Paper 3580-2019

Macros I Use Every Day (And You Can, Too!)

Joe DeShon

ABSTRACT

SAS® macros are a powerful tool that can be used in all stages of SAS program development. Like most programmers, I have collected a set of macros that are so vital to my program development that I find that I use them just about every day. In this paper, I demonstrate the SAS macros that I developed and use myself. I hope they will serve as an inspiration for SAS programmers to develop their own set of must-have macros that they can use every day.

INTRODUCTION

The SAS Macro facility is best thought of as a text-substitution tool in Base SAS. Tasks that are repetitive, complicated, or confusing can be helped by writing a macro to do the bulk of the work. Then you can simply pass parameters to that macro and let it do the "heavy lifting" for you.

There are a few tasks that I do just about every day. I found it best to write macros to help me with those tasks. Some of them may seem rather simple or mundane, but I found them to be extremely useful. I hope that you will either use these macros "as-is", or that they will serve as an inspiration for you to write macros that will help you in your job.

SCOPE AND AUDIENCE OF THE PAPER

This paper assumes a fundamental knowledge of Base SAS programming and of the SAS Macro facility.

There are several ways a SAS macro can be defined to a program: it can be defined in-line within the source code, it can be defined as part of an AUTOCALL option, or it can be included in a %include statement. This paper will assume that the macros are defined in-line within the code of the program. But certainly, if the macro is used practically every day, it should be defined in some sort of common library and loaded automatically.

TWO TYPES OF MACROS

SAS macros tend to fall in one of two general groups: *function* macros and *statement* macros.

Function macros tend to emulate SAS numeric or string functions. They take from one to many parameters, perform some sort of calculation or transformation, and then return a single value -- just as a SAS function does. They are often used inside a SAS assign statement.

Statement macros take on the form of a SAS statement. They may or may not accept parameters. They may return a single SAS statement, or they may return several SAS statements, sometimes constructing entire data steps, procedures, or even entire work flows -- all from a single SAS macro statement.

There are examples of both types of macros in this paper.

%ROUND2PENNY

When doing financial calculations such as calculating averages, interest, or discounts, the results are often figured well beyond two decimal points to the dollar. By default, SAS retains all numeric variables as floating point decimals (rather than with fixed decimals), so SAS programmers can decide if and when to round the results to the nearest penny.

Rounding is a trivial exercise in SAS, using the ROUND() function. In fact, SAS's ROUND() function is superior to similar functions in other languages because you can round to any number: to the nearest penny, nickel, quarter, or even to the nearest seven dollars or any other number:

```
FUNCTION              RESULT
--------------------  ----------------------------------------------
round(12345.789,0.01)  12345.68  (Round to the nearest penny)
round(12345.789,0.05)  12345.80  (Round to the nearest nickel)
round(12345.789,0.25)  12345.75  (Round to the nearest quarter)
round(12345.789,7)     12348.00  (Round to the nearest multiple of 7)
```

If it's so easy and flexible, why create a special macro just for rounding to a penny?

First, because it explicitly documents what is going on. If a calculation of a value happens over several steps, it is important that the rounding occurs at a specific, deliberate spot -- usually as one of the last calculations performed. During development and debugging, there may be a question about when rounding occurs. A ROUND() function could easily get lost in the maze of other calculations. But the %round2penny() macro is easy to find and explicitly notes the purpose of the process.

Second, when the ROUND() function is used to store the result back into the original value, the variable must be referenced twice:

```
avg_month_sales_01 = round(avg_month_sales_01,0.01);
```

That can be a problem when many variables are to be rounded at the same time. Can you find the bug in the following program?

```
avg_month_sales_01 = round(avg_month_sales_01,0.01);
avg_month_sales_02 = round(avg_month_sales_02,0.01);
avg_month_sales_03 = round(avg_month_sales_03,0.01);
avg_month_sales_04 = round(avg_month_sales_04,0.01);
avg_month_sales_05 = round(avg_month_sales_05,0.01);
avg_month_sales_06 = round(avg_month_sales_06,0.01);
avg_month_sales_07 = round(avg_month_sales_01,0.01);
avg_month_sales_08 = round(avg_month_sales_08,0.01);
avg_month_sales_09 = round(avg_month_sales_09,0.01);
avg_month_sales_10 = round(avg_month_sales_10,0.01);
avg_month_sales_11 = round(avg_month_sales_11,0.01);
avg_month_sales_12 = round(avg_month_sales_12,0.01);
```

Obviously, putting *January* sales into the *July* variable is not what was intended, but in the maze of numbers, that could easily be overlooked.

It is much more explicit to have a macro handle both sides of the equation:

```
%round2penny(avg_month_sales_01);
%round2penny(avg_month_sales_02);
%round2penny(avg_month_sales_03);
%round2penny(avg_month_sales_04);
%round2penny(avg_month_sales_05);
%round2penny(avg_month_sales_06);
%round2penny(avg_month_sales_07);
%round2penny(avg_month_sales_08);
```

```
%round2penny(avg_month_sales_09);
%round2penny(avg_month_sales_10);
%round2penny(avg_month_sales_11);
%round2penny(avg_month_sales_12);
```

Following is the macro code:

```
/*************************************************************************/
/* round2penny.sas --                                                    */
/* Create a macro that rounds a value to the nearest penny (0.01).       */
/*************************************************************************/

%macro round2penny(whatvar);
  if &whatvar ne . then &whatvar = round(&whatvar,0.01);
%mend;
```

This is a "statement" macro, so it should be used as a stand-alone statement in a data step. The value of the variable is changed in the process:

```
data _null_;
  sales_01 = 123.01;
  sales_02 = 123.01;
  sales_03 = 789.99;
  avg_sales = mean(sales_01,sales_02,sales_03);
  putlog avg_sales=;
  %round2penny(avg_sales);
  putlog avg_sales=;
run;
```

The resulting log:

```
avg_sales=345.33666667
avg_sales=345.34
```

%EXPO_ROUND(VAR)

A special case of rounding occurs when you don't know exactly how many digits you want to round to. Typically, the larger the number, the greater the number you want to round to.

For example, if the number is in the hundreds, you may want to round to the nearest hundred. But you may want to round a number in the thousands to the nearest thousand. Or the nearest ten thousand, and so on.

This is the concept of "exponential rounding", where the number being rounded to is proportional to the number that is being rounded. It is useful in statistics to "bin" or to categorize numbers. It can also be used to give a general idea of the number on a report (for example, a sales amount) without reporting the exact number.

The %expo_round function does just that. It works just like the %round_to_penny macro, except that it rounds to various integers, depending on the size of the number being rounded:

```
/*************************************************************************/
/* expo_round.sas --                                                     */
/* This macro performs a form of expoential rounding.                    */
/* It rounds numbers to increasingly larger values as their value        */
/* It can be used for reporting sales when an exact number is not needed */
/* or when giving an exact number might be more confusing.               */
/* This is a FUNCTION-type macro.                                        */
/* USAGE:                                                                */
/*  approx_sales = %expo_rounded(sales);                                 */
/*************************************************************************/

%macro expo_round(what_var);
  ifn(&what_var lt     1000,round(&what_var,     100),
  ifn(&what_var lt    10000,round(&what_var,    1000),
  ifn(&what_var lt   100000,round(&what_var,   10000),
  ifn(&what_var lt  1000000,round(&what_var,  100000),
  ifn(&what_var lt 10000000,round(&what_var, 1000000),
                            round(&what_var,10000000),.)))))
%mend;
```

Notice that this always leaves one significant digit followed by multiple zeros:

```
data _null_;
  a = %expo_round(123);
  b = %expo_round(2234);
  c = %expo_round(32345);
  d = %expo_round(423456);
  e = %expo_round(5234567);
  putlog a= b= c= d= e= ;
run;
```

The resulting log:

```
a=100 b=2000 c=30000 d=400000 e=5000000
```

It would be easy to change the code so that the result is two significant digits, if that is desired.

%MISS2ZERO / %ZERO2MISS AND %MISSNOTZERO / %ZERONOTMISS

In SAS programming, you often run across numeric values that are "missing" (many other systems use the word "null").

Missing values may be generated by:

- Performing an illegal math operation, such as dividing by zero
- An SQL JOIN or a data step MERGE, where the value of the key is in one file, but not in another
- Any of several math functions that create missing values in some circumstance

Sometimes it is desired to automatically convert missing values to zero (or the other way around). And sometimes it is desired to treat a missing value as if it were zero (or the other way around) without *actually changing* the value of the variable.

The following four macros do just that.

Suppose you want to add the values in the variables A and B, and place them in the variable C. You know that B may be missing in some cases. If it is missing, the value in C would be missing, regardless of the value in A. But suppose you want to treat the case as if B is zero – even if it is actually missing. The %zeronotmiss macro does that for you:

```
C = A + %zeronotmiss(B);
```

In the above case, if the variable B is missing, it is treated as if it were zero. The value of B is not changed; only in the context of this statement is it treated differently.

Suppose, however, you want to convert B to zero if it is found to be missing. That could be accomplished with the following:

```
If B = . then B = .;
```

But, much like the rounding examples given previously, that requires putting the variable name on both sides of the equation. It's possible that a type could go unnoticed.

Much more direct is the %miss2zero macro, which changes the value of the variable to zero if it is found to be missing.

%miss2zero(B);

Notice that the %zeronotmiss macro is a *function* macro; it returns a value but does not actually change the value of the of the variable. The syntax is similar to that of a SAS *function*.

The %miss2zero macro is a *statement* macro; it actually changes the value of the variable. The syntax is similar to that of a SAS *statement*.

These two macros have "cousins" which operate the same way, but the other way around: %missnotzero and %zero2miss.

The source for the four macros follows:

```
/*************************************************************************/
/* miss2zero.sas --                                                      */
/* Create a macro that flips missings to zero.                           */
/* This macro actually CHANGES the value of the variable.                */
/* If you don't want to change the value, but merely USE a zero instead  */
/* of a missing value, use the macro zeronotmiss instead.                */
/*************************************************************************/

%macro miss2zero(whatvar);
  if &whatvar = . then &whatvar = 0;
%mend;
```

```
/*************************************************************************/
/* zero2miss.sas --                                                      */
/* Create a macro that flips zeroes to missing.                          */
/* This macro actually CHANGES the value of the variable.                */
/* If you don't want to change the value, but merely USE a missing value */
/* instead of a zero, use the macro missnotzero instead.                 */
/*************************************************************************/

%macro zero2miss(whatvar);
  if &whatvar = 0 then &whatvar = .;
%mend;
```

```
*********************************************************************/
/* missnotzero.sas --                                                */
/* Create a macro that uses a missing value if the value passed to it is  */
/* zero.                                                             */
/* This macro actually DOESN'T CHANGE the value of the variable, it only  */
/* uses a missing value if it was given a zero.                      */
/* If you want to CHANGE the value, use the macro zero2miss instead.      */
/*********************************************************************/

%macro missnotzero(whatvar);
  choosen((&whatvar=0)+1,&whatvar,.)
%mend;
```

```
/*********************************************************************/
/* zeronotmiss.sas --                                                */
/* Create a macro that uses zero if the value passed to it is missing.    */
/* This macro actually DOESN'T CHANGE the value of the variable, it only  */
/* uses a zero if it was given a missing value.                      */
/* If you want to CHANGE the value, use the macro miss2zero instead.      */
/*********************************************************************/

%macro zeronotmiss(whatvar);
  choosen((&whatvar=.)+1,&whatvar,0)
%mend;
```

%FILE_DATE(FILE_PATH_AND_NAME)

Many of my projects involve importing data in Excel spreadsheets that had been created by a variety of sources. It is important that the file being imported is the most recent file. So I needed an easy way of determining the age of the file.

The SAS FINFO function served my purpose well, but I was always confused by the syntax. It required a file to be opened with an FOPEN function to get a file ID. And I had a hard time remembering that "Last modified" was the way to identify my desired statistic of that file.

The %file_date macro handles all that automatically. Simply supply the fully-qualified path and name of the file and it will return the date the file was last modified.

A common use is to subtract that date from TODAY() to get the age of the file in days. Since this is a *function* macro, it can be embedded inside a complete SAS statement as follows:

```
if today() - %file_date("c:\my_path\foo.txt") gt 10 then do;
  putlog "ERROR: File is too old to be any good!";
  abort;
  end;
```

The complete code follows:

```
/*************************************************************************/
/* file_date.sas --                                                      */
/* This is a "function" macro to easily determine the update date of any */
/* file.  It can be used to test the age of an imported spreadsheet to   */
/* make sure it is a reasonably-recent version.                          */
/* USAGE: my_file_date = %file_date("fully_qualified_path_and_name");    */
/* The path and file may or may not be quoted.                           */
/*************************************************************************/

%macro file_date(file_path_and_name);
  ((filename("_fd&sysjobid","%bquote(&file_path_and_name)")*0) +
  input(finfo(fopen ("_fd&sysjobid"), "Last modified"),anydtdte18.))
%mend;
```

%RAND_BETWEEN(NUMBER1,NUMBER2)

SAS has several functions that generate random numbers. But they usually make random numbers between zero and one. Most of the time, I need a random integer between one and *n*. The syntax to generate such numbers is difficult to memorize. So I borrowed a trick from Excel and created a macro that duplicates the Excel rand_between() function. You simply pass it a high and low parameter and it will generate a random number greater than or equal to the low number and less than or equal to the high number.

Since this is a function macro, it can be used in the middle of a SAS statement:

```
a = 1;
b = 10;
if %rand_between(a,b) = 7 then do;
  putlog "NOTE: I just hit a lucky seven!";
  end;
```

Following is the source code:

```
/*************************************************************************/
/* rand_between.sas --                                                   */
/* This macro returns a random integer between two numbers.              */
/* USAGE:                                                                */
/*    e.g. to return a random string 'A', 'B', 'C', or 'D':              */
/*      mystring = substr('ABCD',%rand_between(1,4),1);                  */
/* NOTES:                                                                */
/* The macro acts like a regular SAS function and can be used in any     */
/* expression that needs a numeric value.                                */
/* It expects two numbers as parameters.                                 */
/*************************************************************************/

%macro rand_between(number1,number2);
  choosen((&number1=. or &number2=.)+1,
    int(ranuni(0)*(max(int(&number1),int(&number2))-
min(int(&number1),int(&number2))+1))+min(int(&number1),int(&number2))
    ,.)
%mend;
```

%GET_SAS_PROGNAME

It is often handy to know the name of the program that is currently running. SAS offers two Windows environment variables that give this information: SAS_EXECFILEPATH and SAS_EXECFILENAME. They can be accessed through the %sysget macro function.

But since they are system-wide environment variables, they only give information on the most recent SAS program being executed. This can be a problem if multiple sessions are running at the same time.

To remedy this, the %get_sas_progname macro puts both of those environment variables into SAS macro variables.

If this macro is placed at the top of the program, the macro variables will always be available within the SAS program. They can be placed in headings of a printed report, put in the SAS log for later analysis, or tested to be sure the correct version of the program is running.

The usage is to simply place the macro anywhere in open code. The variables will be displayed in the SAS log and will be assigned and available for use later in the program:

```
%get_sas_progname;
proc print data = sashelp.class;
  title 'PROGRAM: ' "&sas_execfilename";
run;
```

Following is the source code:

```
/*************************************************************************/
/* get_sas_progname.sas --                                               */
/* This program gets the name and path of the program currently being    */
/* executed from the Windows environment variables.  It needs to be run  */
/* at the very top of the program.  The Windows environment variables may */
/* be changed every time a SAS program is initiated in different         */
/* sessions.  This program "freezes" the values in SAS macro variables so */
/* they will be the same for any given session throughout the run of the  */
/* program.                                                              */
/*************************************************************************/

%macro get_sas_progname;
  %global sas_execfilepath sas_execfilename;
  %let sas_execfilepath = %sysget(SAS_EXECFILEPATH);
  %let sas_execfilename = %sysget(SAS_EXECFILENAME);
  %put NOTE: Macro variable %str(&)sas_execfilepath = &sas_execfilepath..;
  %put NOTE: Macro variable %str(&)sas_execfilename = &sas_execfilename..;
%mend;
```

%SWAP(VAR1,VAR2)

There are often times when it is necessary to interchange or "swap" the values of two variables with each other. This may be necessary when writing a customized sorting or categorizing algorithm.

Some languages offer this feature as a swap function. There is no such function in Base SAS. Usually, the solution is to write lines of code that first place one of the values in a temporary variable, move the values around appropriately, and then drop the temporary variable as follows:

```
temp_var = a;
a = b
b = temp_var
drop temp_var;
```

This is an awkward and error-prone solution, especially if many values need to be swapped at the same time. It could be improved by creating a macro to do the work, but there is still

the problem that a temporary variable needs to be created and dropped. And there are no controls on abusing the process by attempting to swap to variables of different types or lengths.

This problem was discussed in the following SAS Communities forum:

https://communities.sas.com/t5/SASware-Ballot-Ideas/Create-a-new-function-SWAP/idi-p/327598

It was suggested that the ALLPERM function could be used. If there are only two items in the list (as there would be during any swap), the second permutation is always just the reverse of the first, effectively swapping the values of the two items in the list.

The macro in this paper duplicates the "swap" function of other languages, using the ALLPERM function. It also makes all the appropriate checks for variable type and size.

Following is the source code:

```
/*************************************************************************/
/* swap.sas --                                                           */
/* Use call ALLPERM to swap two values without having to create a        */
/*  temp variable.                                                       */
/* ALLPERM checks to make sure the variables are the same type and same  */
/* length.                                                               */
/* Even though ALLPERM checks for different types and lengths, the error */
/* message isn't very helpful, because it refers to ALLPERM, but doesn't */
/* give  line number.  Also, the programmer may not be aware that the    */
/* SWAP macro even calls ALLPERM.  So this macro delivers a slightly more */
/* meaningful message, giving the names of the variables and the fact    */
/* that it is the SWAP macro causing the error.  It then passes control  */
/* to CALL ALLPERM, which will abort the data step.                      */
/* Notice that the above condition is a RUNTIME error, not a COMPILE     */
/* error.                                                                */
/**
    https://communities.sas.com/t5/SASware-Ballot-Ideas/Create-a-new-
function-SWAP/idi-p/327598
**/
/*************************************************************************/

%macro swap(var1,var2);
  /** When there are only two permutations,                  **/
  /** The SECOND permutation is the reverse of the ORIGINAL! **/
  if vtype(&var1) ne vtype(&var2) then do;
    putlog "ERROR: SWAP macro called for variables &var1 and &var2, but they
are different types!";
    end;
  if vlength(&var1) ne vlength(&var2) then do;
    putlog "ERROR: SWAP macro called for variables &var1 and &var2, but they
are different lengths!";
    end;
  call allperm(2,&var1,&var2);
%mend;
```

%MAKE_VAR(VAR,LEN,LOGOPTION)

Here's an example of a macro that is primarily used by other macros.

Sometimes, I write macros that will be inserted in the data steps of many different programs. Occasionally those macros need to create temporary variables for their own use.

These macros must be able to create variables that do not duplicate variable names that already exist in the data step.

One way of solving this problem is to create variable names that are esoteric in their design or that follow a predictable standard for temporary variable naming (such as beginning with temp_var...).

The %make_var macro does just that. It creates a randomly-generated esoteric variable name and places it in a macro variable with a meaningful name. Since the variable name is just that – a name without context – it can be used as a SAS variable name, a LIBREF, a custom format name – just about anything that you need it for.

It is intended to be ephemeral – you will probably want to include it in a drop statement when using it as a variable name in a data step so that it will not be written to the output dataset.

Although it can be used directly in a data step, its true power is recognized when it is used as part of a macro definition. An illustration of that is in the next section.

The source code follows:

```
/*************************************************************************/
/* make_var.sas --                                                       */
/* This routine creates a unique macro variable name that can be used in  */
/* any routine without fear of colliding with other variables of the same */
/* name.                                                                 */
/*************************************************************************/

%macro make_var(var,len,logoption);
%global &var;
%let &var = _%substr(%sysfunc(compress(%sysfunc(uuidgen()),'-')),1,31);
%if "&len" ne "" %then %do;
  %let &var = %substr(&&&var,1,&len);
  %end;
%if %upcase("&logoption") ne "NOLOG" %then %do;
  %put NOTE: New macro variable &var created as &&&var;
  %end;
%mend;
```

%PROGRESS:

When working with very large datasets containing hundreds of millions of rows, it can take a long time for a data step to finish. You may want a macro to give a visual indication of how many records it had processed up to that moment.

That means setting up some temporary variables. But you don't want the variable names to collide with any other variables that might already be in the data step.

For example, you can't have variables named START_TIME or COUNTER because the data step calling the macro may already contain variables with those names.

Calling the %make_var macro solves that problem. In this case, you can create temporary variables for the current date/time stamp, the start time, and the counter. These variables are actually macro variables which refer to random and esoteric variables. A drop statement guarantees that the temporary variables are not written to the output dataset.

```
/*************************************************************************/
/* progress.sas --                                                       */
/*************************************************************************/

%macro progress;
  %make_var(progcntr);
  %make_var(recocntr);
  %make_var(datstrng);
  drop
    &progcntr
    &recocntr
    &datstrng
    ;
  length &datstrng $ 21;
  &progcntr + 1;
  &recocntr + 1;
  if &progcntr = 100000 then do;
    &datstrng = put(datetime(),datetime21.);
    file log;
    put &datstrng ' RecordCounter=' &recocntr comma13.;
    &progcntr = 0;
    end;
%mend;
```

CONCLUSION

The macro facility is one of the most flexible and most powerful features of Base SAS. The power of the macro facility lies in its ability to abbreviate, standardize, and automate frequently-used section of codes.

No matter how trivial or mundane the process may appear to be, look for occasions where a simple SAS macro can make your job more efficient, more readable, and easier to maintain.

CONTACT INFORMATION

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

Joe DeShon
Senior Business Analyst
Boehringer Ingelheim Animal Health, Inc.
816-210-0950
joedeshon@yahoo.com