# Introduction to Character String Functions

Jason Ford, Bureau of Labor Statistics

## ABSTRACT

Character string functions allow a user to manipulate character variables in a variety of ways. Users can create new variables out of parts of existing character variables, verify the contents of variables, find information within a variable, concatenate variables, and eliminate unnecessary blanks. This paper introduces a programmer to the functions in SAS® that do these various tasks.

## INTRODUCTION

Training of new SAS® programmers usually begins with a focus on numerical data. That approach makes sense, because calculations are at the heart of SAS®. Programmers will usually soon discover, however, that many programming problems involve character variables.

A character variable is a variable whose values can consist of both nonnumeric and numeric characters. Such variables include cases where all the values have just letters, but also include values that are a mix of letters, numbers, and other characters. Even variables with all numbers can be saved as character variables, although doing so would not be advisable if the numbers are intended for mathematical calculations.

SAS® has many functions to manipulate these variables. Some things users can do with character variables include:

1. create new variables out of parts of existing variables
2. verify the contents of variables
3. find information within a character variable
4. concatenate variables
5. eliminate unnecessary blanks.

## THE LENGTH STATEMENT: AN IMPORTANT SAFETY MEASURE

When creating a new variable using a character function, a good strategy is to assign a LENGTH statement to avoid unexpected results. Without a LENGTH statement, SAS® will give a variable a default length, which could mean a lot of blank spaces.

SAS® places blank spaces at the end of character variables if the assigned characters in the variable do not take up the entire length. For example, if the programmer assigns the value of “dog” to a character variable with a length of six, for example, SAS® would save that value as the letters d, o, and g followed by three blanks.

In some cases, those blanks at the end of a value can be unexpectedly large. For example, the following code uses the TRANWRD function, which can change one word in a character variable to another word:

```
DATA roads;
       INPUT the_word_street $ 1-6;
       DATALINES;
street
;
DATA roads2;
       SET roads;
       abbreviation_st=TRANWRD (the_word_street,'street','st');
run;
```

Most new SAS® programmers would not guess that the length of abbreviation_st would be 200 characters long! Because Abbreviation_st was not assigned a length, SAS® used the default length of 200. This one value gets stored as the letters s and t followed by 198 blank spaces.

A better approach is to assign the length using the LENGTH function:

```
DATA roads2;
      SET roads;
      LENGTH abbreviation_st $ 2;
      abbreviaton_st=TRANWRD(the_word_street,'street','st');
run;
```

## PROGRAMMING TERMINOLOGY: ARGUMENTS

Character functions generally have one to three **arguments.** An argument is simply a term for "a piece of information a SAS® program needs so it can do what you want it to do." The arguments are separated by commas. The following statement has three arguments:

```
abbreviaton_st=TRANWRD (the_word_street,'street','st');
```

Character functions usually have one to three options, although some of the concatenation arguments can have many more arguments.

## SUBSTR: BREAKING A CHARACTER VARIABLE APART BASED ON POSITION

SUBSTR takes some part of a character string and creates a new variable. The three arguments are:

1. The original variable name
2. The position where we want to start taking information for the new variable.
3. The length of data we want to take.

Let us say we have a variable called Pile_of_rocks that has just one record. This one record is EGGGEE. Assuming E is for emeralds and G is for gold, let us say we have a leprechaun who wants just the gold. [Because much of working with character variables involves looking for meaningful information ("gold") amidst much of what do not want ("other rocks"), the leprechaun analogy will be used a few times in this paper.]

```
Gold=SUBSTR(Pile_of_rocks,2,3);
```

The value of "Gold" is GGG. The variable Gold starts at the second position and goes for three spaces including that second position. The variable Gold thus has the values of the second, third, and fourth position.

Since the program did not assign a length, however, "Gold" gets the same length as "Pile_of_rocks." The value of "Gold" would be GGG followed by three blanks. We could use a LENGTH statement to get rid of those blanks.

```
LENGTH gold $3.;
```

If we left off the third argument of the SUBSTR function, the resulting function would just take all characters to the end of the variable. Let's say we had the following code:

```
Gold=SUBSTR (Pile_of_rocks,2)
```

The one record in the variable Gold would then equal GGGEE.

Leaving off the third argument would be useful in some cases. If we had a list of names where all last names started on character 17, for example, this approach might be useful.

## THE SCAN FUNCTION—BREAKING A CHARACTER VARIABLE APART BASED ON WHAT IS IN THE CHARACTER VARIABLE

For SUBSTR to get useful results, the data usually have to be lined up in columns. Say we had a list of a thousand names that we wanted to break into the first and last name. If every person's first name started at one position and last name started at another position, SUBSTR would work well. SUBSTR could separate the following data into first and last names, for example:

```
Jonathan Brown
Amelia   Hernandez
Helen    Wong
```

On the other hand, if the data were presented as follows, getting useful results from SUBSTR would be difficult:

```
Jonathan Brown
Amelia Hernandez
Helen Wong
```

The SCAN function would work well in this latter case. The SCAN function allows the programmer to extract parts of a character string for a new variable based on the information in that string. One programmer described the SCAN function as "breaking a character string into words." That description is good, but we should remember that "words" can be defined any number of ways. SCAN can use a blank space as a separator, which indeed breaks a variable into words. It can also use a comma, backslash, or anything else the user wishes.

In the following example, the user uses a blank space to separate first, middle, and last names.

```
DATA names;
       INPUT name $30.;
       DATALINES;
       John Hammond Smith
       Dave Ramon Hernandez
       Jean Marie Yang
              ;
Run;

DATA names2;
       SET names;
       LENGTH first_name $30.;
       LENGTH middle_name $30.;
       LENGTH last_name $30.;
       First_name=SCAN (name,1,'');
       Middle_name =SCAN (name,2,'');
       Last_name=SCAN (name,3,'');
run;
```

The data set names2 would be as follows:

| Obs | name | First_ name | Middle_ name | Last_name |
|---|---|---|---|---|
| 1 | John Hammond Smith | John | Hammond | Smith |
| 2 | Dave Ramon Hernandez | Dave | Ramon | Hernandez |
| 3 | Jean Marie Yang | Jean | Marie | Yang |

SCAN has three arguments. The first is the variable name. The second is the starting position, and the third is the separator.

The best way to explain the second and third arguments is to begin by explaining the third argument. The separator is what SAS® uses to break the character string into parts. The separator could be a blank space, comma,

semicolon, or any other character. The example above used a blank space. (If you put in multiple characters in the third argument, it will interpret those characters as an "or" statement and use the first of the list of characters it finds.)

The second argument, and perhaps least intuitively obvious, is the starting position. A value of 1 indicates that the created variable should start at the beginning and go to the first instance of the separator. A value of 2 indicates that the created variable take everything from the first instance of the separator to the second instance of the separator. A value of 3 indicates that the created variable should take everything from the second instance of the separator to the third instance of the separator, and so on for higher values.

If we use SCAN and the program gets to the end of the character variable before finding another instance of the separator variable, the SCAN function will still work. It will just take everything from the last separator variable to the end of the character variable.

In the example above, First_name is equal to the value of the name variable from the start to the first set of blanks. Middle_name is equal to the value of the name variable from the first set of blanks to the second set of blanks. Last_name is equal to the value of the name variable from the second set of blanks to the end. Since no third set of blanks exists, Last_name just takes all the characters to the end of the character variable.

If we had multiple blanks in a row, SCAN would have treated those the same way as if one blank were present. The same is true if we had used commas as the separator and we had multiple commas next to each other.

Let us add a complicating factor. Some people have more than one middle name, and some people do not have a middle name at all. How do we deal with this data set?

```
DATA names;
INPUT name $30.;
DATALINES;
John W. Smith
Dave T. Hernandez
Jean T. R. Yang
Ben Arlend
Elizabeth Duroska Jefferson
;
```

We would have to make some choices and do some complex programming to get the middle name. We can still get the first and last names without much difficulty, however. Fortunately, SCAN allows for negative values in the third argument, which starts the scan on the right instead of the left:

```
DATA names2;
       SET names;
       LENGTH first_name $30.;
       LENGTH last_name $30.;
       first_name=SCAN (name,1,'');
       last_name=SCAN (name,-1,'');
run;
```

The data set names2 would be as follows:

| Obs | name | first_name | last_name |
|---|---|---|---|
| 1 | John W. Smith | John | Smith |
| 2 | Dave T. Hernandez | Dave | Hernandez |
| 3 | Jean T. R. Yang | Jean | Yang |
| 4 | Ben Arlend | Ben | Arlend |
| 5 | Elizabeth Duroska Jefferson | Elizabeth | Jefferson |

As mentioned, SCAN does not have to work just with blank spaces as separators. Here is an example of the SCAN function working with comma-separated values. The third argument is now a comma:

```
DATA rocks;
       INPUT rock_list $30.;
       DATALINES;
       gold,silver,emeralds
       gold,slate
       gold,diamonds,jade
;
DATA rocks2;
       SET rocks;
       Rock1=SCAN(rock_list,1,',');
       Rock2=SCAN(rock_list,2,',');
       Rock3=SCAN(rock_list,3,',');
Run;
```

The data set rocks2 would be as follows:

| Obs | rock_list | Rock1 | Rock2 | Rock3 |
|---|---|---|---|---|
| 1 | gold,silver,emeralds | gold | silver | emeralds |
| 2 | gold,slate | gold | slate | |
| 3 | gold,diamonds,jade | gold | diamonds | jade |

This last example brings up another point. The variable Rock3 had no value for the second observation because only one comma was in the row. The second argument of Rock3 is 3, meaning Rock3 has the values of the characters from the second comma to the third comma. Since the second comma does not exist, Rock3 is null for that observation. SAS® will not produce an error in this case.

## THE VERIFY FUNCTION--CHECKING WHAT IS INSIDE A CHARACTER FUNCTION

The VERIFY function returns the first position in a character string that does not meet certain criteria as set by the user. The first argument is the variable name. The second argument is the list of characters that are acceptable data. VERIFY will return the first position that has a character that is not included as being acceptable data. If the character in the first position does not meet the criteria, the variable will return a 1. If the character in the first position meets the criteria but the character in the second position does not, VERIFY will return a 2, and so on for all values.

If all the characters meet the criteria, VERIFY will return a value of zero. Thus, checking for a value of zero can be a useful way to VERIFY the results. Unlike most exams, a value of 0 with the VERIFY function generally means "you passed!"

For example, let's say our aforementioned leprechaun wants to check if a few rock piles are all gold, as given by the letter G. The first argument in the example below is the variable "rocks." Since the leprechaun wants to check that everything is a G, the second argument will be just G.

```
DATA rocks;
INPUT rocks $ 1-3;
DATALINES;
GGG
EGG
GEF
;

DATA rocks2;
       SET rocks;
       check_for_gold=VERIFY(rocks,'G');
run;
```

The data set rocks2 will be as follows:

| Obs | rocks | check_ for_gold |
|---|---|---|
| 1 | GGG | 0 |
| 2 | EGG | 1 |
| 3 | GEF | 2 |

The first observation has three values of G, so all characters meet the criteria of G. The VERIFY function thus returns 0. In the second observation, the character in position 1 was not a G, so VERIFY returns a 1. In the third observation, both the characters in the second and third position were not G, so the VERIFY function returns a 2. It returns the position of the first character it finds that does not meet the criteria.

If our leprechaun was satisfied with having either emeralds (E) or gold (G), he could expand the criteria in the second argument:

```
DATA rocks2;
       SET rocks;
       check_for_emeralds_or_gold=VERIFY(rocks,'EG');
run;
```

The data set rocks2 would be as follows:

| Obs | rocks | check_for_ emeralds_or_gold |
|---|---|---|
| 1 | GGG | 0 |
| 2 | EGG | 0 |
| 3 | GEF | 3 |

In this case, the VERIFY function returns zeroes for the first two observations because the observations have only the values of G and E. In the third observation, the returned value is 3 because the third position has an F instead of a G or E. Had we made check_for_gold equal to VERIFY (rocks, 'EGF'), check_for_gold would have been a zero for all three observations.

A few functions exist that are really specialized versions of the VERIFY function. The NOTDIGIT function returns the first position of the character that is not equal to 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. NOTALPHA returns the position of the first character that is not a letter. NOTALNUM returns the first position that is not a letter or number.

## FINDING SUBSTRINGS IN CHARACTER STRINGS: THE FIND FUNCTION

A number of functions exist for finding strings within a character string. A substring is any continuous part of a character string. If a character string was 'GOLD' for example, 'G', 'GO', 'OL' , 'LD' and 'OLD' would all be examples of substrings.

If a programmer only learns one function for finding substrings, the FIND function is an excellent choice. The FIND function will return the first position of a substring within a character string.

In the following example, the leprechaun wants to know where to find the gold amidst a pile of rocks. In this example, find_gold gets a value of 5 because the substring "gold" starts at the fifth position.

```
DATA Find_gold;
INPUT rocks $ 1-12;
DATALINES;
coalgoldlead
;

DATA Find_gold_2;
       SET Find_gold;
       Find_gold=FIND(rocks,'gold');
run;
```

Like the VERIFY function, the FIND function will return a 0 if the string is not present.

What makes the FIND function very useful is that it allows for a third argument that gives conditions about the search. One of the options for this third argument is "i", which instructs SAS® to ignore case. Find_gold would also get a value of 5 in the following code:

```
DATA Find_gold;
INPUT rocks $ 1-12;
DATALINES;
coalGoLdlead
;

DATA Find_gold_2;
       SET Find_gold;
       Find_gold=FIND(rocks, 'gold','i');
run;
```

If we did not have an 'i' in that third argument, Find_gold would have returned a zero. SAS® would have looked for a lower-case 'gold' and not taken into account the substring 'GoLd'. But with the 'i' as the third argument, SAS® ignores case and treats 'gold' and 'GoLd' as equivalent.

## CHANGING PARTS OF CHARACTER STRINGS: THE TRANSLATE, COMPRESS, AND TRANWRD FUNCTIONS

TRANSLATE replaces one letter with another. Going back to our leprechaun example, imagine a leprechaun wants to turn all his emeralds (E) into gold (G). The letter E would get replaced by the letter G.

The TRANSLATE function takes three arguments: the variable name, the character (or characters) you want in the character string, and the character (or characters) you have in the character string:

```
DATA pile_of_rocks;
INPUT rocks $;
DATALINES;
GEGEGG
;

DATA pile_of_rocks_2;
       SET pile_of_rocks;
       LENGTH all_gold $6.;
       All_gold=TRANSLATE (rocks,'G','E');
run;
```

In the above example, the variable All_gold would have a value of GGGGGG. Each E (the third argument) gets turned to a G (the second argument.) While the example above does have a LENGTH statement, TRANSLATE is one of the functions where the new variable gets assigned the length of the original variable. Without the LENGTH statement, "All_gold" would have the same length as "rocks".

Alternatively, if we had to change more than one character, we could do so. Let's say fool's gold (F) was involved and the leprechaun wanted to change it to gold (G) as well:

```
DATA pile_of_rocks;
INPUT rocks $;
DATALINES;
GEGFGG
;
```

```
DATA pile_of_rocks_2;
       SET pile_of_rocks;
       All_gold=TRANSLATE (rocks,'GG','EF');
run;
```

The first character in the third argument (an E) gets changed to the first character in the second argument (a G). Likewise, the second character in the third argument (an F) gets changed to the second character in the second argument (also a G). If we ran this code, All_gold would have the value of GGGGGG.

A related case to the TRANSLATE function is the COMPRESS function. COMPRESS deletes unwanted characters. The COMPRESS function only has two arguments. The first argument is the variable and the second argument is the characters that are to be deleted. Let's say our leprechaun wants to get rid of the Emeralds (E) and Fool's Gold (F):

```
DATA pile_of_rocks;
INPUT rocks $;
DATALINES;
GEGFGG
;

DATA pile_of_rocks_2;
       SET pile_of_rocks;
       All_gold=COMPRESS (rocks,'EF');
run;
```

All_gold will have a value of GGGG.

With the COMPRESS function, the created variable takes the length of the original variable. "All_gold" would have the same length as "rocks." In an actual scenario, you may not want to set the LENGTH statement when using COMPRESS. You may have records where COMPRESS does not eliminate any characters because the characters in the second argument are not present.

We can also run COMPRESS with only one argument. In this case, SAS® will get rid of all blank spaces:

```
DATA pile_of_rocks;
INPUT rocks $ 1-6;
DATALINES;
G G GG
;

DATA pile_of_rocks_2;
       SET pile_of_rocks;
       LENGTH All_gold $4.;
       All_gold=COMPRESS (rocks);
run;
```

All_gold will have a value of GGGG. If we had not set the LENGTH statement, All_gold would have been assigned the same length as the variable rocks for a length of 6. Two blank spaces would have been added to the end of the record.

If we want to change strings of characters, we can use the TRANWRD function. With TRANWRD, we put what we have as the second argument and what we want as the third argument. (It is thus the reverse of the TRANSLATE function.) Let's say the leprechaun wanted to change lead to gold:

```
DATA pile_of_rocks;
INPUT rocks $ 1-20;
DATALINES;
GOLDLEADGOLDGOLDLEAD
;

DATA pile_of_rocks_2;
       SET pile_of_rocks;
       LENGTH all_gold $20.;
       All_gold=TRANWRD(rocks,'LEAD','GOLD');
run;
```

The value of All_gold would be GOLDGOLDGOLDGOLDGOLD. Setting the length beforehand is important when using TRANWRD. If we took out the LENGTH statement in the above example, the new variable "All_gold" will get the default length of 200.

As with most SAS® character functions, TRANSLATE, COMPRESS, and TRANWRD are case-specific. If we had a mix of cases, we might wish to use UPCASE, LOWCASE, or PROPCASE to get the values we want. These functions make everything either upper case, lower case, or the first letter of each word capitalized, respectively. Here is an example showing the upcase function used with the TRANWRD function:

```
DATA pile_of_rocks;
INPUT rocks $ 1-20;
DATALINES;
GOLDLEAdGOLDGOLDLeAD
;

DATA pile_of_rocks_2;
       SET pile_of_rocks;
       All_gold=TRANWRD(UPCASE(rocks),'LEAD','GOLD');
run;
```

Again, the result would be GOLDGOLDGOLDGOLDGOLD. With character variable functions (and many other functions), SAS® allows the programmer to put multiple functions in one line. The danger is that it makes the code hard to follow. Good documentation of the code is especially important when taking this approach.

## CONCATENATION

Concatenation allows the combination of two different character variables into a new variable. Let's say our leprechaun wants to combine piles of gold. Here, we can use any number of concatenation functions.

The first tool many SAS® programmers learn for concatenation is the || operator, which just concatenates two character strings together into one with no modifications:

```
DATA gold_piles;
INPUT gold_pile_1 $char3. gold_pile_2 $char3. ;
DATALINES;
GG  GG
 ;
DATA combined_gold_pile_result;
       SET gold piles;
       combined_gold_pile=gold_pile_1||gold_pile_2;
run;
```

The problem is that this approach does not get rid of blank spaces. The variable combined_gold_pile has the value GG  GG. The || operator is a good choice if you know that you have no leading or trailing blanks.

SAS® introduced a number of new functions in version 9 that allow for control of the type of concatenation. The CALL CATS function will strip out leading and trailing blanks. The syntax is that the new concatenated variable is the first argument, followed by all the variables that are to be concatenated. CALL CATS can concatenate many variables in a single step, although the example below just concatenates two variables:

```
DATA gold_piles;
INPUT gold_pile_1 $char3. gold_pile_2 $char3. ;
DATALINES;
GG  GG
 ;

DATA combined_gold_pile_result;
       SET gold_piles;
       LENGTH combined_gold_pile $4;
       CALL CATS (combined_gold_pile,gold_pile_1,gold_pile_2);
run;
```

The value of the record for combined_gold_pile will then be GGGG. The leading and trailing blanks from gold_pile_1 and gold_pile_2 get eliminated.

Using the LENGTH statement is important with the various concatenation functions. Without a LENGTH statement in the example above (or some other way to assign the length), the variable combined_gold_pile would have a null value.

While the example above shows two variables (gold_pile_1 and gold_pile_2) being concatenated, the CALL CATS function can handle multiple variables. If you wanted to concatenate ten different variables, for example, you could just use CALL CATS with 11 arguments: the variable you wanted to create followed by the ten variables you want to concatenate.

If you just want to get rid of the trailing blanks, you can use CALL CATT. The syntax is the same as CALL CATS. In this example, the length was increased by one to account for the one retained leading blank from gold_pile_2:

```
DATA gold_piles;
INPUT gold_pile_1 $char3. gold_pile_2 $char3. ;
DATALINES;
GG  GG
 ;

DATA combined_gold_pile_result;
       SET gold_piles;
       LENGTH combined_gold_pile $5;
       CALL CATT (combined_gold_pile,gold_pile_1,gold_pile_2);
run;
```

The result will be GG GG. The leading blank from Gold_pile_2 will still be included, but the trailing blank from Gold_pile_1 will be gone.

If you want to put in a separator between the variables, use CALL CATX. In this case, the separator is the first argument. The new variable is the second argument. All variables that are to be concatenated form the remaining arguments. This code uses the separator of '|'.

```
DATA gold_piles;
INPUT gold_pile_1 $char3. gold_pile_2 $char3. ;
DATALINES;
GG  GG
;

DATA combined_gold_pile_result;
```

```
        SET gold_piles;
        LENGTH combined_gold_pile $5;
        CALL CATX ('|',combined_gold_pile,gold_pile_1,gold_pile_2);
run;
```

The result will be GG|GG. As with CALL CATS, CALL CATT and CALL CATX can concatenate more than two variables at once.

## FUNCTIONS DEALING WITH BLANKS: COMPBL, STRIP, TRIM, AND LEFT

Blanks can be a problem when dealing with character variables. Data may have multiple blanks where we only want a single blank, or may have leading and trailing blanks that cause trouble.

If we want to turn multiple blanks between characters into single blanks, we can use COMPBL. That function can be useful for deleting excess blanks between words:

```
DATA reduce_blanks;
INPUT rocks $ 1-30;
DATALINES;
Gold
Lead    Emeralds
Gold  Jade Diamonds
Gold           Diamonds
;
DATA reduce_blanks_2;
        SET reduce_blanks;
        rocks_corrected=COMPBL(rocks);
run;
```

The variable Rocks_corrected will have the following value. The multiple blanks get reduced to a single blank. Here are the values of reduce_blanks_2:

| Obs | rocks | rocks_corrected |
|---|---|---|
| 1 | Gold | Gold |
| 2 | Lead  Emeralds | Lead Emeralds |
| 3 | Gold  Jade Diamonds | Gold Jade Diamonds |
| 4 | Gold  Diamonds | Gold Diamonds |

The STRIP function gets rid of both leading and trailing blanks. It will not affect blanks interspersed among other characters, however.

```
DATA reduce_blanks;
INPUT rocks $ 1-35;
DATALINES;
Gold Emeralds Lead
Gold
Lead Emeralds
 ;

DATA reduce_blanks_2;
    SET reduce_blanks;
    LENGTH rocks_corrected $ 18.;
    rocks_corrected=STRIP(rocks);
run;
```

The result will be that each record in rocks_corrected with start with a character with blank spaces to pad the length out to 18. (The first record's characters have a length of 18, which is why that length was chosen.)

If you just want to remove trailing blanks, you can you use the TRIM function. Without some statement to limit the length of the new variable, however, the TRIM function will do nothing. The reason is that the new variable will inherit the same length as the old variable and add blank spaces to the end. The result will be no change between the new and old variable. Using a LENGTH statement will stop this problem from happening.

If you want to eliminate leading blanks, use the LEFT statement. Unless the length of the new variable is limited in some ways (such as by a LENGTH statement), the blanks of the new variable will be moved to the end of the variable. However, that outcome may still be useful in some cases.

## CONCLUSION

A new SAS programmer will find their job much easier if they become familiar with basic character functions. Many problems in SAS can be solved by manipulating character strings.

Many other useful character functions exist. See http://support.sas.com/publishing/pubcat/chaps/59343.pdf

**ACKNOWLEDGMENTS**

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

**CONTACT INFORMATION**

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

Jason Ford
Bureau of Labor Statistics
Ford_j@bls.gov
(202) 691-6267