Paper B2B-06

Repetitive Tasks and Dynamic Lists: Where to Find What You Need and How to Use It

Richard Addy, Rho

ABSTRACT

Often, the same operation needs to be performed for a number of items, and the number and value of those items might not be known until it is time for a program to be run. Print out the contents of an entire set of datasets within a library, generate a narrative file for each subject in a study, read in every tab from a set of Excel spreadsheets – requests for these tasks (or something similar) come up frequently.

These repetitive tasks can be handled by looping through across a list of items, but sometimes, it is not clear how to populate that list. This paper discusses methods for generating and managing such a list dynamically, using the list within a simple macro loop, as well as how to access information within SAS to find some common elements – data sets within a library, variables within a data set, files within a directory, and tabs within an Excel spreadsheet.

INTRODUCTION

A programmer will frequently need to perform the same task over similar items, and sometimes, the specifics of the task might not be known until the program is run - print all data sets in a library, for example. The task is defined in the sense that is clear what needs to be done, but sometimes details are missing – how many data sets are in that library? What are their names?

This paper gives an overview of an efficient way of handling tasks that need to be performed over and over through the use of macro variables and a macro %DO loop. First, elements that need to be updated with every repetition of a task are identified and replaced with macro variables – values for these variables can be assigned directly, or populated dynamically from a data set within the program. Then, a macro %DO loop is built to take advantage of these macro variables.

Some of the functions and data sets described in this paper require SAS® version 9 or higher, but the general strategies described are broadly applicable to previous SAS® versions as well.

DEFINING THE TASK

Often, a programmer needs to perform the same task over and over again. Sometimes, very little needs to be changed between one iteration and the next – a data set name needs to be updated, or different variables need to be used. As an example, suppose the following task has been set: print all the data sets in a directory; this directory has been assigned a libname of 'FINAL'. There are three data sets in the directory: a demographic data set (DEMO.sas7bdat), an adverse events data set (AEXP.sas7bdat), and one for concomitant medications (CMED.sas7bdat). The task could be accomplished with the following code (with somewhat rustic results):

```
PROC PRINT DATA = FINAL.AEXP; RUN;
PROC PRINT DATA = FINAL.CMED; RUN;
PROC PRINT DATA = FINAL.DEMO; RUN;
```

That's not a lot of code; it's not hard to just copy and paste one line after the other. But what if the directory had more than three data sets? What if there were hundreds of data sets? What if another programmer wanted to base their program on this code, but needed to look at data sets in another folder, and the data sets may or may not have the same names? These sorts of obstacles can be handled efficiently in two ways: using macros to handle tasks and sections of code that get repeated, and writing code that dynamically determines the specific items needed at runtime.

A good way to tell where there is a good location for a macro variable is to examine the code and see what would need to be updated once the code has been copied and pasted. In this example, the only thing that differs in each PROC is the name of the data set, so that's a good place to start. Another thing to consider is whether the information needed to populate those macro variables is available at the time the code is programmed, or if that information won't be needed until the program is actually run. In the former case, the macro variables can be coded as part of the program; in the latter, often SAS can provide the information needed dynamically; it's just a matter of knowing where to look.

Once likely candidates for macro variables have been identified, a loop can be constructed to efficiently repeat a series of tasks. SAS® provides this functionality through the use of a macro %DO loop – it's very similar to a DO

loop within a DATA step, but allows for more flexibility – a macro %DO loop can include multiple PROCs, and isn't limited to just DATA steps.

MACRO VARIABLES

Macro variables are proxies – they are a way for a programmer to indicate that a value will be resolved at run time. They can be used in DATA steps, in PROCs, in titles, path statements – almost anywhere. In a basic sense, macro variables are treated as text – whenever SAS® sees a reference to these macro variables, it just replaces that reference with whatever value has been defined for that macro variable. Macro variables, once assigned, stay assigned until their values are updated, or their scope ends. The scope of a macro variable, by default, depends on where it is assigned. If a macro variable is defined in open code, it stays assigned for the rest of the SAS® session – it's a global macro variable. Macro variables defined as part of another macro have a scope of that macro – the variables are available while the macro is running, but not once it completes (and if the macro is run more than once, variables assigned in one instance are not available in other instances).

There are a few ways to define macro variables – the %LET statement can define a macro variable directly, but can only assign one value at a time, and generally, the values being assigned need to be known ahead of time. For example, the following code creates 4 macro variables – three for data set names, and one for the count of data set names:

```
%let ds1 = AEXP;
%let ds2 = CMED;
%let ds3 = DEMO;
%let dsnum = 3;
```

Once these statements are included in a program, these macro variables can be used in place of the data set names and count. To let SAS® know that it's found a macro variable (and not some piece of regular code), a macro variable is preceded by at least one ampersand. If a macro variable is referenced completely, only a single ampersand is required. Since macro variables are text fragments, a programmer may need to tell SAS® where a macro name ends and regular code picks up. Delimiters include a space, an equal sign, an ampersand, a percent sign, a semicolon, and a period. Often, there will be a space or semicolon at the end of macro variable, so no explicit delimiter is needed.

For example, once the macro variables have been assigned, the following lines of code both do the same thing:

```
PROC PRINT DATA = FINAL.AEXP; RUN;
PROC PRINT DATA = FINAL.&ds1; RUN;
```

When SAS® gets to the '&ds1.', it looks up the macro variable ds1 and replaces it with whatever value it finds – in this case, AEXP.

Multiple ampersands are used if a macro variable needs to include another macro variable as part of its name. For example, if a program loops through a list of macro variables. In the example above, the first time through the loop, we would want to end up with AEXP, then CMED, then DEMO (or in other words, &ds1, followed by &ds2, followed by &ds3). Multiple ampersands tell SAS® to make another pass and resolve the macro variable name after it has resolved other references.

So, a reference to &&ds&j would tell SAS® to hold off on trying to resolve the first section of the macro name until the second section had been resolved. If &j had a value of one, SAS® would resolve &&ds&j to &ds1, and then to AEXP.

Macro variables can also be defined using values in data sets – this can be handled in a PROC SQL procedure:

```
PROC SQL NOPRINT;
    SELECT DISTINCT (variable of interest)
    INTO :(macro variable name) - :(macro variable name)(number of
          macro variables to create)
    FROM (data set);
QUIT;
```

Other statements can be added to the PROC SQL code to modify or subset the dataset of interest – a WHERE clause could be used to restrict records, for example, or an ORDER BY statement to generate values in a sorted manner. Here's the code for getting values of the variable ID from the data set FINAL.DEMO into macro variables, using an arbitrary upper limit on number of subjects expected:

```
PROC SQL NOPRINT;
    SELECT DISTINCT ID
    INTO :ID1 - :ID999
    FROM FINAL.DEMO
    ORDER BY ID;
QUIT;
```

If there are fewer than 999 unique values of ID in FINAL.DEMO, macro variables are only created for the number of values found. If there are more than 999 unique values, only 999 macro variables are created. Neither case is pleasant – in the former, the programmer doesn't know how many variables were created, and in the latter, some values will be omitted. A way around this is to first find out how many unique values there are and then created just the number of variables needed:

```
PROC SQL NOPRINT;
    SELECT COUNT(DISTINCT ID)
    INTO :nID
    FROM FINAL.DEMO;
QUIT;

%let nID = &nID;

PROC SQL NOPRINT;
    SELECT DISTINCT ID
    INTO :ID1 - :ID&nID
    FROM FINAL.DEMO
    ORDER BY ID;
QUIT;
```

The %let statement between the two PROC SQL steps removes leading blanks from the value of macro variable &nID.

Macro variables can also be generated as part of a DATA step, using CALL SYMPUTX (if using SAS® version 9 or higher) or CALL SYMPUT. Both functions do the same thing, but CALL SYMPUTX has some nice features that makes its use preferred – it strips trailing and leading spaces, for example, and handles numeric values a little more smoothly. The syntax for CALL SYMPUTX follows:

```
DATA _NULL_;
    SET (data set of interest);
    CALL SYMPUTX ('macro variable name', variable of interest);
RUN;
```

CALL SYMPUTX only resolves a macro value reference when the DATA step is compiled, so in general, a macro variable cannot be used in the same DATA step where SYMPUTX defines it. Also, if the same macro variable name is used in multiple rows, the macro variable will be assigned the value based on the last record used.

Here's the DATA step equivalent of the previous example -

```
PROC SORT NODUPKEY DATA = FINAL.DEMO OUT = DEMOLIST (KEEP = ID);
    BY ID;
RUN;

DATA _NULL_;
    SET DEMOLIST END=EOT;
    RECNUM = STRIP(PUT(_n_, BEST.));
    CALL SYMPUTX( 'ID' || RECNUM, ID);
    IF EOT THEN DO;
           CALL SYMPUTX( 'nID', RECNUM);
    END;
RUN;
```

The initial NODUPKEY sort is used to make sure the values used in the macro variables are unique. The RECNUM variable keeps count of the number of records encountered, both to use as a count of total number of records, and to

make the individual ID macro variable names unique. At the end of the DATA step, the value of RECNUM is assigned to the macro variable &nID, so the number of ID values can be used later. The EOT section isn't strictly necessary – since &nID would have been assigned the last value of RECNUM anyway, the result would have been the same without it, but it makes the intent of the code a little clearer.

No matter how the macro variables are assigned - %LET, PROC SQL, or SYMPUTX in a DATA step– the scope of the macro variables is by default based on where the macro variables are created. If done inside a macro; their scope is that macro. If done in open code, their scope is the SAS® session.

Once a macro variable is assigned, its current value can be written to the log with a %PUT statement. Executing a %PUT _USER_; statement will write all currently assigned macro variables and their values to the log. You can also use a few options to make things a little clearer – if OPTIONS MPRINT (for macro print) is set, the log will include lines generated when a macro is run and OPTIONS SYMBOLGEN will show the values of the macro variables as they are resolved. This can make the log a little noisy – you can switch this behavior off with OPTIONS NOMPRINT and OPTIONS NOSYMBOLGEN. Macro variables can also be included in titles, footnotes, filenames, and other quoted text – SAS® will resolve them as long as they are in double quotes.

THE MACRO %DO LOOP

Once a series of macro variables have been created and the number of these variables is known, they can be used in a macro %DO loop. A %DO loops is very similar to a DO loop within a DATA step, but there are some important differences:

- A %DO loop must be used inside a macro; it cannot be used in open code (The macro that holds the %DO loop can be very simple and need not contain anything more than the loop, although it can, if needed)
- A %DO loop can include multiple DATA steps and PROCs

Macros are defined first, and then called. The syntax of a simple macro containing a %DO loop follows:

```
%MACRO LOOPER;
    %DO (iterator) = 1 %TO (number of items);
           (DATA steps, PROCs, etc.)
    %END;
%MEND LOOPER;
%LOOPER;
```

The definition of the macro begins with a %MACRO statement and ends with %MEND - it can be called immediately or later in the program. The code within the %DO loop is executed once per iteration, and the iterator is itself a macro variable that can be used with the loop.

Going back to the example of printing the contents of datasets within a library – there are 4 macro variables (&ds1, &ds2, &ds3, and &dsnum). The %DO loop for this task would look like:

```
%MACRO LOOPER;
    %DO j = 1 %TO &dsnum;
           PROC CONTENTS DATA = FINAL.&&ds&j; RUN;
    %END;
%MEND LOOPER;
%LOOPER;
```

&dsnum has a value of 3, so the %DO loop will execute three times. The first time through the loop, the macro variable &j has a value of 1. &&ds&j resolves first to &ds1 and then to AEXP. The second time through the loop, the macro variable &j has a value of 2. &&ds&j resolves first to &ds2 and then to CMED. On the final trip through the loop, the macro variable &j has a value of 3. &&ds&j resolves first to &ds1 and then to DEMO.

SOURCES OF INFORMATION

So, to handle as repetitive task, if a programmer has a data set with the values needed, they can get those values into macro variables and use a macro %DO loop to perform all the needed tasks as many times as necessary. But where to go to find the required information in the first place?

Sometimes, the data used within a program will be sufficient; if a program is meant to generate one report per site in a project, the site values can be pulled directly from the data sets used in programming. But frequently, a program needs information about data sets and variables, not just information from data sets and variables.

Information about items SAS ® knows about can be found in SASHELP views and DICTIONARY tables – this includes information about libraries, data sets, variables, external files, and pretty much anything that SAS® can access. Both SASHELP and DICTIONARY contain the same information, but they are accessed differently. SASHELP views can be used as an input to a DATA step; PROC SQL must be used to read data from DICTIONARY tables.

There are two places to get started: SASHELP.VMEMBER/DICTIONARY.MEMBERS is a useful source of information about data sets and other files. SASHELP.VCOLUMN/DICTIONARY.COLUMNS can be read to get information about variables in data sets. The SASHELP views can be viewed directly from the Explorer pane in SAS®; and either the SASHELP views or the DICTIONARY tables can be used to generate other datasets, if a programmer needs some specifics about how things are set up.

In SASHELP.VMEMBER/DICTIONARY.MEMBERS, the libname of data sets is stored in a variable called LIBNAME; the data set names are in MEMNAME. Values in these data sets are in upper case. SASHELP.VCOLUMN/DICTIONARY.COLUMNS have a similar structure – In addition to the LIBNAME and MEMNAME variables, variable names are in NAME. These views and tables also contain a lot of other information that could prove useful, depending on what needs to be accomplished – locations, labels, types, etc.

Information about data sets and their variables, formats, catalogs can be accessed directly from a SASHELP view or DICTIONARY table. For other file types, SASHELP.VEXTFL/DICTIONARY.EXTFILES can be used as a starting point, and the functions DREAD and DNUM can be used to obtain more information (see the example "get a list of all .TXT files in a directory" below).

Once the source of information is located, it can be subset to a data set and that can be used to populate a list of macro variables for use in a macro %DO loop. Here are some examples for using SASHELP views and the DICTIONARY tables:

A LIST OF ALL DATA SETS CURRENTLY IN THE WORK LIBRARY

```
%* Get countof # of work data sets;
PROC SQL NOPRINT;
    SELECT COUNT(*)
    INTO :nWKDS
    FROM DICTIONARY.MEMBERS
    WHERE LIBNAME = "WORK" AND MEMTYPE = "DATA";
QUIT;

%let nWKDS = &nWKDS;

%* Create macro variables for each data set name;
PROC SQL NOPRINT;
    SELECT MEMNAME
    INTO :WKDS1 - :WKDS&nWKDS
    FROM DICTIONARY.MEMBERS
    WHERE LIBNAME = "WORK" AND MEMTYPE = "DATA";
QUIT;
```

RENAMING COMMON VARIABLES PRIOR TO A MERGE

Two data sets in two libraries need to be merged (DRAFT.JOBS and NEW.POSITIONS), but may have some variables in common other than the BY variables (STATE, JOBNUM) used in the merge. In order to prevent values in one data set from overwriting those in the other (and to avoid instances where variables might be of different types in each data set), rename all the variables in common prior to performing the merge.

```
%* Get names of variables in each dataset;
PROC SORT DATA = SASHELP.VCOLUMN OUT = JOBSLIST (KEEP = NAME);
    BY NAME;
    WHERE LIBNAME = "DRAFT" AND MEMNAME = "JOBS" AND
    NAME NOT IN ("STATE" "JOBNUM");
RUN;
```

```
PROC SORT DATA = SASHELP.VCOLUMN OUT = PSTNSLIST (KEEP = NAME);
    BY NAME;
    WHERE LIBNAME = "NEW" AND MEMNAME = "POSITIONS" AND
    NAME NOT IN ("STATE" "JOBNUM");
RUN;


%* FIND VARIABLES IN COMMON;
DATA COMMVAR;
    MERGE JOBSLIST (IN=INJOBS) PSTNSLIST (IN=INPSTN);
    BY NAME;
    IF INJOBS AND INPSTN;
RUN;


%* GENERATE MACRO VARIABLES;
DATA _NULL_;
    SET COMMVAR END=EOT;
    RECNUM = STRIP(PUT(_n_, BEST.));
    CALL SYMPUTX( 'CVAR' || RECNUM, NAME);
    IF EOT THEN DO;
           CALL SYMPUTX( 'nCVAR', RECNUM);
    END;
RUN;


%* RENAME VARIABLES IN COMMON;
%MACRO RENAMER;
    DATA JOBS;
           SET DRAFT.JOBS;
           %DO j = 1 %TO &nCVAR;
                  RENAME &&CVAR&J = &&CVAR&J.._J;
           %END;
    RUN;


    DATA POSITIONS;
           SET NEW.POSITIONS;
           %DO j = 1 %TO &nCVAR;
                  RENAME &&CVAR&J = &&CVAR&J.._P;
           %END;
    RUN;
%MEND RENAMER;
%RENAMER;
```

IMPORT ALL TABS FROM AN EXCEL SPREADSHEET

Some care needs to be taken with this approach – it relies on SAS®' default behavior, and is not well-suited to spreadsheets that are not well structured. Also, the entries in SASHELP.VCOLUMN/DICTIONARY.COLUMNS end with a "$' and may contain quotes or other special characters – these need to be handled if the tab name is meant to also be used as the name of the output data set.

```
%* Assign a libname to the file;
LIBNAME XLIN "C:\TEMP\SUMMARY_20130715.XLSX" MIXED = YES;


%* GET TAB NAMES;
DATA XLINFO (KEEP = MEMNAME MEMNAME2);
    SET SASHELP.VCOLUMN;
    WHERE LIBNAME = "XLIN";
    %* MEMNAME2: TAB NAME SUITABLE AS DATA SET NAME;
    MEMNAME2 = TRANWRD(MEMNAME, '#', '_');
    MEMNAME2 = COMPRESS(MEMNAME2, "'$");
RUN;
```

```
%* GENERATE MACRO VARIABLES;
DATA _NULL_;
    SET XLINFO END=EOT;
    RECNUM = STRIP(PUT(_n_, BEST.));
    CALL SYMPUTX( 'TABIN' || RECNUM, MEMNAME);
    CALL SYMPUTX( 'TABNM' || RECNUM, MEMNAME2);
    IF EOT THEN DO;
           CALL SYMPUTX( 'nTAB', RECNUM);
    END;
RUN;


%* IMPORT TABS;
%MACRO IMPORTER;
    %DO J = 1 %TO &nTAB;
           DATA &&TABNM&J;
                  SET XLIN."&&TABIN&J"N;
           RUN;
    %END;
%MEND IMPORTER;
%IMPORTER;


%* RELEASE SPREADSHEET;
LIBNAME XLIN CLEAR;
```

GENERATING PATIENT PROFILES AS SEPARATE FILES

```
%* Get ID values;
PROC SQL NOPRINT;
    SELECT COUNT(DISTINCT ID)
    INTO :nID
    FROM FINAL.DEMO;
QUIT;


%let nID = &nID;


PROC SQL NOPRINT;
    SELECT DISTINCT ID
    INTO :ID1 - :ID&nID
    FROM FINAL.DEMO
    ORDER BY ID;
QUIT;


%MACRO REPORTER;
    %DO J = 1 TO &nID;
           %* START A NEW FILE FOR EACH SUBJECT;
           ODS PDF FILE="C:\TEMP\PROFILE_&&ID&J…PDF";
                  PROC PRINT DATA=REPORT1;
                         WHERE ID = "&&ID&J";
                  RUN;
                  PROC PRINT DATA=REPORT2;
                         WHERE ID = "&&ID&J";
                  RUN;

           ODS PDF CLOSE;

    %END;
%MEND REPORTER;
%REPORTER;
```

GET A LIST OF ALL .TXT FILES IN A DIRECTORY

The following example assumes a FILENAME statement has been executed which points to a directory named TEXTLOC, and there are one or more text files (.txt) expected to be in that location at the time the program is run.

```
%* Get path location

%* Get directory path;

DATA _NULL_;
       SET SASHELP.VEXTFL (WHERE = (FILEREF="TEXTLOC"));
       CALL SYMPUTX('TextPath', xpath);
RUN;


%* Open the directory and loop through its members;
%* If a .txt file is found, create a macro variable containing the filename;
DATA _NULL_;
       %* Find the directory;
       RC=FILENAME("TextDir", "&TextPath");
       %* Open the directory;
       Textdir = DOPEN("TextDir");
       %* Find out how many total files are in the directory;
       ItemTot = DNUM(TextDir);

       %* Start a counter of text files found;
       TXTNUM = 0;

       %* Loop through files, and see if any end in .txt;
       DO j = 1 TO ItemTot;
              FILENAME = UPCASE(DREAD(TextDir, j));
              IF SUBSTR(REVERSE(STRIP(FILENAME)),1,4) = 'TXT.' THEN DO;
                     %* FOUND A TEXT FILE;
                     TXTNUM =  TXTNUM + 1;
                     TXTNUMC = STRIP(PUT(TXTNUM, BEST.));
                     CALL SYMPUTX( 'TXTFILE' || TXTNUMC, FILENAME);
                     CALL SYMPUTX( 'TEXTTOT', TEXTNUMC);
              END;
       END;

       %* CLOSE DIRECTORY;
       RC = DCLOSE(TextDir);
RUN;
%PUT TOTAL NUMBER OF TEXT FILES FOUND: &TEXTTOT;
```

CONCLUSION

This paper described a broad strategy for handling repetitive tasks that often face SAS® programmers:

1. Identify what gets repeated
2. Find a source of information for the repeated elements
3. Generate a data set with that information
4. Construct a series of macro variables from that data set
5. Use a macro %DO loop to repeat the needed tasks, using those macro variables

This approach is applicable to a wide variety of tasks and has the advantage that the data set used to generate the macro variables can be populated when the program is run – specific values needed don't need to be known at the time the program is written.

RECOMMENDED READING

- Base SAS® Procedures Guide
- Carpenter, Arthur, “The Path, The Whole Path, And Nothing But the Path, So Help Me Windows“. P Proceedings of the SAS® Global Forum 2008 Conference. Cary NC: SAS Institute Inc. Available at http://www2.sas.com/proceedings/forum2008/023-2008.pdf
- Cohen, John. 2012. “A Tutorial on the SAS Macro Language.” Proceedings of the SAS® Global Forum 2012 Conference. Cary NC: SAS Institute Inc. Available at http://support.sas.com/resources/papers/proceedings12/249-2012.pdf
- Davis, Michael. 2001. “You Could Look It Up: An Introduction to SASHELP Dictionary Views.” Proceedings of the Twenty-Sixth Annual; SAS® Users Group International Conference. Cary NC: SAS Institute Inc. Available at http://www2.sas.com/proceedings/sugi26/p017-26.pdf

CONTACT INFORMATION

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

Name: Richard Addy
Enterprise: Rho, Inc
Address: 6330 Quadrangle Drive
City, State ZIP: Chapel Hill, NC, 27517
Work Phone: (919) 595-6579
Fax: (919) 408-0999
E-mail: Richard_addy@rhoworld.com
Web: http://www.rhoworld.com/