PhUSE US Connect 2018

Paper CT16

Proc FCMP: Stay Magical

Keerthana Palwai, Syneos Health, Hyderabad, India

ABSTRACT

Earlier SAS has comprised of a simple data steps and few of the procedures. Then we started fledging and improvising programs using Macro statements, which in turn follow the scripted instructions .The ultimate solution where you do not need to parse each and every parameter, is using PROC FCMP via the RUN_MACRO function. The true value can be empowered only when we start to learn and try using it. The procedure can quickly be learned and this allows to explore some of the most significant and impressive aspects of the FCMP procedure. The paper comprises of macros vs functions with the limitations of data step functions, the big picture behind programming with run_macro routine and building off prior information with User written call routines with PROC FCMP. We can create functions to work as independent units and this paper would be focused on how it can be used in our programs for easy calculations, standardization, providing ease of access with examples, simplification of code and to develop a dynamic programming structure.

INTRODUCTION

A SAS® data set might contain hundreds or even thousands of variables. Programmers typically run PROC FREQ on categorical variables and PROC MEANS on continuous variables as part of checking data quality. Determining which variables are categorical and which variables are continuous can be a challenge. Typing long lists of variable names is tedious and error prone. The PROC FREQ statement option NLEVELS can be used to determine the number of levels of each variable. The Output Delivery System (ODS) can be used to save this information to a SAS data set.

Functions in SAS

What role do SAS functions really perform for us? We use them so often we might take their operation for granted. Functions accept numeric or character values as parameters and produce a single return value. The return value can represent the result of a calculation, lookup-operation, or string transformation. Below are a few familiar examples.

- **STD(1,2,3,4,5) – What is the standard deviation of these five values?**
- **SUBSTR('name',1,3) – What are the first three characters in the string**
- **DATE() – What is today's date as a SAS date value?**

I like to think of functions as answering a question. When we use a function in a DATA step it is like we have an expert at our side, and the function call is our Phone-a-Friend. Rather than have to spell out calculations manually, such as for the standard deviation of a set of variables, we simply call the STD() function. The DATA step pauses, passes the parameters to the function, and it calculates the result. Functions are often visualized as a 'black box'.

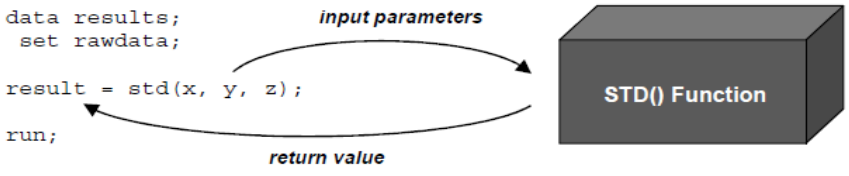

Figure 1. Diagram of SAS function during program execution

DATA Step Context:

- Functions typically operate in context of a DATA step.
- Functions take values *from* the DATA step as parameters and return values to the data step as a result.

Encapsulation:

- The black box represents encapsulation –we do not know or care how the function is being implemented by SAS, we only care that it returns the correct result.
  Ex: STD() – Calculations are long and cumbersome.

PROC FCMP:

- **CREATING FUNCTIONS AND SUBROUTINES**

PROC FCMP enables you to write functions and CALL routines using DATA step syntax. PROC FCMP functions and CALL routines are stored in a data set and can be called from several SAS/STAT, SAS/ETS, or SAS/OR procedures such as the NLIN, MODEL, and NLP procedures. You can create multiple functions and CALL routines in a single FCMP procedure step.

Functions are equivalent to routines that are used in other programming languages. They are independent computational blocks that require zero or more arguments. A subroutine is a special type of function that has no return value. All variables that are created within a function or subroutine block are local to that subroutine.

Example :

This following example defines a function and a subroutine. The function begins with the FUNCTION statement, and the subroutine begins with the SUBROUTINE statement. The DAY_DATE function converts a date to a numeric day of the week, and the INVERSE subroutine calculates a simple inverse. Each ends with an ENDSUB statement.

```
proc fcmp outlib = sasuser.MySubs.MathFncs;
   function day_date(indate, type $);
      if type = "DAYS" then wkday = weekday(indate);
      if type = "YEARS" then wkday = weekday(indate*365);
      return(wkday);
   endsub;
   subroutine inverse(in, inv);
      outargs inv;
      if in = 0 then inv = .;
      else inv = 1/in;
   endsub;
run;
```

The function and subroutine follow DATA step syntax. Functions and subroutines that are already defined in the current FCMP procedure step, as well as most DATA step functions, can be called from within these routines as well. In the example above, the DATA step function WEEKDAY is called by DAY_DATE.

The routines in the example are saved to the data set *sasuser.MySubs*, inside a package called MathFncs. A package is any collection of related routines that are specified by the user. It is a way of grouping related subroutines and functions within the data set. The OUTLIB= option in the PROC FCMP statement tells PROC FCMP where to store the subroutines it compiles, and the LIBRARY= option tells it where to read in libraries (C or SAS).

- **Writing a User-Defined Function**

The following program shows the syntax that is used to create and call a PROC FCMP function from a DATA step. This example computes the study day during a drug trial.

The example creates a function named STUDY_DAY in a package named TRIAL. A package is a collection of routines that have unique names and is stored in the data set sasuser.funcs. STUDY_DAY accepts two numeric

arguments, *intervention_date* and *event_date*. The body of the routine uses DATA step syntax to compute the difference between the two dates, where days that occur before *intervention_date* begin at -1 and become smaller, and days that occur after and including *intervention_date* begin at 1 and become larger. This function never returns 0 for a study day.

STUDY_DAY is called from DATA step code as if it were any other function. When the DATA step encounters a call to STUDY_DAY, it will not find this function in its traditional library of functions. Instead, SAS searches each of the libraries or data sets that are specified in the CMPLIB system option for a package that contains STUDY_DAY. In this example, STUDY_DAY is located in *sasuser.funcs.trial*. The program calls the function, passing the variable values for *start* and *today*, and returns the result in the variable SD.

```
options pageno=1 nodate;
proc fcmp outlib=sasuser.funcs.trial;
   function
study_day(intervention_date,
event_date);
       n=event_date-intervention_date;
       if n <= 0 then
       n=n+1;
       return (n);
   endsub;
```

```
options cmplib=sasuser.funcs;
data _null_;
   start = '15Feb2006'd;
   today = '27Mar2006'd;
   sd = study_day(start, today);
   put sd=;
run;
```

Output from the STUDY_DAY User-Defined Function sd=41

➢ Using Library Options

You can use PROC FCMP with the OUTLIB= or INLIB= options. The syntax for this procedure has the following form:

```
proc fcmp outlib=libname.dataset.package
          inlib=in-libraries;
routine-declarations;
```

The OUTLIB= option is required and specifies the package where routines declared in the *routine-declarations* section are stored.
Routines that are declared in the *routine-declarations* section can call FCMP routines that exist in other packages. To find these routines and to check the validity of the call, SAS searches the data sets that are specified in the INLIB= option. The format for the INLIB= option is as follows:

```
inlib=library.dataset
inlib=(library1.dataset1 library2.dataset2 ... libraryN.datasetN)
inlib=library.datasetM - library.datasetN
```

If the routines that are being declared do not call FCMP routines in other packages, then you do not need to specify the INLIB= option.

➢ Declaring Functions

You declare one or more functions or CALL routines in the *routine-declarations* section of the program. A routine consists of four parts:

1) A name 2) A body of code 3) One or more parameters 4) A return statement

You specify these four parts between the FUNCTION or SUBROUTINE keyword and an ENDSUB keyword. For functions, the syntax has the following form:

```
function name(argument-1, ... , argument-n);
```

```
   program-statements;
   return (expression);
endsub;
```

After the FUNCTION keyword, you specify the name of the function and its arguments. Arguments in the function declaration are called formal arguments and can be used within the body of the function. To specify a string argument, place a dollar sign ($) after the argument name. For functions, all arguments are passed by value. This means that the value of the actual argument, variable, or value that is passed to the function from the calling environment is copied before being used by the function. This copying ensures that any modification of the formal argument by the function does not change the original value.
The RETURN statement is used to return a value to a function. The RETURN statement accepts an expression that is enclosed in parentheses, and contains the value that is returned to the calling environment. The function declaration ends with an ENDSUB statement.

NLEVELS

There are many ways to obtain the number of unique values taken by a given variable in a given data set. Perhaps the most straightforward is to use a PROC FREQ with the NLEVELS option.

```
ods output nlevels=NLevels_out;
   proc freq data=sashelp.class nlevels;
      tables age / noprint;
   run;
```

TableVar	NLevels
Age	6

Output 1. NLEVELS output object

The NLEVELS option tells the procedure to generate an output dataset containing the variable name and 'NLevels', which is a count of the number of distinct levels of that classification variable. The ODS statement names this output object as the data set NLevels_out.
From the output we see variable Age in SASHELP.CLASS has 6 unique values.If Age had contained missing values, the output data set would also contain 'NMissLevels' and 'NNonMissLevels'.
Since our programmer will need to determine NLEVELS for a large number of variables and data sets, a macro would probably be used, accepting data set name and variable name as parameters. Ultimately the goal would be a SAS data set containing at least these three variables: data set name, variable name, and the corresponding NLevels.

```
%macro NLevels(dsn, var, out);
ods output nlevels=NLevels_out;
proc freq data=&dsn nlevels;
tables &var / noprint;
run;
data &out;
set NLevels_out;
dsn = "&dsn"; *add data set name to output;
run;
proc datasets library = work nolist;
delete NLevels_out; quit; %mend;
```

This will work for one data set and variable, but how will the output data sets, NLevels_out, be combined across macro calls? The programmer could implement a new parameter for output data set name so that the output datasets do not overwrite one another. Still, this results in a large number of intermediary data sets being generated which would then have to either be appended together afterwards or appended to a pre-existing shell data set.
It would be great if we could circumvent the entire issue of managing output data sets, as all we really want is one number, NLevels, for each variable and data set combination.
Let us restate the problem in the form of a question:
What is the NLevels number for this variable in this data set? Alternatively, expressed in the form of a function:
`NLevels = NLevels(dsn, var);` *the function approach;
We cannot simply put the macro on the right of the equal sign:
`NLevels = %NLevels(dsn, var);`
Because the macro contains multiple statements, it will generate syntax errors when it resolves. Macros are *not* functions.

ENCAPSULATING SAS PROCEDURES AND DATA STEPS IN FCMP FUNCTIONS -

To enable calling any SAS program from within an FCMP subroutine or function, special function is provided by PROC FCMP: RUN_MACRO**.** The RUN_MACRO function executes a predefined SAS macro. This function is quite powerful because they allow virtually any SAS program to be called from within the DATA step and any procedure that can use FCMP. Both pass the current value of local function variables to and from the executed macro or program through the use of macro variables.
These two functions work pretty much in the exact same way. For simplicity, in this paper we will demonstrate only the RUN_MACRO function. The syntax of the RUN_MACRO function is as follows:

```
rc = run_macro('macro_name', variable_1, variable_2, …);
```

where ***rc*** is 0 if the function was able to submit the macro. The return code indicates only that the macro call was executed.
macro_name specifies the name of the SAS macro to be executed. The macro itself should not have explicit arguments.
variable_n specifies optional PROC FCMP variables whose current values are passed to and from macro variables of the same name. Before PROC FCMP executes the macro, local SAS macro variables are defined with the same name and value as the FCMP variables. After PROC FCMP executes the macro, the macro variables' values are copied back to the corresponding FCMP variables. This example creates a macro called TEST_MACRO, and then uses the macro within a PROC FCMP function to subtract two numbers.
Notice that we can execute and test the function directly from PROC FCMP. The following example demonstrates calling the function from within the DATA step:
The *cmplib* global option is used to allow the DATA step to find the FCMP function. Output from executing this program: diff=4.6

```
/* Create a macro called testmacro
*/
%macro test_macro;
%let difference = %sysevalf(&a -
&b);
%mend test_macro;
/* Use testmacro within a function
to subtract two numbers */
proc fcmp outlib =
sasuser.ds.functions;
function subtract_macro(a,b);
rc = run_macro('test_macro', a, b,
difference);
if rc eq 0 then return(difference);
else return(.);
endsub;
```

```
/* test the call */
a = 5.3;
b = 0.7;
diff = subtract_macro(a, b);
put diff=;
run;
```

```
options cmplib = (sasuser.ds);
data _null_;
a = 5.3  ;
b = 0.7;
diff = subtract_macro(a, b);
put diff=; run;
```

This, of course, is a very simplistic example of how to call a SAS macro from an FCMP function. The important point is that through this mechanism, any SAS procedure or DATA step can be called from an FCMP function, and all the power of the SAS macro language can also be used.
Regarding the passing of FCMP variables to and from SAS macro variables, character variables are passed in quotation marks. It is often necessary to use the DEQUOTE function to remove quotation marks from these macro variables for proper usage inside the SAS macro language:

```
%let ds_name = %sysfunc(dequote(&ds_name));
```

MACROS VS. FUNCTIONS

SAS macros are designed to generate output and data sets, not values themselves. In the past, programmers went to great lengths to create a return value from a macro with so-called 'function-style macros'. These macros could only contain macro statements, as they had to resolve to their 'return' value. Thanks to RUN_MACRO, in PROC FCMP, this is no longer necessary. Why imitate a function when we can now write our own functions to suit our purpose?
RUN_MACRO routines give us a way to retrieve a metric from macro execution, by wrapping the macro call inside an FCMP function. The function executes the macro, and returns a single value to the DATA step. But in order to write our own functions we first need to take a look 'inside the box', and get a crash course in PROC FCMP.

THE RUN_MACRO FUNCTION

We only need one line of code inside our PROC FCMP function in order to call a macro – the special RUN_MACRO function (hereafter, 'command'). In a RUN_MACRO routine, the FCMP function often serves as a wrapper.

```
PROC FCMP outlib=work.functions.wrapper;
            function nlevels(dsn $, var $);
            rc= run_macro('get_nlevels', dsn, var
       return(nlevels);
      endsub;
  RUN;
  options cmplib=work.functions;
```

```
/* Think:
%Local dsn = dsn;
%Local var = var;
%Local nlevels = nlevels;

%get_NLevels();

dsn = symget('dsn');
var = symget('var');
nlevels = symget('nlevels');

*/
```

During execution, when the command is reached, the function takes each variable on the RUN_MACRO parameter list and creates a local macro variable with the same name and value. Then the macro is executed, and the values of each macro variable at the end of the macro execution are written back to the corresponding variables in the function.
Parameters on the RUN_MACRO command represent two-way communication between the function and the macro. The rc variable returns 0 if the macro was submitted successfully, otherwise it returns an error code.
There are some slight modifications that need to be made to our %NLevels() macro for it to be 'function-ready'.

```
%Macro Nlevels(dsn,Var);

    Ods output
nlevels=Nlevels_out;
    proc freq data=&dsn
nelevels;
        tables &var/noprint;
    RUN;
%MEND;
```

```
%macro get_nlevels(); 1
%let dsn=%sysfunc(dequote(&dsn.)); 2
%let var=%sysfunc(dequote(&var.));
ods output nlevels=Nlevels_out;

proc freq data=&dsn nlevels;
    tables &var/noprint;
    RUN;
data _null_;
    set Nlevels_out;
    call symput('nlevels',Nlevels); 3
RUN;
    Proc datasets library=work nolist;
        delete Nlevels_out;quit;

        %mend get_Nlevels;
```

THE BIG PICTURE

We can visualize a RUN_MACRO routine in terms of three components: the calling data set, the wrapper function, and the underlying macro. The function is called once for each observation in the calling data set, which means the macro is executed once for each observation in the calling data set. So far this is looking a lot like CALL EXECUTE. The advantage of RUN_MACRO functions is that we have a built-in channel to *report back* to the calling data set.

Figure 2. Diagram of a RUN_MACRO function during program execution

If the DATA step represents the executive who is giving the instructions, then the underlying macro represents the programmer that actually performs the task. The RUN_MACRO function serves as a middleman, facilitating two-way communication between the calling data set and the macro. Unlike typical macros which generate data sets or output, macros used in RUN_MACRO functions generate *information*. By extracting values and metrics from the execution of a macro, RUN_MACRO routines give us the capability to answer high-level questions about data sets, variables, and entire analytical processes. Rather than Phone-a-Friend, think of RUN_MACRO routines as Phone-a-Programmer. Execution of a macro within the function does not affect the logic of the calling DATA step. Our function executes as normal and retains the property of encapsulation. Consider the alternative: a nested macro approach. If we use a macro do loop to fire off calls to a sub macro, we have to make sure our macro variables are of the proper

scope and resolve at the appropriate time for the whole process to work. Whereas RUN_MACRO routines have distinct layers –
DATA step and macro – which operate independently, communicating via the function. The advantages are clear. And we have seen that it takes only a few lines of code to create FCMP functions to wrap our macros.
The real question is, where are we going to find a data set with the list of parameters for our RUN_MACRO function?

CALLING DATA SET

RUN_MACRO function takes data set name and variable name as parameters.
Therefore, the calling data set must contain a list of data set names and variable names. Happily, a data set already exists that meets just these criteria – the SASHELP view of the dictionary table COLUMNS.

```
data metadata;
  set sashelp.vcolumn;
  where libname="SASHELP";

  nlevels = nlevels(memname, name);

run;
```

Memname	Name	NLevels
SASHELP.Class	Name	19
SASHELP.Class	Sex	2
SASHELP.Class	Age	6

Output 2. Output data set with NLEVELS values

CONCLUSION

The intention of this paper is to develop a dynamic programming structure via the RUN_MACRO function from PROC FCMP. Instead of using nested macro loops, we now have the ability to explicitly program at the meta-level using the familiar logic of the DATA step. Macros can return values to the DATA step, which can then compile these values as inputs for further processing.

REFERENCES

SAS Support user guide.
Dylan Ellis "Run Macro Run with proc FCMP"
Stacey M. Christian "Adding Statistical Functionality to the DATA Step with PROC FCMP"

ACKNOWLEDGMENTS

I would like to thank my managers Neelam, Namrata, Supriya and Michael for all the support.

RECOMMENDED READING

"To FREQ, Perchance to MEANS "by Christopher J. Bost.
"Run_Macro Run" by Dylan Ellis.

CONTACT INFORMATION

Author Name ; Keerthana Palwai
Company : Syneos Health
Address : Unit 301A & 301B,3rd Floor,Bldg.No.12B,Raheja Mind space,
Hitechcity, Madhapur, INDIA
City / Postcode : Hyderabad-500081
Work Phone: 04047572159
Fax:
Email:keerthipalwai@gmail.com
Web: