Writing cleaner and more powerful SAS code using macros

Patrick Breheny

Outline

1. Quick Review of Macro Basics
2. Working with Macro Strings
3. Getting More Out of Macros:
   a) Program Control
   b) Interfacing with Data

The Compilation of SAS Programs

- SAS code is compiled and executed alternately in steps:
  - For example, a data step will be compiled and executed, then a procedure step will be compiled and executed
- IMPORTANT: Macros are resolved PRIOR to the compilation and execution of the SAS code

SAS Compilation (cont’d)

- Code without Macros:

<table>
<thead>
<tr>
<th>SAS Code</th>
<th>Compilation</th>
<th>Program</th>
<th>Execution</th>
<th>Results</th>
</tr>
</thead>
</table>

- Code with Macros:

<table>
<thead>
<tr>
<th>SAS Code</th>
<th>Macro Processing</th>
<th>SAS Code</th>
<th>Compilation</th>
<th>Execution</th>
<th>Results</th>
</tr>
</thead>
</table>

Macro Code

- The two basic elements of macro code are macro variables and macros. In SAS code:
  - &name refers to a macro variable
  - %name refers to a macro
- Macro code consists of these two elements and their relationship to each other
- Text without %’s or &’s (called constant text) is unaffected by macro processing

Why Use Macros

- Macros automatically generate SAS code
- Macros allow you to make more dynamic, complex, and generalizable SAS programs
- Macros can greatly reduce the effort required to read and write SAS Code
Working with Macro Strings

The Implicit Handling of Strings

• Because macros and macro variables can only be assigned strings of text, string functions on macro variables are handled implicitly:
  - Assignment: No quotes are necessary around the value of a macro variable (\%let mac_var = Hello;)
  - Concatenation: survey_&state concatenates &state with "survey_"

• Most of the time, this is very convenient, but any time you avoid giving explicit instructions, computers may do something other than what you want!

Concatenation

• The expression survey_&state is unambiguous, but what about &state_survey?

  \%put survey_I&ate;
  survey_IA

  \%put &state_survey;
  WARNING: Apparent symbolic reference STATE_SURVEY not resolved.
  \%put &state_survey;

• A period is the signal in SAS to end a macro variable name:

  \%put &state._survey;
  IA_survey

Double vs. Single Quotes

• Double quotes and single quotes affect macro variables differently:

  proc import datafile="H:\Macro Workshop\survey_I&ate.xls"
  out=survey_IA replace;
  run;
  ERROR: Unable to import, file H:\Macro Workshop\survey_I&ate.xls does not exist.

• Note that macro variables inside single quotes are not resolved

SAS Characters with Special Meaning

• Suppose we wish to assign a macro variable a string with semicolons, commas, or quotes

• The macro function %str can be used, for example, to pass an entire statement into a macro:

  \%macro reg(predictors, options);
  proc reg data=dataset;
    model outcome = \%str(predictors);
    \%options
  run;
  \%end reg;
  \%reg(age sex, \%str(age, age - sex / compstat));
Evaluating Numeric Strings

- Remember, macro variables are strings, not numeric quantities:

```
DATA SUM;
  SUM = 1+1;
  PUT SUM;
END;
```

- The function `%eval` can be used to obtain the (integer) numeric value of an expression containing macro variables:

```
DATA TOTAL;
  TOTAL = %eval(SUM);
  PUT TOTAL;
END;
```

- Note: Floating point evaluations can be performed with `%sysevalf`.

Getting More Out of Macros

Program Control

- The most powerful feature of macros is their ability to use conditional and iterative statements.
- Data steps provide these same statements, but their effect is limited to a single data step.
- Program control through macros can extend across multiple data steps and procedures.

Conditional Statements

- Conditional statements in macros work just like those in data steps.

```
%if (STATE eq IA) %then
   PUT Iowa;
%else
   PUT Not Iowa;
%end;
```

Iterative Statements

- Iterative macro statements will also be familiar to anyone who has used the data step versions.

```
%do i = 1 %to 10;
   PUT %eval(i**2);
%end;
```

- Note: `%do...%while` and `%do...%until` statements are also available.
Macro Program Control Statements

- Macro program control statements are not valid in open code
- They must be contained within macros

Macro "Arrays" (cont'd)

- Instead, we must force the macro processor to make multiple passes over our code:

```
%let state1 = AL;
%let state2 = AZ;
...
%let state50 = WY;
```

- If we were in the i\textsuperscript{th} iteration of a loop, how would we access the i\textsuperscript{th} member of the list?

```
%put state%i;
IA2
```

Example

- Suppose we wish to create a report by state of county rankings for a number of categories:

```
%macro report:
 %do i = 1 %to 50;
   %do j = 1 %to 25;
     %county_sort(&vars);
     state=state%i,
     order=descending);
   %end;
 %end;
 %end report;
```

Nesting Macro Calls

- As we just saw, it is often a good idea to nest macro calls:

```
%macro a;
 SAS code.
%end a;
```

- It is not a good idea to nest macro definitions:

```
%macro a:
 SAS code.
%macro b;
 SAS code.
%end b;
%end a;
```

Nesting Macro Calls (cont'd)

- When nesting macro calls, be careful to avoid variable collisions:

```
%macro print_sum;
 %do i = 1 %to 10;
   %put sum(i);
 %end;
 %end;
```

```
%macro sum(i);
 %let current_sum=0;
 %do i = 1 %to 10
   %eval(sum(i);
   %let current_sum=+current_sum + i;
 %end;
 %eval(current_sum)
 %end;
```

- Scoping issues can be avoided by using %local to define macro variables
Suppose we submitted the following code to SAS:

```sas
data nevclata;
set survey_ya;
let AgeSq = Age**2;
run;
```

What would happen?

### How symput Works

- Calling the symput routine pauses execution of the data step and writes a data value to a macro variable
- Syntax:
  ```sas
  CALL SYMPUT('macro-variable', data-variable);
  ```
- Both arguments to symput can be expressions
- IMPORTANT: You CANNOT access a macro variable within the same data step it is created

### symputx: A Better symput

- CALL SYMPUTX is a variant of SYMPUT introduced in SAS 9 that has similar syntax, but handles the input of numeric values better
- The following example illustrates the difference between the two commands:

```sas
data _null_;
call symput('symput',5);
call symputx('symput',5);
run;
```

### Example

- Suppose we want to compare two groups, but the preferred method depends on sample size:

```sas
%macro compare(data, class, cutoff=20);
data _null_; set data; call symputx('noba',noba);
run;
%if (noba < cutoff) then do;
  proc means data=data; class class;
run;
%end;
%else do;
  proc means data=data; class class;
run;
%end;
%end;
```

- symget is much more straightforward:
  ```sas
  data-variable = symget('macro-variable')
  ```
Putting it all Together

- As a final example, suppose we want to create a list of indicator variables for the values of a categorical variable in a data set.
- Note that if we don't know the values in advance, we have to approach the problem in two steps:
  1. Determine the new variables we are to create.
  2. Create a data set in which we assign values to the new variables.

Putting it all Together (cont'd)

(maple make_ind(den,cat));
proc sort data=den out=sorted;
  by cat;
run;
data null;
  set sorted end=eof;
  by cat;
  if first.cat then
do:
  total;
  call symput("cat.ind":compress(tot),compress(4cat));
  end:
  if eof then call symput("tot",tot);
run;

Putting it all Together (cont’d)

(maple make_ind(den,cat));
proc sort data=den out=sorted;
  by cat;
run;
data null;
  set sorted end= eof;
  by cat;
  if first.cat then
do:
  total;
  call symput("cat.ind":compress(tot),compress(4cat));
  end:
  if eof then call symput("tot",tot);
run;

Putting it all Together (cont’d)

(data &den_ind;
  set &den;
  &do i=1 to level(4cat);
    &if (compress(4cat) eq "&4cat.ind":i) then &4cat.ind":i = 1;
    else &4cat.ind":i = 0;
  &end;
run;
&end make_ind;

References

- The SAS Macro Language Reference: