PhUSE EU Connect 2018

Paper CT07

Make Your Macro Great from the Very Beginning

Yuliia Bahatska, Syneos Health, Berlin, Germany
Vladlen Ivanushkin, DataFocus, Berlin, Germany

ABSTRACT

If someone asked me to describe my first macro with one word this word would be “clumsy”. It did its job, but it was not elegant, nor was it flexible and I am very happy it is now safely buried under tons of code in some old project where no one would ever find it. But does it really have to be like this? In this paper I would like to share with you the advice I would have given to myself six years ago to make my macro better from the very beginning. These hints are not applicable in every single situation, but they definitely help to increase flexibility, reusability and performance of the macros.

INTRODUCTION

Macro programming is the area of SAS® programming where flexibility and reusability play the key part. The more flexible your macro is the less effort you need to use it again. In this paper I will not describe all the steps of writing a macro from the beginning to the final %MEND statement but rather will concentrate on the pieces of code that can be included in the macro so that it can be reused in further projects with no or minimum modifications and cover more scenarios and programming needs.

THE COUNTW FUNCTION

In our work we often have a situation when we need to repeat the same outputs using different subgroup variables. Of course, if there are two such variables it’s easy to write two macro calls, but what if there are ten? In this situation your macro would benefit from having a parameter that can receive a list of values, so your macro call would look like this:

```
%subgroup_analysis(data=data1, subgroups=subgroup1 subgroup2 subgroup3);
```

To make this work you can use the COUNTW function in combination with the %SYSFUNC. The COUNTW function does exactly what it sounds like: it counts words and the %SYSFUNC enables it to work in macro language. As a result, you can parse the macro parameter subgroup (for example, using the %SCAN macro function) and repeat the required steps for each of its values:

```
%macro subgroup_analysis(data=, subgroups=);
   %do i=1 %to %sysfunc(countw(&subgroups));
       %local subgroup;
       /*get a value number &i from the subgroup parameter*/
       %let subgroup=%scan(&subgroups, &i);


     <And here you can include the code that needs to be repeated for the specific
     value of subgroup.>
     …
   %end;
%mend subgroup_analysis;
```

This doesn’t require too much effort but will definitely make your macro more flexible.

USE OF REGULAR EXPRESSIONS

Knowing just basics of the regular expressions can save really a lot of time and make the programmers’ life easier, especially while developing macros. There are numerous situations when a regular expression may be a good solution and here are some useful ones for macro programming.

CHANGING DELIMITERS IN A STRING

It may happen that the macro parameter that can receive a list of values doesn’t need to be parsed but rather is used as is, for example if it receives a list of one or more by-variables. Before I read about this trick I always tried to avoid using the PROC SQL and to use the DATA step or other procedures instead so that I didn’t have to use a comma between the values. This was often frustrating, especially when many-to-many merge was needed. However, using a simple regular expression solves this problem: you can create another local macro variable that would contain a comma-delimited list of the same values. If byvar is a macro parameter that can receive a list of values, the code below will help to create a supplementary local macro variable, which would contain the same list from byvar but comma-delimited:

```
%local supp_byvar;
```

```
%let supp_byvar=%sysfunc(prxchange(s/\s+/%str(,)/, -1, &byvar));
```

The regular expression above replaces one or more blanks between the values with a comma. For example, the value of 'Ab C D_e F' will be changed to 'Ab,C,D_e,F. In this particular case the combination of the %SYSFUNC and the standard SAS functions may be applicable, however in more complicated situations getting the required result using just the standard functions will be quite difficult. At the same time, with the regular expression – it is still easy enough. Below you can see the example of the PROC SQL that uses &supp_byvar:

```
proc sql noprint;
   create table example as
   select count(distinct usubjid) as count, &supp_byvar
   from adsl
   group by &supp_byvar;
quit;
```

SEARCHING WORDS IN A STRING

It's also quite often required to search some substring in a string. And, instead of writing complex combinations of SAS functions, it may be more efficient and handier to use a regular expression as this is one of their purposes. For example, if you need to know whether your string contains some word from a list, instead of writing cumbersome code like

```
%if %sysfunc(findw(&string, WORD1)) or %sysfunc(findw(&string, WORD2)) or …
   or %sysfunc(findw(&string, WORDN))… ,
```

you can replace this with

```
%if %sysfunc(prxmatch(/\b(WORD1|WORD2|…|WORDN)\b/, &string))… .
```

If the required list of words is given as a macro-parameter, you can easily use it for the regular expression regardless of the delimiters. Just insert | symbols using either regular expression like below:

```
%sysfunc(prxchange(s/\s+/|/, -1, &list_of_words))
```

Or a combination of standard SAS functions:

```
%sysfunc(tranwrd(%cmpres(&list_of_words), %str( ), |))
```

Below you can see a sample macro to check if the string contains the words of interest:

```
%macro find_words(string=, words=);
    %local wrd_ex_loop wrd_ex_RE;
    %let wrd_ex_loop=NO;
    %let wrd_ex_RE=NO;

    /*To check whether some word exists in the sentence, we can create a do loop
      and check each word step-by-step*/
    %do counter=1 %to %sysfunc(countw(&words));
        %let word=%scan(&words, &counter);
        %if %sysfunc(findw(&string, &word)) %then %let wrd_ex_loop=YES;
    %end;
    %put NOTE: Does at least one word exist (do loop)? - &wrd_ex_loop;

    /*However more efficient and easier way is to use the regular expressions*/
    %let wrds_RE=%sysfunc(prxchange(s/\s+/|/, -1, &words));
    %if %sysfunc(prxmatch(/\b(&wrds_RE)\b/, &string)) %then %let wrd_ex_RE=YES;
    %put NOTE: Does at least one word exist (regular expression)? -&wrd_ex_RE;
%mend find_words;

%find_words(string=%str(STRING with WORD1, WORD2, WORD55 etc.), words=WORD5 WORD7);
```

This macro uses both a combination of SAS functions and a regular expression to check if the words are contained in the string. If you run the code above, you will get the following results:

NOTE: Does at least one word exist (do loop)? - NO
NOTE: Does at least one word exist (regular expression)? NO

SUBTRACTION OF TWO STRINGS

Another example that may be useful is when you are given two sets of words and you need to get a subtraction of them. It may be quite difficult (or would require a lot of operations) to achieve this using only the SAS functions, however it may be easy using the regular expressions.

```
%let words=WORD1 WORD2 WORD3 WORD4 WORD5;
%let words_to_ignore=WORD4 WORD1;
```

```
/*Insert | symbols to use in regular expression*/
%let words_to_ignore_del=%sysfunc(prxchange(s/\s+/|/, -1, &words_to_ignore));
%put %cmpres(%sysfunc(prxchange(s/\b(&words_to_ignore_del)\b//, -1, &words)));
```

The following is written to the log - `WORD2 WORD3 WORD5.`
So it's really simple to exclude or replace within some text a set of words from a string using the regular expressions.

THE GETOPTION FUNCTION

Sometimes you might need to change some SAS options inside your macro. However, one of the important rules of macro programming is to leave the SAS session the way it was before the execution of the macro. In this situation a helpful technique is to use the GETOPTION function in order to save the option in a local macro variable and then to set it back in the end of the macro execution. The GETOPTION function returns the value of a SAS system or graphics option. The example below illustrates how it helps to keep the initial options untouched. Let's assume for some reasons you need to change page size inside your macro:

```
%local l_pagesize;
%let l_pagesize=%sysfunc(getoption(PAGESIZE));
options pagesize=60;
…
options pagesize=&l_pagesize;
```

This way the initial value of pagesize is stored in macro variable l_pagesize and when the job of your macro is done you can easily restore original pagesize setting. Of course, this is also applicable for other options and using this trick helps to avoid unexpected surprises during your programming.

Leaving the SAS environment unchanged, as best as possible, is not limited to restoring session options. It is usually a good idea to clean up the intermediate datasets that your macro created. However as you might need to look into these datasets during debugging, the solution here might be to have a macro parameter that is responsible for saving or deleting the supplementary datasets after the macro execution. To delete all datasets created by the macro it is helpful to have their names start with the same prefix and then apply the PROC DATASETS:

```
proc datasets memtype=data nolist nowarn;
   delete _temp: <names of all macro-created temporary datasets>;
quit;
```

COMPLEX MACRO PARAMETERS

In case you need two sets of related parameters, like a list of datasets related to a list of conditions or variables, it may be a good idea to use a complex macro parameter. The idea is to create this parameter in a way that related single parts are ordered in a desired way using pre-specified delimiters. As an example, all datasets have to be sorted in the same way but with some exceptions:

```
%macro sort_variables(datasets=, sort_vars=, exception=);
   %do counter=1 %to %sysfunc(countw(&datasets));
      /*Retrieve single dataset from the list*/
      %let dataset=%scan(&datasets, &counter);

      /*If a dataset name is found in the exception rules, then get the sort
        variables for the exception*/
      %if %index(&exception, &dataset.:) %then %do;
         /*Remove the dataset name and all preceding text from the exception
           text*/
         %let _sort_except=%sysfunc(prxchange(s/.*&dataset.://, 1, &exception));
         /*Select the part before # symbol which is the list of vars for sort*/
         %let sort_except=%scan(&_sort_except, 1, #);
      %end;
      %else %let sort_except=;

      /*Use exception rules if not missing, otherwise use the sort_vars macro var*/
      proc sort data=&dataset out=&dataset._srt;
         by %sysfunc(coalescec(&sort_except, &sort_vars));
      run;
   %end;
%mend sort_variables;
```

Here is how an example macro call could look like:

```
%sort_variables(
   datasets  = DATA1 DATA2 DATA3 DATA4 DATA5,
   sort_vars = VAR1 VAR2,
```

```
      exception = DATA3: VAR1 VAR3#DATA5: VAR2 VAR4)
```

The macro results in a PORC SORT by variables VAR1 VAR2 for datasets DATA1, DATA2 and DATA4, but by VAR1 VAR3 for DATA3 and by VAR2 VAR4 for DATA5.
Of course, any delimiters can be used, they just must be consistent with the delimiters used within the macro code.

MACRO QUOTING/UNQUOTING

If you need to use a macro parameter and set the default value of this parameter to a macro variable which is only resolves during the macro execution, using quoting function is necessary. For instance, a macro for a set of tables or listings is needed and the footnote should contain some value from a data and we use the footnote text itself as a macro parameter. The idea is shown in the schematic macro below:

```
/*To avoid any problems, we can just use the NRSTR function when initializing the
  default parameter */
%macro print_note(data=, footnote=text from the dataset - %nrstr(&studyid));
   proc sql;
      select studyid into: studyid from &data;
   quit;

   /* UNQUOTE should be used to resolve the previously quoted macro parameter*/
   %put NOTE: %unquote(&footnote);
%mend print_note;

data data1;
   studyid='12345';
run;

%print_note(data=data1, footnote = text from the dataset – %nrstr(&studyid));
```

Keep in mind that if you don't quote it you will get the WARNING:

```
WARNING: Apparent symbolic reference STUDYID not resolved.
```

Also if you don't unquote it when you need to use it, you will get "text from the dataset - &studyid" instead of the expected "text from the dataset – 12345".

MACRO PARAMETER CHECKS

It is good programming practice to include macro parameter checks. They can really help you debug and develop your macro and save quite some time. In case some check fails it's much more effective to get a message about the problem instead of waiting until your macro executes and gives you an error. And after that you will need to spend time to identify the problem. Also, it is a good idea to include the %GOTO operator in case an issue is found in some macro parameter to avoid execution of the macro with a wrong parameter and losing time.
The checks are specific to each macro, however here are a couple of general checks which are used quite often:

- a macro parameter is not empty for cases when it must be specified;
- a dataset/directory/library/format exists;
- a variable specified in a parameter is of required type. Like NUMBER, YESNO, CHAR etc.;
- a variable exists in a dataset (if the source dataset is known).

Let's use the example below to illustrate this:

```
%macro mparam_check(data=, make_print=) / minoperator;
   %local param_error;
   %let param_error=0;

   /*Make the macro parameters case insensitive*/
   %let data=%upcase(&data);
   %let make_print=%upcase(&make_print);

   /*Harmonize the parameter*/
   %if &make_print=YES %then %let make_print=Y;
   %else %if &make_print=NO %then %let make_print=N;

   /*Check whether the DATA parameter is not empty*/
   %if &data= %then %do;
       %put ERROR: Macro parameter DATA is empty, please specify!;
       %let param_error=1;
   %end;
   /*In case it is not empty, check whether a dataset exists*/
   %else %if ^%sysfunc(exist(&data)) %then %do;
```

```
      %put ERROR: The dataset specified in macro parameter DATA does not exist,
      please check!;
      %let param_error=1;
   %end;

   /*Check whether the specified parameter value is assigned as desired*/
   %if ^(&make_print in (Y N)) %then %do;
      %put ERROR: Macro parameter MAKE_PRINT has wrong format or empty, please
      update!;
      %let param_error=1;
   %end;

   /*In case any error is found, print a message and terminate the macro*/
   %if &param_error=1 %then %do;
      %put ERROR: The macro terminated due to errors in the macro parameters.;
      %goto exit;
   %end;

   <code to be performed based on (verified) macro parameter values>

   %exit:
%mend mparam_check;

%mparam_check(data = data, make_print=yesno);
```

If you run this macro with the parameters as above you will get the following results:

ERROR: The dataset specified in macro parameter DATA does not exist, please check!
ERROR: Macro parameter MAKE_PRINT has wrong format or empty, please update!
ERROR: The macro terminated due to errors in the macro parameters.

If any check fails, the information is given in the log and the macro execution is aborted. It's worth mentioning that for this example the MINOPERATOR option is used which allows the macro processor to recognize the IN logical operator. The MINOPERATOR option is available starting from SAS 9.2.

STANDARDIZATION OF MACRO PARAMETER VALUES

Another helpful handling of macro-parameters is omitting typos like double blanks or incorrect case. So, for some parameters which get text values it may be useful to standardize the values at the beginning of the macro by removing redundant blanks with the %CMPRES function and/or upcasing the text with the %UPCASE function. Also, it often makes sense to harmonize parameter values, like in the example above (i.e. YES to Y and NO to N for macro parameter MAKE_PRINT). This will help to avoid situations when a macro fails to run as expected because some of the parameters have been assigned the values in a wrong way.

CONCLUSION

The list of the tricks described in this paper can, of course, be expanded but from my experience they have proved to be the most frequently used and the most helpful ones. I hope that using this paper will enable the programmers to make their code more efficient and flexible so that having written the macro once they would be able to use it again and again.

CONTACT INFORMATION

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

Yuliia Bahatska
Syneos Health
Joachimstaler Str. 10–12,
10719 / Berlin, Germany
Email: yuliia.bahatska@syneoshealth.com

Vladlen Ivanushkin
DataFocus GmbH
Lothringer Str. 23,
D-50677 Cologne
Email: vladlen.ivanushkin@datafocus.de