PharmaSUG China 2019 - Paper CC-050

Writing robust SAS macros

Yan Qiao, BeiGene, Beijing, China

ABSTRACT

A robust macro should be well documented and tested, should validate its inputs and handle failure in a controlled and elegant way, should notify the caller of the outcome of execution and clean up after itself, and should be easily maintained. This paper described specific techniques for achieving these.

INTRODUCTION

Many companies have their own SAS macros library, which can be used systematically and cover almost every part of programming work. It is important that each individual macro should be robust and reliable, and all the macros should interact in a predictable manner and work systematically. A robust macro should be well documented and tested, should validate its inputs and handle failure in a controlled and elegant way, should notify the caller of the outcome of execution and clean up after itself, and should be easily maintained.

This paper concentrates primarily on specific techniques of achieving these properties with the SAS Macro Language:

1. Defining parameters;
2. Parameter validation;
3. Checking and reacting to outcomes of global statements, data steps, procedure steps and macros calls;
4. Returning status to the caller;
5. Restoring the environment;
6. Testing and documentation.

DEFINING PARAMETERS

1. Use named parameters instead of positional parameters

The macro language allows parameters to de defined as positional or named, these are several advantages to use named parameters: a default value may be specified; the order in which the parameters are specified by the caller is irrelevant; it is easier to assign the corresponding values to a parameter, especially when there are a large number of parameters; new functionality can be added to a macro without requiring existing calls of the macro to be changed: additional parameters need only have a default which invokes the previous behavior.

2. Choose suitable parameter names

To choose the parameter names, a good guide is to follow the naming conventions used for options and statements in SAS procedures. For example, *data* for a primary input dataset, *out* for a primary output dataset, *infile* for a primary input external file, *file* for a primary output external file, *var* for a list of variables, or *by* for a list of variables to use for sorting.

3. Define valid values for parameter

For parameters which take a limited number of values, valid values should be defined in a consistent way. For example, parameters which act as switches should be consistently defined as having values as 1/0, Y/N, YES/NO. when such parameters are coded with character values, case should not be important.

PARAMETER VALIDATION

1. Required parameters are not missing

A general way to check that all required parameters have a value is to lop over a list of these parameters, checking that they have been specified. For example:

```
%local _pi _params _param rc;
%let _params=.data.out.by.;
%let _pi=1;
%do %while(%scan(&_params,&_pi,.)^=%str()) ;
    %let _param=%scan(&_params,&_pi,.);
    %if %nrbquote(&&&_param) = %str() %then %do ;
        %put ERROR: &sysmacroname: Parameter %upcase(&_param) is required.;
        %let rc=1;
        %end;
    %let _pi=%eval(&_pi+1) ;
    %end;
%if &rc=1 %then %return ;
```

We use the %nrbquote( ) function to prevent special characters I the argument from causing unexpected results.

2. Parameters take valid values

We also need to check that the validated parameters take valid values: the value is numeric or character; the value is in a specified list of range. When the parameter does not take valid value, the macro should report error and terminate prematurely. The macro should also inform the caller which values are valid for this parameter.

3. Existence of input

We need also check the existence of the input dataset or file. For example:

```
%if &_param=data %then %do ;
    %if %sysfunc(exist(&&&_param))=0 %then %do ;
        %put ERROR: &sysmacroname: %upcase(&&&_param) does not exist.;
        %let rc=2;
        %end ;
%end;
%if &rc=2 %then %return ;
```

CHECKING AND REACTING TO OUTCOMES

The make sure the macro executes fluently and identify error step if there is any, it is good practice to check that each global statement, data step or procedure step has run correctly before continuing with subsequent steps. SAS provides a number of automatic macro variables which may be used to check whether a step or global statement executed successfully. They are:

syserr containing a return code status set by some SAS procedures and the data step;

sysfilrc containing the return code from the last filename statement;

syslibrc containing the return code from the last libname statement.

In all three cases, a return code of 0 signifies success. For syserr, values less than or equal to 4 signify warnings, while values above 4 are errors. We can use syserr to identify unexpected error after each step. For example:

```
proc sort data=&data out=&out ;
  by &by ;
```

```
  %if (%quote(&where) ^= %str()) %then %do ;
      where &where ;
    %end ;
  run ;
%if &syserr > 4 %then %do ;
    %put ERROR: &sysmacroname: error sorting the dataset. ;
    %return ;
    %end;
```

If there is an error in the sort step, the macro exits prematurely, avoiding a needless execution of the subsequent data step.

RETURN CODE

In most programming languages, when defining a function the language offers a way for the function to provide a return value. We can also do so when defining SAS macros. A possible implementation of returning status is shown in the following example:

```
%macro finddups(data=_last_,out=_dups,by=, rc=rc_finddups) ;
    %if %quote(&rc) = %str() %then %do ;
        %put NOTE: &sysmacroname: Using RC_&sysmacroname for the RC parameter ;
        %let rc=rc_&sysmacroname ;
        %end ;
  %global &rc ;
  %let &rc=9999 ;

  proc sort data=&data out=&out ;
    by &by ;
   run ;
   %if &syserr>4 %then %do ;
       %let &rc=1 ;
      %return ;
      %end ;
    ...
    %let &rc=0;
%mend;
```

We use rc finddups as the default value for the return code parameter rc. If it has been set to missing, we reset it to the default. Further, we adopt the convention that a return code of zero indicates success. If an error occurs in the sort step, we set a non-zero return code and exit. In order to eliminate the possibility of incorrectly returning zero when a problem happens which our code does not detect, we initialize the return code to a suitable non-zero value, in this case 9999. If all steps in the macro execute successfully, the return code is set to zero as the last action in the macro.

RESTORING THE ENVIRONMENT

It is good practice for a macro to restore the environment to the state before the macro was called, whether the macro finishes execution or terminates due to error. At the point where we decide on premature termination, we need to consider whether clean-up is necessary. For example, in parameter validation section, it is safe to use %return statement for premature termination because no change has been made to the environment and no clean-up is needed. However once we have created temporary data sets, set options, or assigned library or file references, we must use the %goto statement to direct to the end of the program and then clear the environment. For example:

```
proc sort data=&data out=_finddups_tmp ;
      by &by ;
```

```
    run ;
%if &syserr>4 %then %do ;
    %let &rc=2 ;
    %goto DONE ;
    %end ;

data &out ;
     set _finddups_tmp ;
    ...
   run ;
%if &syserr>4 %then %do ;
   %let &rc=3 ;
   %goto DONE ;
   %end ;

%let &rc=0 ;
%DONE:
%if %sysfunc(exist(_finddups_tmp)) %then %do ;
    proc datasets lib=work nolist ;
        delete _finddups_tmp;
        quit ;
   %end ;

%mend;
```

TESTING AND DOCUMENTATION

Good documentation can contribute to the robustness of a macro by reducing the likelihood that the macro will be called with inappropriate parameters. In order to achieve this goal, documentation needs to cover a number of different aspects. First and foremost, a concise description of the purpose of the macro should give the user or prospective user a clear idea of what the program does and the circumstances in which is may be used, allowing the user to decide if the macro is appropriate for the task in hand. Secondly, the parameters must be well described including sufficient detail on type, expected values and default values. Possible interactions between parameters should also be addressed. Thirdly, any inputs to the macro, in particular files or data sets, should be clearly described, including specification of relevant structural or variable attributes. Fourthly, a sufficiently detailed description of the outputs of the macro should be provided, covering any type of output produced. Finally, at least one realistic example should be provided. Unless relevant to the use of the macro,the documentation should not describe the technical implementation details, although this may be provided as an appendix or, preferably, as a separate document.

We also recommend writing the user documentation before the macro is written. Since the documentation describes the user interface in detail, this may serve as the specification for the macro.

CONCLUSION

This paper describes some useful techniques to develop robust SAS macros. Robust macros can find and report problems, instead of producing meaningless results. They can notify the caller about the problem and guide the caller on how to use them correctly. Moreover, they are easily maintained and need significantly less support.

REFERENCES

Martin, G. 2009 “Techniques for writing robust SAS macros” PhUSE Global 2009 Conference

Available at https://www.phusewiki.org/docs/2009%20PAPERS/AD01.pdf

CONTACT INFORMATION

Yan Qiao

Principle Statistical Programmer, BeiGene

yan.qiao@beigene.com