Paper SBC-125

Quick 'n Dirty - Small, Useful Utility Macros

Harry Droogendyk, Stratia Consulting Inc., Lynden, ON

ABSTRACT

Macros are useful to define "canned" code that may be made available to other users in your organization. But, there's also simple, less sophisticated macros that are useful in your day to day coding efforts, if only to reduce keystrokes. This presentation will demonstrate a few of these macros, even ones that generate only partial statements:

%dups	generate code snippet to identify duplicate observations
%fiscal	return formatted SAS dates offset by fiscal year
%cleanup	cleanup work datasets and/or global macro variables
%single	resolve macro variables within single quotes

KEYWORDS: utility, macros, fiscal, delete, duplicate, single quote

INTRODUCTION

SAS® is a toolbox. We often find ourselves reaching for the same, familiar tools as we deal with the typical coding and data analysis tasks common to many of our occupations. While robust, parameterized macros may be overkill for much of what we do, there's enough repeatable coding activity taking place that smaller, utility macros can save us development time, typing effort and reduce errors.

This paper will demonstrate four small macros that I've found useful in reducing mundane effort in my day to day activities. However, the real intent is not merely to provide coding solutions for four coding problems, but to encourage you to think about ways you can use small SAS utility macros to save time and effort.

WHAT IS MACRO AND WHY IS IT HELPFUL?

There's often a great deal of mystery surrounding SAS macro. While the scope of this paper does not allow for a full discussion of the advantages of the SAS macro facility and how best to utilize it, it's important to remember that macro is really nothing more than text substitution. Rather than keying the same SAS code repeatedly into the Program Editor Window, it's often advantageous to define and execute SAS macros to generate repeated code, supplying parameters to create dynamic code and accomplish coding tasks with less effort and reduced errors.

The SAS macro processor runs before the SAS step compiles or runs. The timing difference allows macro invocations to generate SAS code which is substituted into the SAS code before the step is compiled. In the same way, the values of SAS macro variables are substituted into the code prior to compilation and result in dynamic code.

For a more comprehensive treatment of SAS macro, see Demystifying the SAS® Macro Facility - by Example at http://www.stratia.ca/papers/demystifying_macro.pdf.

IDENTIFYING DUPLICATE OBSERVATIONS

Much of the work we perform with SAS deals with data cleansing. Source data is often rife with quality issues, among them, duplicate observations. With the advent of SAS v9, the SORT procedure includes a DUPOUT option to route duplicate observations to an output dataset when the NODUPKEY or NODUPREC options are specified. Unique observations are output to the OUT dataset, and any duplicates are output to the DUPOUT dataset. While this can be useful in some applications, it's sometimes very helpful to keep all observations resulting from the duplicated sort criteria values for further analysis and investigation. Keeping all duplicate observations is accomplished most easily using the data step.

When the BY statement is specified, SAS automatically defines two additional "variables" and makes them available to the data step.

first.*by_variable*	- value of true when the current obs is the first value of this set for the *by_variable*
last.*by_variable*	- value of true when the current obs is the last value of this set for the *by_variable*

When both first.*by_variable* and last.*by_variable* are true, the value of the *by_variable* is unique to the current observation within those variables that precede the *by_variable* in the BY statement. An example will illustrate the concept:

```
data a;
   j = 'a'; k = 1; m = 1; output;
   j = 'a'; k = 1; m = 2; output;
   j = 'a'; k = 1; m = 3; output;
   j = 'a'; k = 2; m = 4; output;
   j = 'b'; k = 2; m = 5; output;
   j = 'b'; k = 2; m = 6; output;
run;

data b;
   set a;
     by j k;

   put j=  k=  first.j=  last.j=  first.k=  last.k= ;
run;
```

```
j=a k=1 FIRST.j=1 LAST.j=0 FIRST.k=1 LAST.k=0
j=a k=1 FIRST.j=0 LAST.j=0 FIRST.k=0 LAST.k=0
j=a k=1 FIRST.j=0 LAST.j=0 FIRST.k=0 LAST.k=1
j=a k=2 FIRST.j=0 LAST.j=1 FIRST.k=1 LAST.k=1
j=b k=2 FIRST.j=1 LAST.j=0 FIRST.k=1 LAST.k=0
j=b k=2 FIRST.j=0 LAST.j=1 FIRST.k=0 LAST.k=1
```

As the log output indicates, **first.k** and **last.k** are both true for the fourth observation, the only one in which the variable **k** is unique within the higher order BY variable, **j**. Observations with duplicate observations may be identified as those with where **first.k** and **last.k** are NOT both true in the same observation.

```
data dups;
   set a;
    by j k;

   if not(first.k and last.k);
run;
```

This very basic technique is one that is often utilized to identify duplicate observations. However, because I'm lazy and don't want to type nor think about the IF statement, I've created a little utility macro to generate the IF statement.

```
%macro dups(v);
   if not ( first.&v and last.&v )
%mend;

data dups;
   set a;
    by j k;
```

```
    %dups(k);
run;
```

Note that the statement generated by the macro **%dups** does not end in a semi-colon. That allows additional logic to be specified behind the **%dups** macro invocation if required. As a result, a semi-colon is required at some point in the line of code containing the **%dups** invocation to complete the emitted statement.

Any rocket science involved in this macro? None whatsoever, but my life is simplified many times a day as I weed through lousy data, looking for the anomalies that skew analysis.

WHAT FISCAL YEAR IS THAT DATE ?!

Many companies do not report financial results on a calendar year basis. For instance, the banks in Canada all have a fiscal year that runs from November 1st to October 31st. When reporting transactions, it's imperative that they be reported in the correct fiscal year and month. Rather than specifying the piece of code to massage the dates in many different programs, a simple macro was defined and utilized in data step, SQL and macro.

The macro essentially offsets the supplied date by two months (accounting for the start date difference from calendar year beginning Jan 1 to bank fiscal year initiating on Nov 1) and formats the new value as specified. If no format parameter is supplied, the **year.** format is defaulted.

Since function syntax must be altered slightly when invoked in macro code, the automatic macro variable **&sysprocname** is interrogated to determine which environment **%fiscal** is being executed from. If **&sysprocname** is empty, ie. has no value, the **%fiscal** macro it is being invoked within macro code, thus necessitating different INTNX syntax including the **%sysfunc** wrapper required to execute functions in straight macro.

```
%macro fiscal(help,date=,format=year.);

    %if &help = ? or %upcase(&help) = HELP %then %do;
        %put ;
        %put %nrstr(%fiscal(date=,format=););
        %put %nrstr(Returns date relative to bank fiscal year for the supplied SAS internal date.);
        %put %nrstr(Parms: &date   - SAS variable containing internal date value);
        %put %nrstr(       &format - output format, eg. month., year., yymmn6., yyq6.);
        %put %nrstr(Use:   fiscal = %fiscal(date=sas_date_variable,format=format););

    %end; %else %do;

        %if &sysprocname = %then %do;                    %*  executing from macro ;
            %sysfunc(intnx(month,&date,2),&format)
        %end; %else %do;                                 %*  executing from data or SQL step ;
            put(intnx('month',&date,2),&format)
        %end;

    %end;

%mend fiscal;
```

The **%fiscal** macro has one position parameter, **help**, and two keyword parameters, **date** and **format**. The positional format **help** is optional and only comes into play when the user executes the macro specifying a value of **?** or **HELP** to produce the macro generated documentation.

```
    %fiscal(?);

    %fiscal(date=,format=);
    Returns date relative to bank fiscal year for the supplied SAS internal date.
    Parms: &date   - SAS variable containing internal date value
           &format - output format, eg. month., year., yymmn6., yyq6.
    Use:   fiscal = %fiscal(date=sas_date_variable,format=format);
```

Three demonstrations of the macro follow:

```
data a;      * define initial data ;
   date_fld = '22dec2007'd;
run;

data b;
   set a;
   fiscal_year = %fiscal(date=date_fld);
   put 'Date is: ' date_fld date9. ', Fiscal Year is: 'fiscal_year;
run;
```

Log output: Date is: 22DEC2007, Fiscal Year is: 2008

```
proc sql;
   select date_fld format yymmddd10.
         ,%fiscal(date=date_fld,format=yyq6.)
     from a ;
quit;
```

Output: 2007-12-22 2008Q1

```
%put Fiscal YYYYMM is:%fiscal(date=%sysevalf('22dec2007'd),format=yymmn6.);
```

Log output: Fiscal YYYYMM is:200802

CLEANING UP BEFORE EXECUTION

Batch SAS processes initiate in a pristine environment unaffected by other SAS processes. In a batch context, SAS configuration file specifications and autoexec programs can be invoked to establish the required environment. However, when executing SAS interactively in display manager, left-over WORK datasets / views and/or global macro variables can have undesired consequences programs are executed successively or iteratively. Different programs may use the same WORK dataset names or identical global macro variable names and the values from one program can affect the successful execution of subsequent programs. To avoid problems between programs, it's advisable to ensure WORK datasets / views and global macros are deleted. The following macro provides that functionality while allowing for some flexibility via macro parameters.

```
%macro cleanup(help,data=Y,macro=Y);

    %if &help = ? or %upcase(&help) = HELP %then %do;
        %put;
        %put %nrstr(%cleanup(data=Y,macro=Y););
        %put %nrstr(Cleans up WORK datasets and global macro variables.);
        %put %nrstr(Parms: &data  - Y=delete work datasets, default=Y);
        %put %nrstr(       &macro - Y=delete global macro variables, default=Y);
        %put %nrstr(Use:   %cleanup(macro=N););

    %end; %else %do;

        %if &data  = %str() %then %let data  = Y; %else %let data  = %upcase(&data);
        %if &macro = %str() %then %let macro = Y; %else %let macro = %upcase(&macro);

        %if &data = Y %then %do;
            %put NOTE: %nrstr(%cleanup is deleting WORK datasets);
            proc datasets lib=work nolist nowarn nodetails
                kill;
            quit;
        %end;
```

```
        %if &macro = Y %then %do;
            %put NOTE: %nrstr(%cleanup is deleting GLOBAL macro variables);
            data _null_;
                length cmd $200;
                set sashelp.vmacro;
                    where scope = 'GLOBAL' and offset = 0 and name ne: 'SYSDB';
                cmd = '%nrstr(%symdel ' || trim(name) || ' / nowarn );';
                call execute(cmd);
            run;
        %end;

    %end;

%mend cleanup;
```

The **%cleanup** macro has one position parameter, **help**, and two keyword parameters, **data** and **macro**. Like the **%fiscal** macro already described, the positional parameter **help** is optional and only comes into play when the user executes the macro specifying a value of **?** or **HELP** to produce the macro generated documentation.

```
%cleanup(?);

%cleanup(data=Y,macro=Y);
Cleans up WORK datasets and global macro variables.
Parms: &data  - Y=delete work datasets, default=Y
       &macro - Y=delete global macro variables, default=Y
Use:   %cleanup(macro=N);
```

The actual execution of **%cleanup** is rather unspectacular. By default, if no parameters are specified on macro invocation, both the WORK library datasets / views and the global macro variables will be deleted. The only messages are those displayed as the WORK library entities are deleted and global macro variables purged. If only one of WORK library entries **or** global macro variables are to be cleaned up, the delete action must be turned off for those entities that are to be kept by specifying **N** in the applicable macro parameter.

The log displays the outcome of the **%cleanup** macro.

```
415  %cleanup;
NOTE: %cleanup is deleting WORK datasets
NOTE: Deleting WORK.A (memtype=DATA).
NOTE: Deleting WORK.B (memtype=DATA).

<snip>

NOTE: %cleanup is deleting GLOBAL macro variables

NOTE: There were 5 observations read from the data set SASHELP.VMACRO.
      WHERE (scope='GLOBAL') and (offset=0) and (name not =: 'SYSDB');

<snip>

NOTE: CALL EXECUTE generated line.
1   +  %symdel SQLOBS / nowarn ;
2   +  %symdel SQLOOPS / nowarn ;
3   +  %symdel N10 / nowarn ;
4   +  %symdel N1 / nowarn ;
5   +  %symdel N11 / nowarn ;
```

RESOLVING MACRO VARIABLES WITHIN SINGLE QUOTES

Macro variables do not resolve when specified between single quotes.

```
%let mvar = Hello;
%put '&mvar';
%put "&mvar";

'&mvar'
"Hello"
```

Within double quotes ? Not a problem. But, what is one to do when the RDBMS system being queried demands that single quotes be used around literals?

It's possible to define the macro variables with single quotes, but that limits the flexibility of those macro variables for other uses, eg. macro variable with date values used in the data set name. The **%single** macro provides a simple, concise solution to the problem by allowing the macro variable to resolve and then wrapping the resulting value in single quotes.

```
%macro single(v);

   %unquote(%str(%')&v%str(%'))

%mend single;
```

The **%single** macro accepts a single parameter, the macro variable which is to be resolved and wrapped in single quotes. The single quotes are masked by specifying them in the macro quoting function %str() . The outermost function removes the macro quoting thus emitting the single-quoted value. Macro quoting is a large topic, see Ian Whitlock's excellent paper at http://www2.sas.com/proceedings/sugi28/011-28.pdf on the topic for a full treatment of the subject matter.

```
%let date = %sysfunc(today(),yymmddd10.);
proc sql;
   connect to db2 ( database = ABC );

   select * from connection to db2 (
      select count(*) as cnt
        from schema.table
       where transaction_date = %single(&date)      );
quit;
```

In the example cited, the formatted date value must be surrounded by single quotes for DB2 to properly evaluate the WHERE condition. See the log output below for the final result:

```
42   options mprint symbolgen;
43   proc sql;
44       connect to db2 ( database = ABC );
45
46       select * from connection to db2 (
47           select count(*) as cnt
48            from schema.table
49           where transaction_date
SYMBOLGEN:  Macro variable DATE resolves to 2008-07-08
49 !                          = %single(&date)     );
SYMBOLGEN:  Macro variable V resolves to 2008-07-08
MPRINT(SINGLE):   '2008-07-08'
50       quit;
```

CONCLUSION

The four simple macros covered in this paper will not revolutionize your SAS coding experience. However, it's hoped that the examples provided will illustrated the helpfulness of simple SAS macros. If you find yourself consistently typing the same chunk of code, even if it's a code snippet, wrap it up into a macro, store it in a central location and make it available to your SAS sessions via the SASAUTOS option. For additional information on SASAUTOS, please see paper SBC-126 in the SESUG 2008 proceedings or my web site below.

REFERENCES

SAS Online Docs - **HTTP://SUPPORT.SAS.COM/91DOC/DOCMAINPAGE.JSP**

ACKNOWLEDGMENTS

Thanks to Laura House for proof-reading this paper. I'll take credit for any errors that remain.

CONTACT INFORMATION

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

Harry Droogendyk
Stratia Consulting Inc.
PO Box 145
Lynden, ON, L0R 1T0
905-296-3595
conf@stratia.ca
www.stratia.ca