

Data Step Functions in SAS® 9

Bob Newman, Amadeus Software, UK

INTRODUCTION

SAS® 9 introduces several new functions and several enhancements to existing functions within the data step. We think many of these are quite useful enhancements and will save us some coding conundrums in the future!

NEW CHARACTER FUNCTIONS

FUNCTIONS THAT SEARCH FOR CLASSES OF CHARACTERS

There are a lot of these, and they come in pairs. There's an ANY function, which searches the target string and locates the first character which belongs to the class; and there's a NOT function to locate the first character which doesn't belong to the class, e.g.

```
p=anylower('String');
```

sets p to 2, the position of the first lower case character in the string. An optional second parameter allows you to specify the starting position for the scan and the direction of scan e.g.

```
q=anyupper('String',-6);
```

sets q to 1, having started at position 6 and scanned the string in reverse.

If no matching character is found, or you specify a starting position of zero, then the value returned is zero.

Here's a table summarising all the functions of this type.

"ANY" function	"NOT" function	Characters Considered
ANYALNUM	NOTALNUM	0-9 a-z A-Z
ANYALPHA	NOTALPHA	a-z A-Z
ANYCNTRL	NOTCNTRL	Control characters
ANYDIGIT	NOTDIGIT	0-9
ANYFIRST	NOTFIRST	Characters valid as first character in a SAS variable name under VALIDVARNAME=V7, i.e. a-z A-Z and underscore
ANYGRAPH	NOTGRAPH	Printable characters other than white space. (Definition varies depending on TRANTAB, ENCODING and LOCALE system options.)
ANYLOWER	NOTLOWER	a-z
ANYNAME	NOTNAME	Characters valid in a SAS variable name under VALIDVARNAME=V7 i.e. a-z A-Z 0-9 and underscore
ANYPRINT	NOTPRINT	Printable characters (including white space). (Definition varies, as for ANYGRAPH).
ANYPUNCT	NOTPUNCT	Punctuation (Definition varies, as for ANYGRAPH).
ANYSPACE	NOTSPACE	White space (including TAB, VT, CR, LF, FF)
ANYUPPER	NOTUPPER	A-Z
ANYXDIGIT	NOTXDIGIT	0-9 a-f A-F

FIND AND FINDC FUNCTIONS

FIND is an enhanced version of INDEX. It has two optional parameters; without them, it is identical with INDEX. Here are some examples of the use of FIND:

```
a=find('Sesquipedalian','quip');
```

As INDEX would, returns a value of 4.

```
b=find('Cardamoms','mo',-9);
```

We specify a starting position. This one starts at position 9 and processes the string in reverse (as specified by the minus sign). Note though that the substring still has to be found in a forward direction, so we find our match at position 6. (A start position of zero is valid but will always fail to find a match and so return a value of zero.)

```
c=find('Cardamoms ','car ','it');
```

This time we specify an optional modifier string, containing the only two modifiers available. “i” means “case insensitive” and “t” means “trim both string and substring”. So we find a match at position 1.

```
d=find('Cardamoms','MO ',4,'it');
```

You can specify both of the optional parameters. Here we scan forwards from position 4 with no sensitivity and all the trimmings, and find a match at position 6.

```
e=find('Cardamoms','MO ','it',4);
```

Specifying the optional parameters in the opposite order also works, and gives the same result. (One is numeric, the other character.)

Similarly FINDC is an enhanced version of INDEXC. Or you can regard as a do-it-yourself ANY or NOT function. Compared with FIND, FINDC supports two additional modifiers. Here are some examples:

```
a=findc('Cabbage White','abc ',-10);
```

Looks for a, b, c or space, starting at position 10 and working backwards. Finds the space, so returns a value of 8.

```
b=findc('Cabbage White','abc ',-10,'t');
```

As before, but here the “t” modifier means we ignore the space and return a value of 5.

```
c=findc('Cabbage White','abc ','oiv');
```

Here we specify both of the new modifiers. “v” means NOT, so we match the first character which is not an a, b or c, or a space. Since matching is now case-insensitive (‘i’), this is the “g”, so the value returned is 6.

The ‘o’ should be used with caution. It means “compile once only”. On subsequent iterations of the data step, neither the substring nor the position nor the modifiers will be recompiled; only the target string can change. If the values of the substring or the position or the modifiers do change, these changes will be ignored. Where this will cause no problems, it will obviously improve efficiency. (Evidently there’s an RXPARSE or PRXPARSE going on in the background!)

CAT FUNCTIONS

There are four breeds of CAT, for concatenating any number of character arguments. They differ in how they handle white space and separators. This table (courtesy of SAS Institute) reveals all.

Function	Equivalent Code
CAT (OF X1-X4)	X1 X2 X3 X4

Function	Equivalent Code
CATS (OF X1-X4)	TRIM(LEFT(X1)) TRIM(LEFT(X2)) TRIM(LEFT(X3)) TRIM(LEFT(X4))
CATT (OF X1-X4)	TRIM(X1) TRIM(X2) TRIM(X3) TRIM(X4)
CATX (SP, OF X1-X4)	TRIM(LEFT(X1)) SP TRIM(LEFT(X2)) SP TRIM(LEFT(X3)) SP TRIM(LEFT(X4))

COMP FUNCTIONS

There are three new functions for comparing pairs of strings.

COMPARE is the only one likely to be much use to us. Essentially, the value returned is the first character position at which the two strings differ. The sign is used to tell us which of the two comes first in a sorting sequence. There are optional modifiers to do with leading blanks, case insensitivity, quoted strings, and truncating to the shorter length; sometimes these options interact in interesting ways.

One simple example:

```
d=compare('Stationary','stationery','il');
```

The comparison is case-insensitive ('i') and ignores leading spaces ('l'). The value returned is -8, the minus sign indicating that the first string comes first alphabetically.

COMPGED calculates the “general edit distance” between two strings. This is a measure of how likely one is to be a misprint for the other. It’s enough to know that this function is there!

COMPLEV calculates the “Levenshtein edit distance” between two strings. This is a special case of the general edit distance. COMPLEV will run more quickly than COMPGED and possibly be easier to understand (though there’s a CALL COMPCOST you’ll need to get to grips with as well). But not yet...

COUNT FUNCTIONS

COUNT counts the number of times a substring occurs in a given string. It has the usual optional modifiers 'i' (case insensitive) and 't' (trim). Example:

```
c=count('Love is the plan. The plan is death','the','i');
```

This returns a value of 2. The documentation warns that overlapping occurrences are liable to give “inconsistent results”, so don’t go doing things like:

```
x=count('mamamamamamamamamamama','mama');
```

COUNTC counts the number of characters in a string that come from a user-defined class; it’s a close relative of FINDC, and can take all the same modifiers. Example:

```
k=countc('3.14159','12345','vo');
```

returns a value of 2 – the decimal point and the 9 being the two non-matching ('v') characters. Note the 'o' modifier – “don’t recompile”. More regular expressions in the background...

LENGTH FUNCTIONS

There are three new LENGTH functions. Recall that the original LENGTH function returns the length of a character variable, excluding trailing blanks, and returning a length of 1 if the variable is missing.

LENGTHC returns the length *including trailing blanks*.

```
len=lengthc("Technically this has trailing spaces   ");
```

returns a value of 40.

LENGTHN returns the length excluding trailing blanks, *returning 0 for a blank string*.

```
len=lengthn("    ");
```

returns 0.

LENGTHM returns the amount of *memory (in bytes) allocated* to the string.

```
len=lengthm("    ");
```

return 4 as there are four spaces and hence bytes reserved for the string.

NUMERIC FUNCTIONS

MATHEMATICAL AND STATISTICAL FUNCTIONS

BETA and LOGBETA handle the Beta function (which is related both to the Beta probability distribution and to the Gamma function!)

MEDIAN (syntax as for MEAN, GEOMEAN and HARMEAN) returns the median of all the non-missing arguments.

MAD (same syntax) returns the median absolute deviation of the median – a perfectly rational thing for you to want.

IQR (same syntax) returns the interquartile range.

PCTL calculates percentiles e.g.

```
n=pctl(25,1,2,3,4,5,6,7,8,9);
```

calculates the 25th percentile of the other 9 parameters, returning the answer 3. There are 5 different definitions of percentile – see the documentation of PROC UNIVARIATE – and 5 different functions: PCTL1, PCTL2, PCTL3, PCTL4, PCTL5. PCTL is equivalent to PCTL5.

LARGEST and SMALLEST select the value from the desired position in a list e.g.

```
n=largest(3,1,2,3,4,5,6,7,8,9);
```

asks for the third-largest of the other 9 parameters, returning the answer 7.

UNFUZZY FUNCTIONS

CEILZ, FLOORZ, INTZ, MODZ, ROUNDZ are equivalent to the old functions without the Z, but they don't do zero fuzzing – the old functions do. The difference only matters for values within 1E-12 of a critical value. For example, for a value that is very, very slightly *less* than 2.5, ROUND will give a value of 3, whereas ROUNDZ will give 2. I find this stuff confusing! I recommend the documentation of ROUNDZ.

The new ROUNDE handles fuzzing in the same way as ROUND, but behaves differently for values exactly half-way between two integers.

VARIABLE FUNCTIONS

VVALUE returns the formatted value of a variable i.e. it's like using the PUT function with the variable's own default format. The argument to VVALUE must be a variable; it can't be an expression. If you want to specify an expression, use VVALUEX. So:

```
ns=vvalue(n);  
nt=vvaluex('n');
```

both give the same result: a text string containing the current value of variable n, suitably formatted.

NEW CALL ROUTINES

Some of these have already been mentioned, or correspond to functions already mentioned, or to functions that existed before V9: CATS, CATT, CATX, COMPCOST, SCAN, SCANQ.

PERMUTATIONS

CALL RANPERM generates a random permutation of N values you specify.

CALL RANPERK generates a random permutation of any K out of N values you specify.

CALL ALLPERM is similar, but not random. It returns one permutation at a time, but a sufficient number of calls will return all possible permutations. The FACT function will tell you how many calls you need.

OTHER CALL ROUTINES

CALL STREAMINIT is used in conjunction with the RAND function for generating streams of pseudo-random numbers.

CALL STDIZE is a statistical routine for standardising a set of values. There are *many* options.

CALL SYMPUTX is a variant of CALL SYMPUT that removes both leading and trailing blanks.

CALL VNEXT enables you to process each variable in a dataset in turn. Each call to VNEXT returns the name of a variable, and optionally its type and/or length. The variables returned include:

- All standard variables
- Automatic variables such as _N_ and _ERROR_
- FIRST.variables and LAST. variables (when there is a BY statement)
- Variables used in calling VNEXT

The order in which variables are returned is not specified and is liable to vary between operating systems and versions of SAS.

ENHANCEMENTS TO V8 FUNCTIONS

COMPRESS

COMPRESS now has an optional third argument that greatly increases its power. It is a list of modifiers, among which “k” specifies “keep”, reversing the normal behaviour of the function. Most of the other modifiers specify categories of character – corresponding to the various ANY and NOT functions – to be added to the list specified by the second parameter. There is also the “O” modifier, meaning “compile once only”. Once again there are regular expressions in the background here.

Take care with this one - the effect of a missing or null second argument seems a bit odd. Some examples:

```
x=compress('Who is Sylvia?',, 'P');
```

This removes the punctuation, returning “Who is Sylvia”. I’d expect it to remove the spaces too, but it doesn’t. To do that, specify either of the following:

```
x=compress('Who is Sylvia?',, 'SP');  
x=compress('Who is Sylvia?',, 'P');
```

Using the “keep” option:

```
y=compress('Who is Sylvia?',, 'UK');
```

This retains upper case letters only, returning ‘WS’.

```
z=compress('Who is Sylvia?',, 'UK');
```

This retains upper case letters and spaces, returning 'W S'.

CHARACTER MATCHING USING PERL REGULAR EXPRESSIONS – THE PRX FUNCTIONS

The PRX family of functions and CALL routines were introduced in SAS 9. They are a powerful set of tools for character matching and substitution. Broadly similar functions are provided by the RX family, which have been around since the later releases of SAS V6; however the RX functions use "SAS regular expressions" (which have a charm all their own), whereas the PRX functions use "Perl regular expressions" which have a standard form likely to be familiar to Unix programmers.

The last word on Perl regular expressions is at <http://www.perl.com/doc/manual/html/pod/perlre.html>. Not all the features documented there are supported by SAS; however SAS does appear to support more features than are covered by the SAS documentation.

The basic structure of a data step that uses PRX functions will be something like:

- On initialisation, use PRXPARSE to parse a Perl regular expression
- For each observation in the dataset, process strings using some of the following (of which the first three are the important ones):
 - PRXMATCH reports whether the target string contains a match for the expression, and if so where
 - PRXSUBSTR is similar to PRXMATCH, but also reports the length of the matching substring
 - PRXCHANGE, in conjunction with an appropriate expression, enables the target string to be changed, by substituting one string for another
 - PRXNEXT enables you to look for the same pattern, or make the same change, several times in the same target string
 - PRXPAREN – where an expression allows for alternatives e.g. `/(dog)|(cat)/`, you can use this to find which of the alternatives gave the match. The value returned in here would be either 1 (dog) or 2 (cat)
 - PRXPOSN can be used to identify, by position and length, the parts of the target string that matched particular patterns within the regular expression
- On termination, use PRXFREE to release the memory used for storage of the parsed regular expression. (The sky rarely falls if you forget to do this.)

There is also a PRXDEBUG function, which we do not expect to need!

A simple example:

```
data _null_;
  set mypets;
  retain prx;
  if _N_ = 1 then prx = prxparse("/(cat|dog|goldfish)/");
  call prxsubstr(prx, mystring, position, length);
  if position ^= 0 then
  do;
    match = substr(mystring, position, length);
    put match:$QUOTE. "found in " mystring:$QUOTE.;
  end;
run;
```

Perl regular expressions for matching are always bracketed by a delimiter character - normally `/`, but you can use any character not appearing elsewhere in the expression. Substitution expressions used with PRXCHANGE take the form:

```
"s/from-string/to-string/"
```

SIMPLE MATCHING

`/a/` matches the first "a" in the target string. `/^a/` will match an "a" only if it is at the beginning of the string. `/a$/` will match only at the end of the string.

The "escape" character is `\`; this indicates that the next character is not special and is to be taken literally so:

```
prx = prxparse("/\\\/"); matches a single backslash.
```

A dot matches any single character, so

```
prx = prxparse("/a.e/");
```

 will find a match in the string "mistaken" at position 5.

Alternative characters are enclosed in square brackets, so

```
prx = prxparse("/[ai].e/");
```

 matches "literate" at position 2. "^" is used to invert selections, so

```
prx = prxparse("/[^aeiou]/");
```

 matches the first character which is not a vowel.

```
prx = prxparse("/[\\^aeiou]/");
```

 matches the first character which is either a vowel or a circumflex.

Ranges are indicated by a hyphen, so

```
prx = prxparse("/[a-eg]/");
```

 matches the first character which is "a", "b", "c", "d", "e", or "g". By default all this matching is case sensitive. You can append an "i" to make it case insensitive so:

```
prx = prxparse("/a/i");
```

 matches the first character which is either "a" or "A".

CHARACTER CLASSES

Here we can match patterns of characters... not so simple (but rather effective) matching!

```
prx = prxparse("/\\d/");
```

 matches the first digit, and

```
prx = prxparse("/\\D/");
```

 the first non-digit. Similarly

```
prx = prxparse("/\\w/");
```

 matches the first "word character" and

```
prx = prxparse("/\\W/");
```

 matches the first "non-word character", word characters being defined as a-z, A-Z, 0-9 and underscore.

```
prx = prxparse("/\\s/");
```

 matches the first white space character, and

```
prx = prxparse("/\\S/");
```

 the first non white space character.

```
prx = prxparse("/\\t/");
```

 or

```
prx = prxparse("/\\x09/");
```

 matches the first tab character; the latter syntax can also be used to specify other characters by hex code.

WORD BOUNDARIES

How to control when is the end of your search source really the end... or not the end? Perhaps if you're sucked into this paper you wondered why you started?

```
prx = prxparse("\\b");
```

 indicates "word boundary" and

```
prx = prxparse("\\B ");
```

 not a word boundary" so

```
prx = prxparse("\\ba/");
```

 matches "Man alive!" at position 5, and

```
prx = prxparse("\\ba\\B/");
```

 matches "a bad troika accident" at position 14 (where the "a" begins a word but does not end it).

ALTERNATIVE STRINGS AND SUBSTRINGS

Use "|" to indicate alternatives :

```
prx = prxparse("/one|two|three/");
```

 matches any of the three strings.

If you want to use PRXPAREN to find which string gives the match, you must make them patterns by putting them in brackets thus:

```
prx = prxparse("/(one)|(two)|(three)/")
```

```
prx = prxparse("/(one)|((?i)two)|(three)/")
```

 specifies case insensitivity for “two” only, so matches “TWO” or “Two” but not “ONE” or “THREE”.

```
prx = prxparse("/(thir|four|fif|six|seven|eigh|nine)teen/")
```

 would match strings such as “fifteen”; PRXPAREN would return the value 1, since the match was in the first set of brackets.

REPETITION

Use “*” for “zero or more times”, “+” for “one or more times”, “?” for “zero or one times”, “{n}” for “exactly n times”, “{n,}” for “n or more times”, “{m,n}” for “between m and n times”.

Beware of “*” finding zero-length matches.

```
prx = prxparse("/s*/");
```

 will find a match in “headmistressship” – at position 1, and of length 0. Since this is the first match it finds, this is the one it will report.

```
prx = prxparse("/s+/");
```

 will match the first “s”, and “/s{2,}/” will find the “sss”.

To find, for example, the second occurrence of something, use PRXNEXT.

MINIMAL REPETITION

This is a nice Perl feature not currently documented by SAS. By default, the match returned is the longest one possible from the earliest possible starting position, so

```
prx = prxparse( "/a.*a/" );
```

will match “capybara”, finding the string “apybara”. Minimal repetition uses “*?” meaning “as few times as possible”, so

```
prx = prxparse( "/a.*?a/" );
```

finds the *shortest* earliest matching string, here “apyba” (and not “ara”, which comes later).

REPETITION USING PATTERNS

You can use “\1”, “\2” etc to refer to patterns previously matched, but be careful – you are specifying not the patterns themselves, but the strings that matched them. Thus

```
prx = prxparse("/(cat|mog).*\1/");
```

would *not* find a match in “transmogrification”.

The first time the pattern “(cat|mog)” matches, it matches “mog”, so “\1” is “mog”. The subsequent “cat” does not therefore match “\1”. “\b(.).\1” matches words such as “baboon” and “cockerel” where the third character is the same as the first.

SUBSTITUTION USING PATTERNS

In specifying the “to” string, you can refer to the patterns matched as “\$1”, “\$2” etc (NB in the “from” string, they are “\1”, “\2” etc.) This enables you to swap parts of the match around.

A useful (if perhaps intimidating) expression is

```
prx = prxparse("s/(\d{1,2})\ /(\d{1,2})\ /(\d{4}|\d{2})/$2\/$1\/$3/");
```

which converts between UK and US date formats.

You can make this more readable by appending an “x”, which makes it an “extended regular expression”, the advantage of which is that you can insert spaces in the “from” string (though NB still not into the “to” string):

```
prx = prxparse("s/ (\d{1,2}) \ / (\d{1,2}) \ / ( \d{4} | \d{2} ) /$2\/$1\/$3/x");
```

Like the original expression, this would change e.g. "12/6/1953" to "6/12/1953".

LOOKAHEAD

This is a form of extended pattern not documented by SAS.

```
prx = prxparse("/a(?=r)/");
```

looks for an "a" followed immediately by an "r" – but the "r" does not count as part of the match. So on "copybara", we will find a match of length 1 at position 6.

NEGATIVE LOOKAHEAD

```
prx = prxparse("/a(?![pr])/");
```

looks for an "a" which is NOT followed by "p" or "r", so will find a match of length 1 in "copybara" at position 8.

Lookbehind and *negative lookbehind* are also standard Perl features, but these do not appear to be currently supported by SAS.

CONTACT INFORMATION

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

Bob Newman
Amadeus Software Ltd.
Orchard Farm
Witney Lane
Leafield
OX29 9PG
Work Phone: +44 (0)1993 878287
Email: bob.newman@amadeus.co.uk
Web: www.amadeus.co.uk

Copyright (©) 2005 Amadeus Software Ltd.

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.

Other brand and product names are trademarks of their respective companies.