what you would expect. These two functions are especially useful when data entry clerks are careless and a mixture of upper

and lower cases values are entered for the same variable. You may want to place all of your character variables in an array and UPCASE (or LOWCASE) them all. Here is an example of such a program:

```
data up_down;
   length a b c d e $ 1;
   input a b c d e x y;
datalines;
M f P p D 1 2
m f m F M 3 4
;
data upper;
   set up_down;
   array all_c[*] _character_;
   do i = 1 to dim(all_c);
      all_c[I] = upcase(all_c[i]);
   end;
   drop i;
run;

title "Listing of Data Set UPPER";
proc print data=upper noobs;
run;
```

This program uses the _CHARACTER_ keyword to select all the character variables. The result of running this program is to convert all values for the variables A,B,C, D, and E to upper case. The LOWCASE function could be used in place of the UPCASE function if you wanted all your character values in lower case.

Converting String to Proper Case

A handy new V9 function is PROPCASE. This function capitalizes the first letter of each word. Here is an example:

```
data proper;
   input name $40.;
datalines;
rOn coDY
the tall and the short
the "%$#@!" escape
;
title "Listing of Data Set PROPER";
proc print data=proper noobs;
run;
```

As you can see in the listing below, this function is useful and easy to use.

```
Listing of Data Set PROPER

name

Ron Cody
The Tall And The Short
The "%$#@!" Escape
```

Substituting One Word for Another in a String

TRANWRD (translate word), can perform a search and replace operation on a string variable. For example, you may want to standardize addresses by converting the words 'Street', 'Avenue', and 'Road' to the abbreviations 'St.', 'Ave.', and 'Rd.' respectively. Look at the following program:

```
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
;
title "Listing of Data Set CONVERT";
proc print data=convert;
run;
```

The syntax of the TRANWRD function is:

```
TRANWRD (char_var,'find_str','replace_str');
```

That is, the function will replace every occurrence of find_str with replace_str. Notice that the order of the find and replace strings are reversed compared to the TRANSLATE function where the to_string comes before the from_string as arguments to the function. In this example, 'Street' will be converted to 'St.', 'Avenue' to 'Ave.', and 'Road' to 'Rd.'. The listing below confirms this fact:

```
Listing of Data Set CONVERT

OBS    ADDRESS

 1     89 Lazy Brook Rd.
 2     123 River Rd.
 3     12 Main St.
```

Fuzzy Merging: The SPEDIS Function

The SPEDIS function measures the "spelling distance" between two strings. If the two strings are identical, the function returns a 0. For each category of spelling error, the function assigns "penalty" points. For example, if the first letter in the two strings is different, there is a relatively large penalty. If two letters are interchanged, the number of penalty points is smaller. This function is very useful in performing a fuzzy merge where there may be differences in spelling between two files. You can also use it with character data consisting of numerals, such as

social security numbers. The syntax of this function is:

```
SPEDIS(string1,string2);
```

The program below demonstrates how this function works.

```
data compare;
   length string1 string2 $ 15;
   input string1 string2;
   points = spedis(string1,string2);
datalines;
same same
same sam
firstletter xirstletter
lastletter lastlettex
receipt reciept
;
title "Listing of Data Set COMPARE";
proc print data=compare noobs;
run;
```

Here is the listing:

```
Listing of Data Set COMPARE

string1        string2        points

same           same             0
same           sam              8
firstletter    xirstletter     18
lastletter     lastlettex      10
receipt        reciept          7
```

Demonstrating the "ANY" Functions

The "ANY" functions of SAS® 9 allow you to determine the position of a class of characters (digits, alpha-numeric, alpha, white space, or punctuation). The complete list is:

```
ANYALNUM, ANYALPHA, ANYDIGIT, ANYPUNCT, and ANYSPACE
```

These functions return the first position of a character of the appropriate class. If no appropriate characters are found, the functions return a 0. We will demonstrate the ANYALPHA and ANYDIGIT functions in the following program:

```
data find_alpha_digit;
   input string $20.;
   first_alpha = anyalpha(string);
   first_digit = anydigit(string);
datalines;
no digits here
the 3 and 4
123 456 789
;
```

```
  title "Listing of Data Set FIND_ALPHA_DIGIT";
  proc print data=find_alpha_digit noobs;
  run;
```

```
Listing of Data Set FIND_ALPHA_DIGIT

                  first_    first_
    string         alpha     digit

no digits here       1         0
the 3 and 4          1         5
123 456 789          0         1
```

Demonstrating the "NOT" Functions

The "NOT" functions work in a similar way to the "ANY" functions except that they return the position of the first class of character that does not match the designated class.

The complete list is:

```
  NOTALNUM, NOTALPHA, NOTDIGIT, NOTPUNCT, and NOTSPACE
```

These functions return the first position of a character that does not match the appropriate class. If no "bad" characters are found (characters that do not match the designation)the functions return a 0. We will demonstrate the NOTALPHA and NOTDIGIT functions in the following program:

```
  data data_cleaning;
     input string $20.;
     only_alpha = notalpha(trim(string));
     only_digit = notdigit(trim(string));
  datalines;
  abcdefg
  1234567
  abc123
  1234abcd
  ;
  title "Listing of Data Set DATA_CLEANING";
  proc print data=data_cleaning noobs;
  run;
```

```
Listing of Data Set DATA_CLEANING

            only_    only_
 string     alpha    digit

abcdefg       0        1
1234567       1        0
abc123        4        1
1234abcd      1        5
```

Notice the use of the TRIM function in the program above. Without it, both NOTALPHA and NOTDIGIT would return the position of the first blank. You can see why we called the data set

DATA_CLEANING—it is an easy way to determine if a string contains only a single class of characters.

The New Concatenation Functions

Although you can concatenate (join) two strings using the concatenation operator (either || or !!), several new functions and call routines, new with SAS® 9, can be used instead. The advantage of these new functions is that they can automatically strip off leading and trailing blanks and can insert separation characters for you. We will demonstrate only two of the concatenation functions here. The CATS function strips leading and trailing blanks before joining two or more strings; the CATX function works the same as the CATS function but allows you to specify one or more separation characters to insert between the strings. The syntax for these two functions is:

```
CATS(string1,string2,<stringn>);

CATX(separator,string1,string2,<stringn>);
```

The program below demonstrates these two function:

```
data join_up;
   length cats $ 6 catx $ 17;
   string1 = 'ABC   ';
   string2 = '   XYZ   ';
   string3 = '12345';
   cats = cats(string1,string2);
   catx = catx('***',string1,string2,string3);
run;
title "Listing of Data Set JOIN_UP";
proc print data=join_up noobs;
run;
```

With the resulting output below:

```
Listing of Data Set JOIN_UP

string1    string2    string3     cats           catx

    ABC        XYZ       12345     ABCXYZ     ABC***XYZ***12345
```

By the way, without the LENGTH statement in this program, the length of the two variables CATS and CATX would be 200 (the default length for the CAT functions). Note that the default length when using the concatenation operator is the sum of the lengths of the arguments to be joined.

The LENGTH, LENGTHN, and LENGTHC Functions

We spoke earlier about storage length of character variables. The collection of LENGTH functions in this section have different and useful purposes. The LENGTHC function (V9) returns the storage length of character variables. The other two functions, LENGTH and LENGTHN both return the length of a character variable not counting trailing blanks. The only difference between LENGTH and LENGTHN is that LENGTHN returns a 0 for a null string while

LENGTH returns a 1. The short program and the listing below demonstrates this collection of functions:

```
data how_long;
   one = 'ABC   ';
   two = ' '; /* character missing value */
   three = 'ABC   XYZ';
   length_one = length(one);
   lengthn_one = lengthn(one);
   lengthc_one = lengthc(one);
   length_two = length(two);
   lengthn_two = lengthn(two);
   lengthc_two = lengthc(two);
   length_three = length(three);
   lengthn_three = lengthn(three);
   lengthc_three = lengthc(three);
run;
title "Listing of Data Set HOW_LONG";
proc print data=how_long noobs;
run;
```

```
Listing of Data Set HOW_LONG

                           length_    lengthn_    lengthc_    length_
one    two      three        one         one         one        two

ABC           ABC   XYZ       3           3           6          1

lengthn_    lengthc_    length_    lengthn_    lengthc_
   two         two        three      three       three

      0           1          9          9           9
```

Comparing Two Strings Using the COMPARE Function

You may wonder why we need a function to compare two character strings. Why can't you simple use an equal sign? The COMPARE function gives you more flexibility in comparing strings. The syntax is:

```
COMPARE(string1, string2 <,'modifiers'>)
```

You may use one or more modifiers from the following list, placed in single or double quotes as follows:

i or I	ignore case
l or L	remove leading blanks
n or N	remove quotes from any argument that is an n-literal and ignore case An n-literal is a string in quotes, followed by an 'n', useful for non-valid SAS names
: (colon)	truncate the longer string to the length of the shorter string. Note that the default is to pad the shorter string with blanks before a comparison. (Note: similar to the =: comparison operator)

For example, if you want to ignore case and remove leading blanks, you could code:

```
yes_or_no = compare(string1,string2,'il');
```

The colon modifier if also useful when you want to truncate a longer string to the length of a shorter string before making the comparison.

Removing Leading and Trailing Blanks Using the STRIP Function

A new V9 function STRIP removes leading and trailing blanks. The two statements:

```
if strip(string) = 'abc' then result = 'yes';

if left(trim(string)) = 'abc' then result = 'yes';
```

are equivalent. Notice that the LEFT function removes leading blanks while the TRIM function removes trailing blanks.

Counting Occurrences of Characters or Substrings Using the COUNT and COUNTC Functions

Two other SAS® 9 functions are COUNT and COUNTC. COUNT is used to count the number of times a particular substring appears in a string. COUNTC counts the number of times one or more characters appear. You can also use modifiers to ignore case (use the 'i' modifier) or ignore trailing blanks (use the 't' modifier) in both the string and the find_string..

The syntax for these two functions is:

```
count(string,find_string,<'modifiers'>)

countc(string,find_string,<'modifiers'>)
```

The following program and listing demonstrate these two functions:

```
data Dracula; /* Get it – Count Dracula */
   input string $20.;
   count_a_or_b = count(string,'ab');
   countc_a_or_b = countc(string,'ab');
   count_abc = count(string,'abc');
   countc_abc = countc(string,'abc');
   case_a = countc(string,'a','i');
datalines;
xxabcxabcxxbbbb
cbacba
aaAA
;
title "Listing of Data Set DRACULA";
proc print data=Dracula noobs;
run;
```

Listing of Data Set DRACULA

string	count_ a_or_b	countc_ a_or_b	count_ abc	countc_ abc	case_a
xxabcxabcxxbbbb	2	8	2	10	2
cbacba	0	4	0	6	2
aaAA	0	2	0	2	4

Conclusions

So ends our tour through some of the more useful character functions. So go out there and have a ball with strings!

Ronald P. Cody, Ed.D.
ron.cody@gmail.com