Creative Uses of Functions

Steven A. Wright
Quintiles, Inc.

A comprehensive knowledge of the appropriate SAS® functions can often make the difference between a difficult and a fun programming task. My goal is to demonstrate some useful applications of a number of my favorite functions.

Some items covered include:

- String manipulations
- Date and time issues
- Sampling methods
- Mathematical tips

Introduction

Studying SAS functions will give you more return on your investment than many other endeavors. This is true because for most data manipulations some function is needed. Familiarity with these tools can make programming much easier.

One of the best places to start is in the section in the front of the Language Guide. Here the functions are grouped, and listed with a brief description of their functionality. The same information is also available in the online help, but I think the book makes for better perusing.

Rather than just go over a number of functions, I want to explore some interesting uses of a subset of these that I have found useful over the years. I will also cover some other areas related to the use of functions.

String manipulations

Partial dates

I thought I had a date? Sometimes partial dates are better than a completely missing date, so dates are entered in character or string format. This way partial date information can be preserved. A SAS date variable is either valid or missing—no fence sitting!

So you are tasked with finding and estimating missing dates. Dates lacking the day have ‘dd’ coded rather than the day and missing months are similarly coded ‘mm’. We know that dates are entered in a mmddyy8 format.

We might pick the pieces off one at a time with a substring function as follows:

```
day = substr(cdate,1,2)
mon =  substr(cdate,4,2)
year =  substr(cdate,7,2)
```

These pieces could then be run through some logic such as:

```
if lowcase(mon) = ’mm’ then do;
   * Month missing, so day assumed missing;
   mon = ’07’;
   day = ’01’;
end;
else if lowcase(day) = ’dd’ then day = 15;
```

Followed by:

```
est_date = mdy(mon,day,year);
```

Alternately we might save some code, and not create extra variables. Here is one possibility:

```
if indexc(cdate, ’dd’) <> 0 then
   est_date =
     input(tranwrd(cdate,’dd’,’15’),ddmmyy8.);
else ‘use tranwrd twice (nested)’
```

I do not see this as being cute programming. On the contrary, it lets you accomplish a specific task is a small area and move onto the rest of the logic of the data step with little interruption of your thought process.

TRANWRD

Some of you may not have yet stumbled onto TRANWRD yet. This has been well hidden in the 6.07 update manual. If you do not have this book—get it or browse the online help in a recent version of SAS. Actually perusal of the whole update manual is recommended as it has more coding enhancements than any other recent release manual I know of.

How does TRANWRD work? It simply does a search and replace like any editor would. Prior to its debut, word to word (actually phrase to phrase) translations involved the use of index and substring functions.

Be careful using TRANWRD, for SAS does not do word wrapping into the next variable string. You can truncate your string and SAS will not even warn you!

Homework assignment: How would you determine whether you truncated your string or not? Hint:

```
If
        a = {a character with valid date}
and
        b = input(a,mmddyy8.)
and
        c = put(b,mmddyy8.)
then
        a = c
most of the time!
```

"Most of the time" because dates do not need to have consistent separators on input. A slash or a dash or both work fine. SAS will properly read 5/30-98.

TRANWRD will do the correct thing only 'most of the time' because the following fails:

```
  length a b c $ 200;
  a = 'sow so sew';
  b = tranwrd(a,'so','sew');
  c = tranwrd(b,'sew','so');
```

In this case a≠c because the translation of 'sew' back to 'so' hit the original 'sew' as well as the one translated from 'sew'. Clearly when they do not match you have to do something more elaborate. But if a=c you are fine. More about this later.

INDEX and siblings
Let's consider a few of the nuances illustrated above starting with INDEXC and its non-identical twin, INDEX. Both functions do double duty by determining **if** and **where** a string occurs in a character variable. They both tell you if the string occurs by returning a non-zero value when a match is found. They differ in that the INDEXC looks at the source string one character at a time, whereas INDEX evaluates for the existence of the entire excerpt. INDEX**C** is the detail-oriented function, looking at **c**omponents; INDEX looks at the whole.

Their usage may further be confusing since INDEX**C** take multiple arguments. Consider the following example:

```
  c = 'is this his code';
```

If we set the excerpt or search string to 'his', INDEXC returns a '1' because that is where it finds the first of any of the characters {h,i,s}. INDEX returns a 5, because that is where the intact string 'his' first appears ('is this his code';). How would you find the second occurrence of 'his' in this string? Happy SUBSTRing!

As a bonus, SAS has recently provided INDEXW as a more discerning brother to INDEX. Its added discretion is demonstrated in that 'his' must occur at a word boundary, thus it returns a value of 9. The INDEXW function also allows the user to specify what characters define a word boundary.

These functions are also gender neutral. We could look for 'her' in the string 'are there functions in her code' with similar results.

Confusion and resolution
Be careful not to confuse the differing implication of "word" in TRANWRD and INDEXW. Both are referring to a string of one or more characters, but TRANWRD does not pay any attention to word boundaries. Note the extra 'w' in the results of the prior piece of code.

```
  B = seww sew sew
  C = sow so so
```

Word boundaries were ignored by TRANWRD. However, what is the result of the following:

```
  x = indexw(b,sew);
```

Does x equal 1 or 6? Neither. The variable **sew** is uninitialized! Computers are so dumb – let's hope they stay that way! Anarchy will reign if SAS were to decide that since 'sew' is used as a string in one place in my data step, that it is 'reasonable' to assume the same elsewhere. It might be nice if it asked me, then fixed my code, but a human arbitrator is needed here – please!

For debugging a problem area, there is no substitute for writing a short piece of code, and using the "put _all_;" statement to display the results. I like this better than a print, because:

- all variables, temporary or permanent are displayed,
- you can display your variables before and after they are modified, and
- you get to see the code and results together.

PUT up or …
PUT and INPUT are not only powerful statements, but also rather awesome functions. These statements and functions are conceptually similar, but have profound differences.

Both **statements** are associated with reading and writing "external" data. External may be log or lis output files or some other file more loosely associated with SAS such as a flat data file. These statements also typically load to or unload from variables, with emphasis on the plural. Consider these examples:

```
input a b c;
input @1 name @13 dob date9.;
put a b c +5 dob worddate.;
```

If you want to read or write to a single line multiple time, use a trailing at sign (“@”) to hold the record open. This allows logic code to be interspersed with the I/O of a single record. This is a particularly significant performance issue since converting external data into SAS variables is quite costly. The efficiency comes when you can logically determine that you do not need to complete the costly data conversion for that record.

Interestingly SAS does not allow you to use functions in your PUT statement, thus uses of the PUT function must be done outside the PUT statement. This is occasionally cumbersome, but generally helps to keep your code clean and focused.

The (IN)PUT functions are a bit more simple-minded than their statement complements, but the functions probably are more frequently used, especially when you are doing data manipulation, validation, or reporting.

Both the PUT and INPUT functions create a single result. They are typically used in assignment or logical statements. Formats are their best friends, and can be used to create some very fast and sophisticated lookup or dictionary coding systems.

The PUT function **always** returns a character string. This makes it very useful in creating output strings. Very often, concatenation of pieces of the final product is used in conjunction with the PUT function. Note that SAS uses two vertical lines to indicate concatenation. Consider the following example. Data typically coming from 2 or more procedures are joined to create a statistical table containing a value and a percentage. The code might look something like this:

```
   val1 = put(sumstat1,3.) || put(prcnt1, 4.) || ‘%’;
   val2 = put(sumstat2,3.) || put(prcnt2, 4.) || ‘%’;
   val3 = put(sumstat3,3.) || put(prcnt3, 4.) || ‘%’;
   put @tab1 col1 @tab2 col2 @tab3 col3 ;
```

The result might look like the following:

```
 10  24%   139  75%   140  77%
```

This is very useful, but what would you suggest is one of the most common snafus encountered in writing code involving concatenation? For me at least, it has been that the concatenated value disappears. Likely story! Consider this example:

```
data _null_;
   length longchr $150;
   longchr = ‘To be or’;
   a = ‘ not to be,’;
   b = ‘ that is the question.’;
   longchr = longchr||a||b;
```

What would you expect the variable “longchr” to contain at the end of the datastep. It has only the first three words! Why? Because there are 142 blanks at the end. The concatenation “happens” but is then trimmed back to 150 characters. Contents of “a” and “b” are added starting at character 151, but is then truncated when stuffed back into the target variable “longchr”.

Similar kinds of problems occur when you do not assign a length to a new character field, and then have variable length input assigned to that field. The first use determines the length of the field. Subsequent use is truncated to that length. A common example of this problem might be as follows:

```
   if smokhbit = 1 then text = ’smoker’;
   else text = ’non-smoker’;
```

The variable will only contain the values: ‘smoker’ and ‘non-sm’ because the first use defined the variable to be a character string with length of 6.

COMPDBL
The COMPDBL function is another tool new to 6.07 that you may not use often, but when you need it, its use will be very obvious. It simply compresses multiple occurrences of contiguous spaces into a single space. If you are comparing multiple word phrases or cleaning up verbatim text, this tool can be invaluable.

Date and time issues

I have already touched on the use of date formats and functions above. For year 2000 compliancy, you ought to visit the SAS web site. In passing I would mention that the date formats and functions accept a 4-digit year.

Date and date time variables are stored as numbers with an associated format. There is nothing magic about them. Dates are stored as the number of days since January 1, 1960. Date time variables are the number of seconds since midnight of the same infamous date.

There are two very important functions, DATEPART and TIMEPART, which allow you to split a datetime variable into its respective date and time pieces. If you wish to calculate the number of days between two events or extract the particular time of the day, you will need to convert the datetime variables. Remember that you can use these functions directly in logic statements, without creating additional variables.

It is useful to know that in some DBMS environments, such as Oracle™, all date variables are datetime. If just the date is entered, the field will have a time of zero hours, minutes, and seconds (or midnight). If you know this, you should minimally format these fields as something like datetime9, which suppresses the presentation of the time value (e.g. 0:00:00). Using PROC ACCESS, these variables can be declared as just dates when they are pulled into SAS. This can eliminate some real headaches in subsequent coding.

As yet another aside, accessing external databases using PROC ACCESS to create access and view descriptors is a clean and controlled method for giving access to external files, and doing some data manipulation, like converting date time variables to date only variables. However this method has its hazards.

The access files, created with PROC ACCESS, do not provide the user with any information regarding the source of the external data. It is possible to compile an access file against test data and move it to a production area, and not realize that it is still pointing to the test database tables. Use of PROC SQL, with a “connect to”, in the code reveals the location of the data table. You might want to note this as a potential validation issue.

Sampling methods

Contiguous sample

It is often wise to test your code on a sample of your data to minimize computer resources. One of the simplest ways is to specify an observation limit in the options statement. You can also specify the starting point with the FIRSTOBS dataset option. Unfortunately both of these options conflict with any use of WHERE functionality, unless the obs limit is set to zero for syntax checking.

You can avoid this conflict by specifying the observation limits as dataset options in steps that do not have WHERE statements associated. You can also turn off the observation limit after initial files are read by resetting the option with ‘obs = max’, after which you can use WHERE functionality.

Periodic sample

Alternately you might want to take a sample that represents your data more broadly than just a contiguous chunk. The data step processor maintains an automatic variable _N_ which can be used to pick off every Nth row of data. An easy way to do this is to use the MOD function. The mod is the remainder in division. If you want every 13th record just divide _N_ by 13 and every time the remainder is say 4, keep that observation. A subsetting IF statement might look like the following:

```
   if mod(_N_,13) = 4;
```

Rows that evaluate as true are kept, others are dropped and the processing goes on to the next observation. What would the following statement do?

```
if mod(_N_,13);
```

It would keep 12 of every 13 records.
If you want to be more apparently precise, you can also subset using the if/then/else structure and specify ’if … then delete;’ if you like.

The subsetting IF statement just above here demonstrates an interesting but unrelated issue, which is the use of logical operators. If the expression is true, it essentially evaluates to 1, otherwise to 0. You can use this concept to do conditional arithmetic without an explicit logic statement.

Suppose we wanted to assign varying weights to some calculation based on age. Consider the following code:

```
   x = (age < 18)*3*factr + (age >= 18)*2*factr;
```

One of the two parenthetical logic constructs will always be zero so you will either get two or three times the factor, and nothing else.

Random sample

Back on the main trail! Using the MOD function as shown above could bias your sample if there is something methodical about your data, and the selected denominator is a permutation of that rhythm. To avoid this potential problem, use a random number generator to create a selection criteria. Be sure to use a uniform number generating function such as RANUNI or its alias UNIFORM. Do not use a highly biased number generator like RANEXP unless you are seeking to achieve some specific goal. If we wanted about a 7% sample we might code this as follows:

```
   if .31 ge ranuni(0) gt .38;
```

You expect that a uniform number generator which churns out numbers between 0 and 1 will generate a number between .31 and .38 about 7 out of a 100 times. Thus you get your desired sample size and do it randomly.

If you plan to rerun this sample and compare results (e.g. as you debug), it would be helpful to put a fixed, non-zero number in for the seed. When zero is used as the argument to a random number generator, the time of day is substituted in as the seed. Thus the selected records vary with each execution of the code. Since SAS, and most languages, use pseudo-random number generators, a fixed seed returns a fixed result.

If you will excuse another rabbit trail, using WHERE logic rather than a subsetting IF is noticeably more efficient. This is because the where functionality looks at the record before all the data are pulled into the data vector. Recent versions of SAS allow you to use functions in WHERE statements or in the WHERE dataset option. Thus you can move the random selection to a WHERE clause and save a lot of I/O time.

Unfortunately the automatic variable _N_ is not available in the context of a dataset option, and is not present at all in a procedure. Therefore the example above using the MOD function will not work as a dataset option. However, the random function will work fine in this context.

Mathematical tips

SAS provides some useful mathematical or statistical functions like MEAN, MIN, MAX, or STD. These sound like useful functions, but why would someone include a SUM function in a computer language. Can't programmers figure out how to add? Unlike spreadsheets and our minds, SAS rightly pays special attention to missing values. If B has a missing value, the result of A+B+C is missing. However the SUM(a,b,c) will be non-missing if any of {a,b,c} are themselves non-missing. All the statistical functions use only non-missing values.

This means that the denominator, N, in the statistical functions is the number of non-missing values contributing to the numerator. There are other helpful uses of the function "N". You might want to set a statistic to missing if "N" becomes too small. It might also be used in a logic statement such as:

```
  select ( n(of age1-age4) );
    when (4) size = 'Group';
    when (3) size = 'Few';
    when (2) size = 'Couple';
    when (1) size = 'Lonely';
    otherwise put 'Null set is inappropriate';
  end;
```

Now for a more purely mathematical consideration, what is the difference between the floor and the ceiling in a round room? Your answer may be relative – relative to whether there is any gravity where you are at. But in SAS the indifferent and somewhat lazy **ROUND** function seeks the shortest distance to the desired integer or other target. To **FLOOR** something implies that you knock it down (if g 0) a notch—to the nearest integer. **CEIL**ing is very energetic and always reaches up to the next integer. Note that ROUND can "round" to any precision either in regard to decimals or whole numbers. For instance you could round zip codes to just the three significant digits which defines the central post office. Neither CEIL, nor FLOOR do anything but convert real numbers to integers.

Closing thoughts

Please note that many of these functions have their counterpart as macro functions. Use of the macro string functions and macro quoting (e.g. %str, %nrbquote, %etc!) are essential for many macro tasks.

Not to open a can of worms at the end of my paper, but just to sound a warning—please note that a number that is rounded may return a different value in the last decimal place than the same number formatted. SAS formats its procedural output for means, P-values, and so forth. If you take these values from the output datasets and round them to the same precision as found in the output, they might not exactly match the printed output. This occurs when the 'rounded off' decimal portion is approximately one half of the last significant digit.

As a final enigma, what is a quintile? PROC UNIVARIATE reports on quartiles. I am here to represent Quintiles. The definition of a quintile has to do with a fifth of the sample. May your forays into functions be marked by many quintiles of success.

Summary

I hope I have helped you better understand and respect the robust set of functions that SAS has provided over the years. Do experiment – using small datasteps. Try new things when you can, because you never know when that new tool will be an absolute must for your next task.