

Create your own Functions using SAS/MACRO and SCL

Edward Foster, Oxford Pharmaceutical Sciences, UK

ABSTRACT

Functions (or methods) are used in all programming languages to encapsulate what sometimes can be complicated code into useable units for other programmers to use. There is a huge range of functions available within SAS® that can be used by the SAS programmer when they create their programs. These functions range from data manipulation/transformation functions such as INPUT or PUT to targeted statistical and financial functions. Sometimes, however, programmers find that there is not a function that can do the task that they require even for relatively common tasks (like obtaining the number of observations in a dataset). The programmer is instead forced to do some pre-processing in order to get the information they need. This paper will outline a technique for creating a range of functions using SAS/MACRO® and SCL that can be used to simplify and standardise these tasks.

INTRODUCTION

Conditional processing of data is sometimes dependent on the properties of a dataset and its variables (dataset metadata). Such properties include items like the number of observations, number of variables or even the type (character or numeric) of a particular variable. These properties and their values can be used to create generic data driven programs which can create reports or institute further data transformations. Dataset metadata can be obtained from proprietary or SAS generated metadata repositories, for example, SASHELP/DICTIONARY tables or PROC CONTENTS output, or even from directly processing the dataset itself. The values of these properties are often obtained using DATASTEP or PROC SQL code which causes the need for a code step to be executed prior to the conditional branch. Using SAS/MACRO® and SCL we can create functions, which act 'in-line' to obtain and return the values of such metadata.

Functions can be said to work 'in-line' because they can be placed directly into a piece of conditional logic (such as an IF statement) and the function will process and return a value within that line of code. If a function is not available then usually a SAS programmer would be required to derive the value themselves using DATASTEP or SQL code, place it in a macro variable and then 'inject' the value of the macro variable into the line of conditional code.

The use of functions greatly helps the user as values can be obtained with just a few simple parameters. Use of functions helps with program creation as it standardises methods for obtaining values. Program maintenance can also be improved as well because a simple function call is much more readable and easier to understand than a whole section of a program devoted to deriving some information for a conditional branch. A well named function also helps show the intent of the function as well as the part of the code within which it is used.

ROLE OF FUNCTIONS

Functions are usually used to manipulate or transform a set of inputs in order to return a derived value back to the executing program. For example, the INPUT function can be used to return a numeric value back to the user based on a character value and a valid SAS format.

Functions both encapsulate and raise the level of abstraction of the code contained within it. What this essential means is that the user of the function (i.e. the programmer) is protected from having to know the exact details of the code contained within the function. In effect all the programmer needs to know is;

- How to reference the function (by calling it)
- What parameters it takes (if any)
- How to capture any return value from the function (again if any)

As stated above a function can have zero or more parameters (or inputs) and may also return a value or not (though the majority in SAS do return a value).

For example:

```
data _null_;
  dt=today();
  chardt=put(dt,date9.);
  call symput('TODAYDT',chardt);
run;
```

PhUSE 2006

In the above code the function TODAY does not take any parameters. This is typical of functions that do not need to perform any derivation of information based on any user input. In this case the function simply finds today's date and returns it. The PUT function however does take parameters; a character value (supplied by the 'dt' variable) and a SAS format (in this case DATE9.). A string is returned and stored in the 'chardt' variable. The final function, CALL SYMPUT, is the only function that does not directly return a value instead passing a value to the macro variable TODAYDT.

Functions are not just useful for manipulating data but can also be used to dynamically change program flow.

For example:

```
%MACRO EXAMPLE01;

    ** COUNT OBERVATIONS **;
    data _null_;
        if 0 then set sashelp.class nobs=nobs;
        call symput('NOBS',nobs);
        stop;
    run;

    ** CONDITIONALLY RUN THE REPORT **;
    title "Listing of class data";
    %if &nobs^=0 %then %do;

        proc print data=sashelp.class noobs label;
        run;

    %end;
    %else %do;

        data _null_;
            file print;
            put // "There is no data in this dataset";
        run;

    %end;

%MEND EXAMPLE01;
%EXAMPLE01;
```

In the above example the CALL SYMPUT function is used to store the number of observations found in the SASHELP.CLASS dataset. The value of this macro variable is then used to conditionally execute either the reporting of the dataset contents or a message stating that there was no data within the dataset.

This works well but deriving the number of observations in the dataset requires the use of a DATASTEP (or PROC SQL) before the conditional branch. This process would be ideal for an 'in-line' function that negates the need for the DATASTEP.

```
%MACRO EXAMPLE02;

    ** CONDITIONALLY RUN THE REPORT **;
    title "Listing of class data";
    %if %nobs(sashelp.class)^=0 %then %do;

        proc print data=sashelp.class noobs label;
        run;

    %end;
    %else %do;

        data _null_;
            file print;
            put // "There was no data in this dataset";
        run;

    %end;

%MEND EXAMPLE02;
%EXAMPLE02;
```

PhUSE 2006

Unfortunately there is no such function in SAS but there is a way to create your own functions that can be used again and again in many different situations.

COMPONENTS NEEDED TO CREATE YOUR OWN FUNCTIONS

There are 2 components of SAS that are needed to create your own functions. These are SAS/MACRO and SCL. They each provide separate key ingredients to this process;

- SAS/MACRO – encapsulates code.
- SCL – provides access to the dataset metadata we require.

SAS/MACRO

While I am calling the code I will be creating 'functions' they are technically SAS macros. However the reason for calling them functions rather than macros is because they are used in a way which is not the norm for traditional SAS macros. SAS programs are usually made up of a number of DATA or PROC steps in order to process data. This data may be manipulated into a form that can either be summarized in a report or passed on to another process such as a load into a data warehouse. SAS/MACRO can then be used to encapsulate these programs (or processes) and by substituting macro variables into key parts of the program (such as a dataset name) and setting the macro call signature to accept parameters, the program can be made more generic. This enables the programmer to change the function of the program to suit his or her own requirements via the macro call. In creating our functions we will use this advantage to allow the user to apply it to the situation at hand. However, unlike traditional macros, in order to act like a function we will actually have a return value.

SAS/MACRO also has one other key ingredient for function creation that I will come to later.

SCL

SAS Component Language (formerly Screen Control Language) or SCL is the second component required to create our functions. SCL is primarily used in SAS/AF programming to create applications that work within the SAS environment. A number of SCL functions allow SCL code to access and manipulate SAS datasets and as such have access to the SAS dataset metadata.

The key SCL functions that control access to SAS datasets are;

- OPEN (*dataset name, dataset access method*)
 - Returns a dataset identifier as a numeric value
 - Returns a zero if the dataset cannot be opened (e.g. because another user has an exclusive lock on the dataset)
 - The dataset identifier returned by the function can be used by other SCL functions to reference the OPEN dataset when they perform their actions
 - In most case you will use 'l' as the dataset access method as this denotes a read-only access method that can help prevent the locking of a dataset to other users
 - Example syntax: DSID = OPEN("SASHELP.CLASS","l");
- CLOSE (*dataset identifier*)
 - Closes the dataset referenced by the dataset identifier and releases it for use
 - Must be called when you have finished with the dataset otherwise your program will still have a handle (or reference) to the open dataset
 - Example syntax: RC=CLOSE(DSID);

Once a dataset has been 'opened' by the OPEN function other SCL functions can then be used to return or update information about the dataset.

For Example:

- ATTRN – returns a number of different numeric metadata items about a dataset including the number of observations and the number of variables. **N.B.** There are a number of different ways to return the number of observations from this function (different counts may include deleted observations etc.) so make sure you select the right one.
- ATTRC – returns character information about a dataset but is generally less useful than ATTRN.
- VARNAME – returns the name of a dataset variable based on its index number (position in the dataset).
- VARNUM – returns the variable index based on the variable name. Returns a zero if the variable does not exist.
- VARTYPE – returns the variable type (character or numeric).

As you can see by combining the elements of both SAS/MACRO and SCL we should be able to create the functions we need. However, how do we combine these two very different components of SAS? The answer to this question comes in the form of the very useful SAS/MACRO function %SYSFUNC.

%SYSFUNC

%SYSFUNC is a very useful part of the SAS/MACRO facility because it gives macro code access to functions it does not natively support. For example a common usage of %SYSFUNC is to get the current date and time (via the TODAY and TIME functions) for placement in a report footnote.

PhUSE 2006

```
footnote1 "Report Generated on %sysfunc(today() date9.) at %sysfunc(time(),time5.)";
```

The functions that %SYSFUNC supports is not just limited to dataset functions, a range of SCL functions are supported too. These functions include all the ones that we require; OPEN, CLOSE, ATTRN etc.

HOW TO RETURN A VALUE

Understanding how to return a value from a macro is not as trivial as you first might expect. Macro variables that are created within a macro are stored within the scope of the macro's local symbol table and are wiped once the macro finishes execution. One way to get the macro variable value outside the scope of the macro is to 'promote' it to the GLOBAL symbol table via the %GLOBAL macro statement. However, this does not help us create a function that works 'in-line' because we will need to first execute the macro and then reference the global macro variable in order to use the value in our conditional branch. What we need to happen is for the macro itself to resolve to the value we want.

The following example shows how we can do this:

```
%MACRO EXAMPLE03;

    BOB

%MEND EXAMPLE03;

data _null_;
    put "The name is %example03";
run;
```

If you run this example you will get the following in the SAS Log.

```
16  %MACRO EXAMPLE03;
17
18      BOB
19
20  %MEND EXAMPLE03;
21
22  data _null_;
23      put "The name is %example03";
24  run;
```

```
The name is BOB
```

As you can see the macro has resolved to the string 'BOB' which is inserted into the PUT statement text. The following example expands this a little.

```
%MACRO EXAMPLE04;

    %let name=FRED;

    &name;

%MEND EXAMPLE04;

data _null_;
    put "The name is %example04";
run;
```

In this example the %LET statement sets the value of the local macro variable NAME. The use of &NAME will resolve the value of name causing the macro to resolve to FRED in the same way as the previous example. **Or does it?**

Note the output to the SAS Log below.

```
3  %MACRO EXAMPLE04;
4
5      %let name=FRED;
6
7      &name;
8
```

PhUSE 2006

```
9   %MEND EXAMPLE04;
10
11  data _null_;
12      put "The name is %example04";
13  run;
```

The name is FRED;

The line in the log has a semicolon at the end. This is because the macro code contains a semicolon after the '&name' line so the macro actually resolves to 'FRED;'. This does not pose a problem in this case but what if the macro was inserted into the following code;

```
%if %example04 = FRED %then %do;
    ...some stuff to do here...
%end;
```

In this case the code would error due to the semicolon being inserted into the %IF line;

```
%if FRED;=FRED %then %do;
    ...some stuff to do here...
%end;
```

In order to stop the error from occurring we need to remove the semicolon from after the '&name' in the %EXAMPLE04 macro. When we create functions we need to bear in mind what the function we have created actually resolves to when it is called. This issue surfaces again in a couple of other areas that I will outline under 'CAVEATS' later in the paper.

BRINGING %SYSFUNC INTO THE MIX

Let's make something useful!

The following code is a simple version of a function that will return the number of observations in a dataset. Well, only one dataset at the moment, SASHELP.CLASS! Note that we are asking ATTRN to return the number of observations in the dataset. ATTRN can also return a range of information, such as the number of variables in a dataset, by modifying the second parameter (in the case of number to variables to NVAR).

```
%MACRO EXAMPLE06;

    %let dsid=%sysfunc(open(SASHELP.CLASS,I));

    %if &dsid^=0 %then %do;

        %let nobs=%sysfunc(attrn(&dsid,NOBS));
        %let rc=%sysfunc(close(&dsid));

    %end;

    &nobs

%MEND EXAMPLE06;

data _null_;
    put "The number of observations = %EXAMPLE06";
run;
```

While we obviously need to upgrade this code to be more generic it is the basis of a function (see below) that will perform in the way we outlined in Example02.

```
%MACRO NOBS(DSN);

    %local nobs dsid rc;
    %let nobs=0;

    %let dsid=%sysfunc(open(&DSN.CLASS,I));

    %if &dsid^=0 %then %do;

        %let nobs=%sysfunc(attrn(&dsid,NOBS));
        %let rc=%sysfunc(close(&dsid));
```

PhUSE 2006

```
%end;  
  
&nobs  
  
%MEND NOBS;
```

Now we have a function that can be used 'in-line' within our conditional processing. Using the same techniques we could create macros that;

- Count the number of variables in a dataset (using ATTRN).
- Create a space delimited list of all the variable names in a dataset using ATTRN to count the number of variables then loop through and use VARNAME to append the variable name to a macro variable. There are 2 useful variations on this:
 - Changeable delimiters – set the default to a space but allow the programmer to change this as appropriate. This is especially useful as you can change the delimiter to a ',' which then returns a list of variables you can use in SQL statements.
 - Character, Numeric or All variable lists – returning a list of character or numeric variables can be a useful addition to just returning all variables because these lists can be used in array processing.
- Return the variable index (position) number using VARNUM. While it may not always be useful to return the variable index number this function comes into its own when testing if a variable exists in a dataset. If the variable does not exist then VARNUM returns a zero so you can use your function to test if a variable first exists (if it returns a value greater than zero for your variable) before processing it.
- Returns the variable type using VARTYPE. This is very useful in generic programs that might not know the type of a particular variable before you process it. By testing the variable type you can alter the way it is processed.

You don't just have to create functions that use SCL functions. Another function I find useful is one which counts the number of words in a string based on a specified delimiter and then resolves to that value.

The final example in this section pulls a few of these functions together into a program:

```
%MACRO EXAMPLE07(DSN);  
  
data test1;  
  set &dsn;  
  
  /*THIS EXAMPLE SHOWS HOW WE CAN CHECK TO SEE IF A VARIABLE EXISTS  
  BEFORE RUNNING A BIT OF CODE*/  
  %if %varnum(&dsn,AGE)>0 %then %do;  
    /*CONVERT AGE TO MONTHS*/  
    age=age*12;  
  %end;  
  %if %varnum(&dsn,BP)>0 %then %do;  
    /*PRINT BLOOD PRESSURES - THIS VARIABLE DOES NOT EXIST SO NOT EXECUTED*/  
    put BP=;  
  %end;  
  
  /*THIS EXAMPLE SHOWS HOW WE CAN TEST THE VARIABLE TYPE TO DO A GIVEN FUNCTION*/  
  %if %numvars(&dsn)>0 %then %do i=1 %to %numvars(&dsn);  
  
    /*IF THE VARIABLE IS NUMERIC THEN CREATE A NEW CHARACTER VARIABLE  
    BASED ON THAT VARIABLE*/  
    %if %vtype(&dsn,%scan(%varlist(&dsn),&i,%str( )))=N %then %do;  
  
      %let var=%scan(%varlist(&dsn),&i,%str( ));  
      new_&var=put(&var,8.2);  
  
    %end;  
  
  %end;  
  
run;  
  
%MEND EXAMPLE07;  
%EXAMPLE07(SASHELP.CLASS);
```

CAVEATS

I mentioned earlier that when we create these functions we need to be mindful of what the macro will actually get resolved to when it is executed. In addition to the semicolon issue outlined earlier there are two further points to note. Firstly, and perhaps the most obvious, is that there can be no DATASTEP code in these functions. Take the following example:

```
%MACRO EXAMPLE08;
  data _null_;
    if 0 then set sashelp.class nobs=nobs;
    call symput('NOBS' nobs);
  run;

  &nobs

%MEND EXAMPLE08;
```

The data step is a perfectly good way to calculate the number of observations in a dataset. However, if you use this macro in the following macro code an error will occur.

```
%if %example08 = >0 %then %do;
  ...some stuff to do here...
%end;
```

The reason for this is that %example08 resolves to the code below because macro code is resolved before DATASTEP and PROC code executes, hence when the macro resolves the data step code has not executed.

```
%if data _null_;
  if 0 then set sashelp.class nobs=nobs;
  call symput('NOBS' nobs);
  run;
= >0 %then %do;
  ...some stuff to do here...
%end;
```

This leads to the related and perhaps surprising second problem, program comments. When you comment your code you have a few options. You can either use the * *; syntax or /* */ syntax. The problem is that the * *; syntax will cause a similar error to the inclusion of data step code in your function. The reason is that the * *; comments are actually SAS statements and therefore these are not stripped out of the macro and so are present when the program executes. The /* */ style comments does not cause this issue so all comments in these functions must use this style.

N.B. This also applies to any program header that you have at the top of the program, even if it is placed outside of the macro.

CONCLUSION

By combining SAS/MACRO and SCL through the %SYSFUNC function we can create SAS macros that act in the same way as functions. These functions can be used 'in-line' as part of conditional programming within macros and data step code. Truly data driven programs are a goal of generic data processing systems in SAS. Programs of this type are reactive to the actual data being processed, testing the datasets for variables and variable types prior to data manipulation. These macros help to achieve this by providing easy access to the information required.

Using these functions also has advantages with regard to code production and maintenance:

- Encapsulation of code in functions protects the programmer from having to understand all the intricacies of the code, all a programmer needs to provide is a set of parameters.
- An appropriately named function within a line of code is easier to understand than a random macro variable. A macro variable could have been generated in a totally different program making understanding and debugging a program difficult. By using a function it is obvious what is happening in the code. A function could be said to be 'strongly-typed'.

These macros are also ideal candidates for inclusion in any global macro library that you may have at your company because they can be used in many different situations to create truly data-driven programs.

CONTACT INFORMATION

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

Edward Foster
Oxford Pharmaceutical Sciences
Lancaster House
Kingston Business Park

PhUSE 2006

Kingston Bagpuize
Oxford / OX13 5FE
Work Phone: +44 (0) 1462 477948
Email: edward.foster@ops-web.com
Web: www.ops-web.com

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.