PharmaSUG 2014 - Paper TT06

Functioning at a Higher Level: Using SAS® Functions to Improve Your Code

Peter Eberhardt, Fernwood Consulting Group Inc., Toronto, Canada

Lucheng Shao, Ivantis Inc., Irvine, CA

ABSTRACT

SAS® provides many built in functions that will help you write cleaner, faster code. With each new release of SAS there are new functions added; in many cases these new functions are overlooked because we have developed coding habits to try to accomplish the same result as these functions. In this paper we will survey some functions we find useful. In addition we will touch on how you can turn your code habits into functions.

INTRODUCTION

Using functions will improve your programs. Functions encapsulate logic, sometimes very complex logic, into one simple statement. Moreover, using the built-in functions relieves you of the necessity of writing and validating the logic already encapsulated in the functions. In this paper we will review just a few functions available in SAS that I use on a regular basis. Of course this means there are many more that are available and important, so this is paper is not meant to be a ranking, only a limited survey.

In general, SAS functions can be divided into two major types:

1. Character functions
2. Numeric functions

In this paper I will extend that slightly by introducing functions that cross these boundaries – first functions which deal with missing values, second, by introducing functions deal with SAS dates (even though SAS dates are really just simple numbers), third by introducing some logic functions, fourth some data conversion functions and finally some character functions.

Although we will see some examples using macro functions, the focus of this paper is on SAS data step (and SQL) function. Before we talk about using functions, let’s look at the definition of a function.

WHAT IS A FUNCTION

Every year the local ballet company holds a black-tie function to raise money; in this case, a function can be defined as a social gathering. The function of a soda bottle is to hold the soda; in this case a function can be defined as the purpose for which the object exists. What then is the thing we SAS programmers call a function?

A general definition of a function is:

> A function is a rule for transforming zero or more values called ***arguments***; the result of the transformation is called the value or result of the function. The transformation can also have side effect; that is permanent changes in the values of arguments.

The SAS 9.4 online help has a similar definition:

> A SAS function is a component of the SAS programming language that can accept arguments, perform a computation or other operation, and return a value. Functions can return either numeric or character results. The value that is returned can be used in an assignment statement or elsewhere in expressions.

SAS also has another type of function, called a CALL routine. CALL routines are similar to functions in that they also perform a computation or other operation; they differ in that they cannot be used in assignment statements, rather, values are returned through the variables passed into the routine.

With this brief introduction to functions we will turn to some of the SAS functions, starting with functions which deal with missing values

MISSING VALUES

Rare is the project that does not have to deal with missing data; in our code we commonly have to both check for and assign missing values. At first glance, this is straightforward; for example we will commonly see code similar to:

```
data studyGroupKeep studyGroupDrop;
     set allPatients;
     if visitCode = .
     then
        do;
          output studyGroupDrop;
          return;
        end;
    /* more code for the keep group */
run;
```

Here we are separating the patients based on the value of the variable ***visitCode***; if the value is missing (.) then we write the record to the ***studyGroupDrop*** table, otherwise we process the record. SAS has 29 different missing values (.A to .Z, ._, ., ‘ ‘) and the simple missing (.) is only one. If visitCode was coded with a value of .N, for example the visit was not applicable, the row would be improperly included in the studyGroupKeep table since the test is only checking for one of the possible missing values. You could change the code to test for all the possible different missing values, or you could use the ***missing()*** function; the ***missing()*** function takes one argument, either numeric or character, and returns a one (1) if the variable is any of the missing values, and zero (0) if the variable contains a non-missing value. This function not only helps catch all possible missing values, it also makes the code easier to understand; moreover, this works if the variable visitCode is character or numeric:

```
data studyGroupKeep studyGroupDrop;
     set allPatients;
     if missing(visitCode)
     then
        do;
          output studyGroupDrop;
          return;
        end;
    /* more code for the keep group */
run;
```

NMISS function and CMISS function are helpful if you want to count the number of missing values. The following table shows the comparison of MISSING(), NMISS() and CMISS().

MISSING()	MISSING() works with only one value that can be either numeric or character. MISSING(Variable)=1 when the data point is missing; MISSING(Variable)=0 when the data point is present.
NMISS()	NMISS() returns the number of missing values. It only takes numeric values, but could be multiple values.
CMISS()	CMISS() is similar to NMISS(). The only difference is that it takes either numeric or character.

In addition to checking for missing values, we often want to initialize variables to missing. A common approach looks like:

```
data totValues;
     set detailValues;
     array tot(*) month1 - month12;
     array nam(*) $ name1 - name5;
     do i = 1 to dim(tot);
```

```
         tot(i) = .;
      end;
      do i = 1 to dim(nam);
         nam(i) = ' ';
      end;
/* more statements */
run;
```

Here we are setting up arrays, then looping over the arrays to set the variables to missing. Of course, this is much better than having 12 separate assign statements for the numeric variables and 5 separate assign statements for the character variables, however there is a better and more efficient way to do this:

```
data totValues;
      set detailValues;
      call missing(of month: name:);
/* more statements */
run;
```

Here we have used the CALL missing routine to set both the numeric and the character variables to missing. Note also the use of the ***of*** keyword to introduce a list of like starting names. Non-rigourous benchmarking of the above two examples executing ten million times gave an average of 5.6 CPU seconds on the array method and 0.84 CPU seconds for the ***call missing()*** method; we have increased simplicity and achieved better performance.

Another very useful function to use when dealing with missing values is the ***sum()*** function. The ***sum()*** function adds numbers, but unlike the addition operator (+), the ***sum()*** function ignores missing values. The following examples first show the effect of adding numbers when using the addition operator then the same process and data using the ***sum()*** function:

```
data add;
      input x1 x2;
      c = x1 + x2;
      put x1= x2= c=;
      datalines;
1 1
2 2
1 .
. 1
;
run;
```

the results in the log:

```
x1=1 x2=1 c=2
x1=2 x2=2 c=4
x1=1 x2=. c=.
x1=. X2=1 c=.
NOTE: Missing values were generated as a result of performing an operation on missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      2 at 418:12
NOTE: The data set WORK.ADD has 4 observations and 3 variables.
```

And with ***sum()***:

```
data sum;
      input x1 x2;
      c = sum(x1, x2);
      put x1= x2= c=;
```

```
     datalines;
1 1
2 2
1 .
. 1
;
run;
```

the results in the log:

```
x1=1 x2=1 c=2
x1=2 x2=2 c=4
x1=1 x2=. c=1
x1=. X2=1 c=1
NOTE: The data set WORK.SUM has 4 observations and 3 variables.
```

The ***sum()*** function can take any number of arguments, separated by a comma (,); it can also take a list of variables using the ***of*** operator:

```
     c = sum(of x:);
```

Not only do we add all the non-missing values to get a non-missing result with the ***sum()*** function, we also have a cleaner log.

DATES

Internally, SAS dates are simple numbers – representing the number of days since Jan 1, 1960. As a result simple operations such as calculating the date five days ago or the number of days between two dates is easy – we can simply add/subtract numbers from dates to get another date or subtract one date from another to get the number of days between the two dates. For many applications this is sufficient. However, when your needs change you can turn to SAS date functions to help you.

Two of the simplest but very useful functions are ***date(), month(), day()*** and ***time(); date()*** returns the current date and ***time()*** returns the current time. Both the date and time are derived from the computer upon which SAS is running. These functions are useful for date stamping reports, creating audit trails and, since they can be accessed through macro code, controlling batch processing:

```
%macro runBatch;
%let saturday = 7;
%let currDate = %sysfunc(date());
%let dispDate = %sysfunc(date(), yymmdd10.);
%let currTime = %sysfunc(time());
%let dispTime = %sysfunc(time(), hhmm5.);
%let currDay  = %sysfunc(weekday(&currDate));
%let currHour = %sysfunc(hour(&currTime));

%* only run on Saturday on or after 8:00 pm;
%if &currDay = &saturday AND &currHour >= 20
%then
  %do;
    footer1 = "Saturday Night Results from &dispDate @ &dispTime"; /*Note: The macro variables
would only be resolved when double quotations are used. Single quotations won't resolve the
macro variables.*/
    %*  more processing;
  %end;
%else
```

```
  %do;
    %*  more processing;
  %end;
%mend;
```

In this code snippet we see the ***date()*** and ***time()*** functions used to capture the current date and time; note also when used with the %sysfunc macro function we can apply a format to the function result, In this example we call the ***date()*** function twice, once to get a date value and once to convert the date to its visual representation. We also use the ***weekday()*** function to get the day of the week and the ***hour()*** function to the current hour then use these values to determine which processing stream to follow. If this code was run on a Saturday at 18:07 the footer would look like:

```
"Saturday Night Results from 2014-04-05 @ 20:07"
```

Of course, now that we have the date and time captured in macro variables, we can use them elsewhere in this macro function,

One of the common problems I come across is the need to bring date variables from an RDBMS into SAS; in many cases, when the dates are brought into SAS they are SAS datetime variables, not SAS date variables. Datetime variables are also simple numbers, but rather than the number of names since Jan 1, 1960 they represent the number of seconds since midnight Jan 1, 1960; needless to say, this means it is not possible to directly compare a datetime variable to a date variable, even though the visual representation looks the same. In order to compare we can convert the datetime to a date using the ***datepart()*** function. I get regular calls from clients using SQL server saying “This code does not work. Why are there no records in todayTrans – I know there are thousands already today??”

```
%let currDate = %sysfunc(date());
/* more processing */
data todayTrans prevTrans;
     set SQLSRV.dailyTrans;
     if transDate = &currDate
        then output todayTrans;
        else output prevTrans;
quit;
```

The problem is the comparison of a date to a datetime; except at exactly midnight on Jan 1, 1960, these two types will never be equal. To make the comparison work we can convert the datetime to a date using the ***datepart()*** function; this function takes a datetime variable and converts it to a date variable

```
%let currDate = %sysfunc(date());
/* more processing */
data todayTrans prevTrans;
     set SQLSRV.dailyTrans (rename=(transDate=transDtTm));
     drop transDtTm;
     transDate = datepart(transDtTm);
     if transDate = &currDate
        then output todayTrans;
        else output prevTrans;
run;
```

Note also the use of the rename dataset option. Although it is not necessary to rename the variable before converting, I generally prefer to do this; this gives the option of keeping both variables if needed, or during testing, having the original value to validate against.

Since SAS dates are simply a number, determining a date a fixed number of days away is simply a matter of adding or subtracting a number; similarly determining the number of days between two dates is simply a matter of subtracting one from the other. However, determining a date that is not a fixed number of days away or an interval that is not days is not so simple, unless you use the functions ***intck()*** and ***intnx()***. The ***intck()*** function will return the number of intervals, for example months or weeks, between two dates, while the ***intnx()*** function will increment/decrement a date by a number of intervals and return the new date (note that these functions also work on datetime and time variables). For more detailed explanation of these functions see Eberhardt [2013b]. The following example demonstrates the use of the ***intck()*** function to determine the number of intervals between two dates; the example shows calculating the number of days, weeks, months, and years between two dates:

```
data _null_;
    start = '01jan2014'd;
    end   = '01jun2014'd;
    days1 = end - start;
    days2 = intck('day',  start, end);
    weeks = intck('week', start, end);
    months = intck('month', start, end);
    years = intck('year', start, end);
    put days1= days2= weeks= months= years=;
run;
```

The results in the log:

```
days1=151 days2=151 weeks=22 months=5 years=0
```

We see that we have two options for determining the days between two dates, one is subtraction and the other is the ***intck()*** function. Since the interval to calculate is passed in as a character variable, using the ***intnx()*** function makes the code more flexible in that the type of interval to calculate can be data drive; for example, one of the variables in the data may contain the interval to calculate:

```
data _null_;
    start = '01jan2014'd;
    end   = '01jun2014'd;
    length interval $5.;
    interval = 'day';
    intervals  = intck(interval,  start, end);
    put interval= intervals=;
    interval = 'week';
    intervals  = intck(interval,  start, end);
    put interval= intervals=;
    interval = 'month';
    intervals  = intck(interval,  start, end);
    put interval= intervals=;
run;
```

The results in the log:

```
interval=day intervals=151
interval=week intervals=22
interval=month intervals=5
```

If we need to determine a date some number of intervals away we can user the ***intnx()*** function; this function returns a date that is some number of intervals (days, weeks, months etc) away. One optional argument can be added to determine where the 'landing spot' is, that is move to the beginning, middle, end. For a week interval, this means moving to the beginning/middle/end of the week and similarly for the other intervals. This is particularly useful to get the first or last date of the current month. The following example shows adding one week to Jan 1, 2014 but ending on different days:

```
data _null_;
     start = '01jan2014'd; *Note: In SAS week starts on Sunday and ends on Saturday;
     week   = intnx('week', start, 1);     * default, beginning;
     weeksE = intnx('week', start, 1, 'e');* end;
     weeksS = intnx('week', start, 1, 's');* same;
     weeksM = intnx('week', start, 1, 'm');* middle;
     put week= yymmdd10. weeksE= yymmdd10. weeksS= yymmdd10. weeksM= yymmdd10.;
run;
```

the results in the log:

```
week=2014-01-05 weeksE=2014-01-11 weeksS=2014-01-08 weeksM=2014-01-08
```

Two dates I commonly have to calculate are the first and last day of a given month. With ***intnx()*** this is easy:

```
data _null_;
     start = '15may2014'd;
     monthB = intnx('month', start, 0, 'b');* beginning;
     monthE = intnx('month', start, 0, 'e');* end;
     put monthB = yymmdd10. monthE= yymmdd10.;
run;
```

the results in the log:

```
monthB=2014-05-01 monthE=2014-05-31
```

Again, because the interval and the endpoint are passed in as characters, the calculation can be data driven; our data can have variables indicating which interval we want and where the end point should be. For example, one record could calculate an interval of weeks and have the end point at the first of the week, and the next record calculate an interval of months and have an endpoint at the end of the month.

As can be seen from these simple examples, the ***intck()*** and ***intnx()*** functions are flexible and powerful, however they do require a good understanding of how they deal with intervals and boundaries.

Another function mdy() can be useful when imputing missing dates. For example, if missing (Day) then Date=mdy(4, 1, 2014).

LOGIC

There are two functions which can perform simple IF/THEN/ELSE logic: ***ifc()*** and ***ifn()***; both functions evaluate a logical expression in the first argument then return one value if the expression is true, one value if the expression is false, and optionally another value of the expression evaluates to missing. Although this seems straight forward, the test for missing can be less than obvious:

```
data ifc;
     input x1 x2;
     length c $7.;
     c = ifc((x1>x2), 'true', 'false', 'missing');
     put '1: ' x1= x2= c=;
     datalines;
1 1
1 2
1 .
. .
0 0
;
run;
```

the results in the log

```
1: x1=1 x2=1 c=false
1: x1=1 x2=2 c=false
1: x1=1 x2=. c=true
```

```
1: x1=. x2=. c=false
1: x1=0 x2=0 c=false
```

In rows three and four we have missing values in our data so at first glance we would expect the results to be missing, however we see that the we only get 'true' and 'false' results. This is because in SAS a logical test only returns true or false, even when dealing with missing values. If we want a result of missing when any part of the logical expression is missing we need to change the expression to resolve to a missing value. We can do this by taking the result of the logical expression and multiplying it my some function of the logical arguments. In the following example I change the logical condition from a simple test (x1=x2) to a arithmetic expression (x1+x2) * (x1=x2); if I was only interested in missing values of x1 this could be simplified to (x1) * (x1=x2).

```
data ifc;
     input x1 x2;
     length c $7.;
     c = ifc((x1+x2)*(x1>x2), 'true', 'false', 'missing');
     put '1: ' x1= x2= c=;
     datalines;
1 1
1 2
1 .
. .
0 0
;
run;
```

the results in the log:

```
1: x1=1 x2=1 c=false
1: x1=1 x2=2 c=false
1: x1=1 x2=. c=missing
1: x1=. x2=. c=missing
1: x1=0 x2=0 c=false
NOTE: Missing values were generated as a result of performing an operation on missing values.
```

This trick works because any logical test will evaluate to either 1 (true) or 0 (false), however in SAS any non-zero/non-missing value evaluates to true while a zero evaluates to false. This means our test (x1 > x2) will always return a one or zero, on the other hand the expression (x1+x2) will return a missing value if either x1 or x2 is missing or a non-missing value when neither are missing. Multiplying any number by a missing value results in a missing value.

CONVERSION

The SAS functions I use most frequently and find the most useful are the ***put()*** and ***input()*** functions. These work much like the input statement in the data step in that they read a variable and create another variable based upon a format or informat; for example I often want a character string variable with the 'decoded' value of a numeric code:

```
proc format;
value mf
0 = 'Male'
1 = 'Female'
other = 'Unknown'
;
run;

data _null_;
     input sex;
     sexDesc = put(sex, mf.);
     put sex= sexDesc=;
     datalines;
1
.
0
2
;
run;
```

the results in the log:

```
sex=1 sexDesc=Female
sex=. sexDesc=Unknown
sex=0 sexDesc=Male
sex=2 sexDesc=Unknown
```

You can say “I could have achieved the same results with ***put sex= mf.*** “, and you would be correct. However, if you want to export your data to another application, for example Excel, SAS formats are not available so you will want to export the ‘decoded’ value.

Another very common and excellent use of ***put()*** and ***input()*** is for format lookups; a format lookup uses a format that can be used to identify the values of a variable that are of interest then applies this format to the variable to select the values of interest. For example, I am interested in looking at the annual billing of 25 of 6,000 possible billing codes; the annual billing table is about 200 million rows. Using formats I can do this easily:

```
data codes;
     length start end  $5.;
     infile datalines;
     if _n_ = 1
     then
       do;
         start = 'other'; end = 'other'; label = 0; output;
       end;
     retain fmtname '$codes';
     input start $4.;
     end = start; label = 1; output;
     datalines;
A111
A222
... <<more codes>>
;
run;

proc format cntlin=codes; run;

data codesofInterest;
     set SQLSRV.annual;
     if put(feecode, $codes.);
     /* more processing */
run;
```

In this example we first create a ‘control in’ dataset, ***codes***, that will be used PROC format; I prefer to use in-stream data (***datalines***) in my work so I can easily see which codes are being analysed without having to open another file. Needless to say, if the codes are already in another data source you can read from this data source rather than in-stream. Within the processing step (***codesOfInterest***) I use ***if put(feecode, $codes.)*** to easily determine which codes to process.

Input() works in a similar manner: instead of applying a format to a variable it applies an informat to the variable.

CHARACTERS

The two most common character string processing I do are concatenation and search for the existence of a substring within a string. An old habit had me concatenating as follows:

```
data _null_;
     length firstName lastName $25. Fullname $50.;
     firstName = "Peter"; lastName = "Eberhardt";
     * first name order;
     fullName = trim(firstName) || " " || trim(lastName);
     * last name order;
     fullName = trim(lastName) || ", " || trim(firstName);
run;
```

Here I was using the ***trim()*** function to remove trailing blanks on both first and last name then using the concatenation operator (**||**) to insert a separator. These operations can be simplified using the ***CAT()*** family of functions (Hadden 2010). The above examples can be re-written as:

```
data _null_;
     length firstName lastName $25. fullName $50.;
     firstName = "Peter"; lastName = "Eberhardt";
     * first name order;
     fullName = catx(" ", firstName, lastName);
     * last name order;
     fullName = catx(", ", lastName, firstName);
     end;
run;
```

The value here is not readily apparent, however, when you have many variables to concatenate the value clearly shines:

```
allCodes = catx(",", code1, code2, code3, treat1, treat2, treat3, treat4);
/* or even more simply */
allCodes = catx(",", of code: treat:);
```

One warning when you move from using the concatenation operator to using cat() functions: by default the ***cat()*** functions return a string of length 200; make sure you explicitly assign a length statement to the result of the concatenation. See Zdeb 2012 for several interesting ways of using the ***cat()*** functions.

Another choice would be strip() function. It retains a character string with all leading and trailing blanks removed. Sometimes we know that there are leading or trailing blanks hanging there but we don't know how many there are. A good thing of applying this strip function is that we can remove all the blanks for sure and then add whatever number of blanks as we want.

To determine if one string is embedded in another we can use the ***find()*** family of functions The ***find()*** functions search the variable in the first argument for the characters in the second argument. In addition, there are two optional parameters, the start position of the search, and search modifiers. The ***find()*** function has two modifiers

- T (or t) – trims the blanks from both seach arguments
- I (or u) – ignore case when doing the seach.

The ***find()*** function will search argument 1 for the for the string in argument 2, whereas the ***findc()*** searches argument 1 for the for any characters in argument 2.

```
data _null_;
string = "cbaabcDEFABC" ;
pos = FIND(String, 'ABC');        put '1 ' pos=;
pos = FIND(String, 'ABC','i');    put '2 ' pos=;
pos = FINDC(String,'ABC','i');    put '3 ' pos=;
string = "This is PharmaSUG" ;
```

```
pos = FIND(String,'is','i',4);    put '4 ' pos=;
pos = FIND(String,'is','i',-99);  put '5 ' pos=;
pos = FINDC(String,'is','ik');    put '6 ' pos=;
run;
```

and the results from the log:

```
1 pos=10
2 pos=4
3 pos=1
4 pos=6
5 pos=6
6 pos=1
```

The first example found the string ABC in position 10 while example 2 found ABC in position 4; this is because in example 2 we instructed ***find()*** to ignore case. Example 3 found ABC in position 1 because we used the ***findc()*** function which searches for any of the characters in ABC, plus we instructed it to ignore case. Examples 4 and five show using a starting position; note that 1 negative start instructs find() to search backward. The last example shows another useful modifier ***k*** (***v*** is also does the same thing but is not documented in 9.4 help); the ***k*** modifier tells find() to look for anything ***except*** the characters in parameter 2.

Commonly used with ***find()*** is the ***substr()*** function; once a sub-string is located we want to extract it:

```
data _null_;
string = "This is PharmaSUG" ;
pos = FIND(String,' is ','i');    put '1 ' pos=;
place = substr(string, pos+4);    put '2 ' place=;
run;
```

results from the log:

```
1 pos=5
2 place=PharmaSUG
```

The string ‘ is ‘ was located at position 5, we took a substring starting at position 9 (jump over the string ‘ is ‘) and extract to the end of the string. ***substr()*** takes an optional third parameter specifying the number of characters to extract; if this option is omitted ***substr()*** extracts to the end of the string;

If you regularly use ***find()*** and ***substr()*** you should also look at using the Perl Regular expression functions; they are more powerful and flexible. See Borowiak 2012 (or any of Ken Borowiak's papers).

CONCLUSION

This has been a brief review of SAS functions I find useful, some of them you may also find useful. You may also use several functions that I have not listed. SAS provides a vast number of functions to help improve the quality of the code you write, and with each new release of SAS there are new functions available to us. One of your programming practices should be to periodically review the SAS functions to find ways to better ways to solve your problems.

REFERENCES

Borowiak, Kenneth [2012] http://analytics.ncsu.edu/sesug/2012/CT-03.pdf

Cody, Ron, [2012] “A Survey of Some of the Most Useful SAS® Functions”, *Proceedings of the SAS Global Forum 2012 Conference*

Eberhardt, Peter. [2010]. “Functioning at an Advanced Level: PROC FCMP and PROC PROTO.” *Proceedings of the SAS Global Forum 2010 Conference*

Eberhardt, Peter. [2013b]. “Why the Bell Tolls 108 times? Stepping Through Time with SAS.” *Proceedings of the PharmaSUG 2013 Conference*

Eberhardt, Peter and Wei Liu. [2013c]. “The Baker Street Irregulars Investigate: Perl Regular Expressions and CDISC” *Proceedings of the PharmaSUG 2013 Conference*

Eberhardt, Peter and Xiaojin Qin. [2013a] “ISO 101: A SAS® Guide to International Dating”, *Proceedings of the SAS Global Forum 2013 Conference*

Hadden, Louise. [2010] “Purrfectly Fabulous Feline Functions” *Proceedings of the SAS Global Forum 2010 Conference*

Zdeb, Mike [2012] http://www.sas.com/offices/NA/canada/downloads/UserGroups/TASS-June2012/Zdeb-CATFunctions.pdf

SAS documentation:
http://support.sas.com/documentation/cdl/en/lefunctionsref/67239/HTML/default/viewer.htm#p0nw8trwk5ooesn1o8zrtd5c6j1t.htm

CONTACT INFORMATION

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

Peter Eberhardt
Fernwood Consulting Group Inc.
Toronto ON Canada
peter@fernwood.ca
www.fernwood.ca
Twitter: @rkinRobin
WeChat: peterOnDroid

Lucheng Shao
Ivantis Inc.
38 Discovery, Suite 150, Irvine, CA, 92618
LShao@ivantisinc.com