Paper 035-2007

Hanging On By a STRING? Using Functions To Untie Text Strings

Jeanina Worden, Pacific Data Designs, Inc., San Francisco, CA

ABSTRACT

Have you ever felt tied down, knotted up, and barely holding on by a STRING? Almost every data programmer can give an example of a variable that, due to data entry requirements, has to be characterized as a text string. They can also describe how problematic they are when, prior to analyzing the data, find these strings need to be cleaned up, formatted and sometimes compared. SAS® provides solutions through a "bundle" of functions you can use on strings. Some can be used to untangle the text by grabbing what is needed, or dropping out what is not needed, like: SCAN, and COMPRESS. While others can evaluate the contents of the string and make a comparison to a constant or another variable, such as: LENGTH, SPEDIS and COMPARE. The functions in this paper are of several levels of difficulty, but can all be used within SAS® version 9.1.

INTRODUCTION

One might think the problem with strings in a dataset is they can contain 1 to 32767 characters, numbers (as characters) and symbols, and that the combinations are endless. But the "problem" is that strings aren't just strings of characters but an exact string of characters. Text strings are to SAS®, an exact combination of characters, in an exact case, in an exact order, so "String" is not the same as "string". Therefore when trying to work with the strings within these character fields, SAS requires you to be exact as well. As a data programmer, when QCing strings, one way to identify problematic strings is by using one or more of the functions SAS® provides. Determining the "right" function to unravel a text string depends, first, on what you are trying to find out.

In this paper some common issues will be discussed:

- Determining the completeness of a delimited date
- Determining the completeness of a phone number
- Determine length of a text string
- Identify misspellings or synonyms of a given string
- Compare two variables

And how the use of these SAS® functions can unravel strings:

- SCAN
- COMPRESS
- LENGTH
- SPEDIS
- COMPARE

WARNING: STRINGS ATTACHED

Before looking at the functions you must put yourself in the role of a data programmer working in the world of pharmaceutical data. With the understanding that, with this data come certain rules, one being that the FDA requires raw data to be entered as it is given. That means if Site A fills out the field for STATE as CA, and Site B fills in Ca, data entry must key strings into the "raw" database exactly as they appear. As was said, these two strings mean completely different things to SAS®, and since the data must be in a format that is consistent and usable for analysis, some sort of manipulation will have to take place. Of course knowing the data well enough to know when there is a problem is critical to completing string related tasks. And while there are several ways in which these functions can be used, the focus of this paper will primarily be to summarize or highlight the issues with strings. We'll start by using a small table of subjects with each variable entered in different formats. While this is an exaggeration, it does represent some issues that may appear in the raw datasets.

Table 1

Lname	Start_dt	Phone	CellPhone	Comments	MedHx	Quest1_v1
Doe	1/1/01	(555) 867-5309	(555) 867-5309	Some sort of comment	Hpertension	Yes

Smith	01/01/2001	5558675309	456789	Another sort of comment	Hypertensive	Yes
Jones	01/UNK/01	555 867 5309	123 456 789	Another comment but a little longer	CHF	N

STRINGS HAPPEN

Lets start by considering the date field, which is START_DT in the Table 1 dataset. Any date is a tricky string since, in most cases, it is actually three strings tied together; the part for the month, day and year. So, in an attempt at uniformity, the form designer has taken great care in providing the delimiter of which should be applied to the date, however, in our example, "strings happen". What if you want to untie these pieces to verify the presence of each piece? The multiple lengths between each delimiter limit your options, with one being the SCAN function. The SCAN function, simply put, pulls out the pieces between the delimiter without regard to the length of each piece. The syntax is:

```
SCAN(string, n<, delimiter(s)>)
```

Here's how it works. The STRING portion of the statement is self-explanatory as it specifies the variable name that contains the string. The delimiters, if used, are the characters you want to use to separate the "words" in a string. So, if you have a string with "one|two|three", and use the '|' as your delimiter, the "words" in the string are "one", "two" and "three". This helps to understand how N represents the number of the "word" in the string you want to scan. So if you were to use '2' as the N, the scan function returns "two", the second "word" from the delimiter. An especially handy feature of this function is that you can also use negative N values. When using a negative N, SCAN returns the "word" with the count starting from the left. For this example if you use '-1' as the N the SCAN function will return "three". Let's look again at our sample date variable.

To get the strings into the three individual parts, the code looks like this:

```
data test2;
set test;
mm = scan(start_dt,1,'/');
dd = scan(start_dt,2,'/');
yy = scan(start_dt,3,'/');
run;
```

Just to demonstrate the negative N, the code could also look like this:

```
data test2;
set test;
mm = scan(start_dt,1,'/');
dd = scan(start_dt,2,'/');
yy = scan(start_dt,-1,'/');
run;
```

This example shows, using the "/" delimiter, the month is the first word from the left, so we use a positive N and the year is the first word from the right.

With the resulting columns:

From:	**To:**			
Start_dt	MM	DD	YY	
1/1/01	1	1	01	
01/01/2001		01	01	2001
01/UNK/01		01	UNK	01

One last note on SCAN, two or more contiguous delimiters only count as one in determining the "word" SCAN will return. So if a date such as "01//2001" appears in the dataset, the string for DD will be 2001.

PULL THE RIGHT STRING

Next we have the variables PHONE and CELLPHONE, how can we determine if the full phone number is present without them being entered in the same format? The easiest way to get a common format is to

remove the spaces and special characters that divide the number. Up to and including version 9.1, COMPRESS has been a valuable tool used frequently to remove unwanted characters. To show how to cut the unwanted characters from the string variable PHONE, we start with the syntax:

```
COMPRESS(<source>, <chars>)
```

SOURCE specifies the variable in which the string resides. CHARS are the characters you want to remove. So, the unwanted "()" and "-", as well as the spaces, are removed from the string, with the following code:

```
data test2;
set test;
phone = compress(phone,"()- ");
run;
```

With the resulting columns:

From:	**To:**
Phone	Phone
(555) 867-5309	5558675309
5558675309	5558675309
555 867 5309	5558675309

It's worth noting, if no CHARS are specified the default results in only the space being removed.

That's pretty straight forward, but the 9.1 upgrade to this function really boosts the power of COMPRESS with a third argument, the modifiers.

```
COMPRESS(<source>, <chars><,modifier>)
```

As with the above example, with no MODIFIER the characters in the CHARS list are removed. However the MODIFIERS can be used in two ways. The first is that the modifiers can be used to expand the CHARS list with a group of characters. For instance, if you wanted to ensure all punctuation and spaces (including blank, tabs, lf, cr) were removed from the PHONE variable, but didn't want to list every possibility, "P" can be added to specify the removal of punctuation and "S" specifies spaces.

```
data test2;
set test;
phone = compress(phone, ," ps"),;
run;
```

Resulting in the same columns as above:

From:	**To:**
Phone	Phone
(555) 867-5309	5558675309
5558675309	5558675309
555 867 5309	5558675309

The second way to use the new modifiers is to instruct COMPRESS to keep the CHARS list. With the MODIFIER "K" the characters specified within the CHARS list are kept rather than removed. Like this:

```
data test2;
set test;
cellphone = compress(cellphone,'0123456789',"k"),;
run;
```

or (using just modifiers)…

```
data test2;
set test;
cellphone = compress(cellphone,'','dk");
run;
```

Resulting in the same columns as above:

From:	**To:**
Phone	Phone

(555) 867-5309	5558675309
456789	456789
123 456 789	123456789

The additional modifiers that can be used are as follows with more specific information in the COMPRESS documentation (modifiers are not case sensitive):

A: adds letters of the alphabet (A - Z, a - z) to the list of characters
C: adds control characters to the list of characters
D: adds numerals to the list of characters.
F: adds the underscore character and letters of the Latin alphabet (A - Z, a – z) to the list of characters.
G: adds graphic characters to the list of characters.
I: ignores the case of the characters to be kept or removed.
K: keeps the characters in the list instead of removing them.
L: adds lowercase letters (a - z) to the list of characters.
N: adds numerals, the underscore character, and letters of the Latin alphabet (A - Z, a - z) to the list of characters.
O: processes the second and third arguments once rather than every time the COMPRESS function is called.
T: trims trailing blanks from the first and second arguments.
U: adds uppercase letters (A - Z) to the list of characters.
W: adds printable characters to the list of characters ("W" for writable, because "P" is used for punctuation).
X: adds hexadecimal characters to the list of characters.

STRINGS ON AND ON

Now we move on to the COMMENT field. Let's suppose this variable will be used in a listing with space for only 25 characters, will they all fit? To determine the length of these strings we can use, surprisingly enough, the function LENGTH. The LENGTH function returns an integer that represents the position of the rightmost non-blank character in string. The syntax for LENGTH is:

```
LENGTH(string)
```

Where STRING specifies the variable with the string to be verified. So to get the length of COMMENT we could use:

```
data test2;
set test;
len = length(comment);
run;
```

Resulting in:

Variable:	**Length Variable:**
COMMENT	LEN
Some sort of comment	20
Another sort of comment	23
Another comment but a little longer	35

One caveat to using LENGTH is that, though it will not count trailing blanks, if the whole field is blank it will return a 1 not a 0.

THE HIGH COST OF STRINGS

Up next is the MEDHX variable. Can we identify any records containing "HYPERTENSION" for a use in a comparison, merge, or so on? If you look at the records in the Table 1 dataset, there are none equal to "HYPERTENSION". Since strings need to be exact when doing any sort of matching, we need a way to identify if two strings are similar to try to correct any misspellings or make adjustments to synonyms. Though unlikely to be used for merging or matching two strings directly, the function SPEDIS can be used to identify misspellings and similar strings so that further action can be taken where necessary. SPEDIS computes an asymmetric spelling distance between two. For each operation the conversion of the keyword

to the query word must go through (delete a letter, add a letter, and so on) there is a cost associated. The distance, or total cost, is then calculated by dividing the conversion cost by the length of the query string. The documentation for SPEDIS lists each operation and its cost:

Operation	Cost	Explanation
match	0	no change
singlet	25	delete one of a double letter
doublet	50	double a letter
swap	50	reverse the order of two consecutive letters
truncate	50	delete a letter from the end
append	35	add a letter to the end
delete	50	delete a letter from the middle
insert	100	insert a letter in the middle
replace	100	replace a letter in the middle
firstdel	100	delete the first letter
firstins	200	insert a letter at the beginning
firstrep	200	replace the first letter

To more clearly explain the concept of SPEDIS, here is a simple example:

The conversion from SUGI to SGF :

Start with **SUGI**…
Delete U (delete a letter in the middle) = 50
Delete the I (delete a letter from the end) = 50
Add the F (add a letter on the end) = 35
…to get **SGF**
Cost = 135

The distance, or the result of running the SPEDIS function is the Cost divided by the length of the keyword:
Distance = (135/3) = 45

The syntax for the SPEDIS function is:

```
SPEDIS(QUERY,KEYWORD)
```

So for our example the statement would be:

```
SPEDIS('SUGI','SGF')
```

The code for the MEDHX variable would be:

```
data test2;
set test
mhxcomp = spedis(propcase(MedHx),'Hypertension');
run;
```

Resulting in:

Variable:	Distance Variable:
MedHx	Mhxcomp
Hypertension	0
Hypertensive	16
CHF	266

The total cost of the conversion is a nonnegative value that is usually less than 100 but never greater than 200 with the default costs. So, if we want to identify similar strings, the lower the number the better. With this information we can list out the strings within a range that ensures the specificity we are comfortable with. Then, if misspellings are identified, corrections can be made, and "similar" strings can be coded accordingly.

IT'S THE SAME STRING

The last function to be covered will help determine the similarity between two response variables using the COMPARE function. COMPARE returns the position of the leftmost character by which two strings differ, or returns 0 if there is no difference. So let's look at the last variable in the dataset, Quest1. Let us suppose this question is something like, "Have you ever smoked?" We have a second dataset with the same questionnaire given at a second visit. We merge the information together for the two visits into Table 2:

Table 2

Quest1_v1	Quest1_v2
Yes	Yes
y	Yes
N	Yes

We want to identify if the responses are consistent between the two questionnaires. While this is similar to the previous task with SPEDIS, let's look at the cost differences:

Variable:	**SPEDIS results:**	
Quest1_v1	Quest1_v2	Distance
Yes	Yes	0
y	Yes	150
N	Yes	90

Our previous example of SUGI to SGF gave 45 as the result of two quite different strings, so we would use a cutoff lower than 45, right? Based on the cutoff of 20, only the first would be identified as the same and if the cutoff was "loosened" to accommodate the second record then the third would also get flagged. COMPARE, on the other hand, has modifiers that can be applied to help with the formatting issues.

You can use one or multiple modifiers (they are not case sensitive):

I = ignores the case in string1 and string2

L = removes leading blanks in string1 and string2 before comparing the values

N = removes quotation marks from any argument that is an n-literal and ignores the case of string1 and string2

: = truncates the longer of the two strings to the length of the shorter string or to one, whichever is greater

The syntax for the COMPARE function is:

```
COMPARE(string-1,string-2<,modifiers>)
```

For our example the code would be:

```
data test2;
set test
RespComp = compare(quest1_v1,quest1_v2,':il');
run;
```

Resulting in:

Variable:	**Length Variable:**	
Quest1	Quest2	RespComp
Yes	Yes	1
y	Yes	1
N	Yes	

CONCLUSION

So instead of spending a bundle of time sorting through strings let SAS® functions help. This paper detailed functions that could aid in untying troublesome string variables. Using them we can grab what is needed, drop out what is not needed, or just evaluate the contents of the string and make a comparison to a constant or another variable.

First we discussed how SCAN could help split a date into three parts: one for month, one for day and one for year, to verify each pieces presence. Then, a review of how COMPRESS can be used to strip away the special characters and spaces in a phone number and how it can also be used to keep the digits either listed or with a modifier. LENGTH was used to determine if a string would fit if used in a 25-character space. SPEDIS was used to flag records that had a medical history with an exact or similar match to "HYPERTENSION". Finally, COMPARE was used to identify if two character strings represent the same value though differing formats.

REFERENCES

SAS Institute Inc. 2004. SAS® 9.1 Language Reference: Dictionary, Volumes 1, 2,and 3. Cary, NC: SAS Institute Inc.

Cody, Ronald (2006), "An Introduction to SAS® Character Functions," Proceedings of the Thirty-One Annual SAS Users Group International Conference.

ACKNOWLEDGMENTS

The programming team at Pacific Data Designs, Inc. contributed extensively to the development of this paper. Their support is greatly appreciated.

CONTACT INFORMATION

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

Jeanina Worden
353 Sacramento, Ste 800
San Francisco, CA 94111
Phone: 415-776-0660
E-mail: jworden@pdd.net