INTRODUCTION TO SAS MACRO LANGUAGE 1

MACRO VARIABLES 1
MACRO FUNCTIONS 5
SYMPUT AND SYMGET FUNCTION TO PASS INFORMATION TO AND FROM A DATA STEP 6
CREATING MACRO VARIABLES USING PROC SQL 8
CREATING A LIST OF FILE NAMES FOR A DATA STEP USING A MACRO PROGRAM 9
A MACRO PROGRAM FOR REPEATING A PROCEDURE MULTIPLE TIMES 11
REFERENCE 16
PRIMARY SIDEBAR 16
FOOTER 16

Introduction to SAS Macro Language

Statistical Computing Seminar: Introduction to SAS Macro Language

This seminar is designed to introduce the basics of SAS macro language. Here is the program on which the seminar is based. In this seminar we will cover the following topics:

- Macro variables
- Macro functions
- symput and symget function to pass information to and from a data step
- Creating a macro variable using proc sql
- Creating a list of file names for a data step using a macro program
- A macro program for repeating a procedure multiple times

The SAS macro language is a very versatile and useful tool. It is often used to reduce the amount of regular SAS code and it facilitates passing information from one procedure to another procedure. Furthermore, we can use it to write SAS programs that are “dynamic” and flexible. Generally, we can consider macro language to be composed of macro variables and macro programs. In this seminar we will demonstrate how to create macro variables and how to write basic macro programs.

Macro Variables

A macro variable in SAS is a string variable that allows you to dynamically modify the text in a SAS program through symbolic substitution. The following example demonstrates how to create and use a macro variable. First we set up some system options to have a more concise output style.

```
options nodate nonumber nocenter formdlim="-";
data hsb2;
  input  id female race ses prog
         read write math science socst;
datalines;
 70 0 4 1 1 57 52 41 47 57
121 1 4 2 3 68 59 53 63 61
 86 0 4 3 1 44 33 54 58 31
141 0 4 3 3 63 44 47 53 56
172 0 4 2 2 47 52 57 53 61
113 1 4 2 2 44 52 51 63 61
 50 0 3 2 1 50 59 42 53 61
 11 0 1 2 2 34 46 45 39 36
 84 0 4 2 1 63 57 54 51 63
 48 1 3 2 2 57 55 52 50 51
 75 1 4 2 3 60 46 51 53 61
 60 1 4 2 2 57 65 51 63 61
 95 0 4 3 2 73 60 71 61 71
104 0 4 3 2 54 63 57 55 46
 38 0 3 1 2 45 57 50 31 56
115 0 4 1 1 42 49 43 50 56
 76 0 4 3 2 47 52 51 50 56
195 0 4 2 1 57 57 60 56 52
;
run;
```

Suppose that we want to look at the means of some variables and then do a regression analysis on the same variables.

```
proc means data = hsb2;
  var write math female socst;
run;
proc reg data = hsb2;
  model read = write math female socst;
run;
quit;
```

We can simplify the program by creating a macro variable containing all the names of the independent variables. A macro variable can be created by using the **%let** statement. All the key words in statements that are related to macro variables or macro programs are preceded by percent sign %; and when we reference a macro variable it is preceded by an ampersand sign &. When we submit our program, SAS will process the macro variables first, substituting them with the text string they were defined to be and then process the program as a standard SAS program.

```
%let indvars = write math female socst;
proc means data = hsb2;
  var &indvars;
run;

proc reg data = hsb2;
```

```
  model read = &indvars;
run;
quit;
```

We can display macro variable value as text in the log window by using **%put** statement.

```
%put my first macro variable indvars is &indvars;
```

In the log window, you will see the following:

```
90   %put my first macro variable indvars is &indvars;
my first macro variable indvars is write math female socst
```

SAS has many **system-defined** macro variables. These macro variables are created automatically when SAS is started. Therefore, they are sometimes called automatic macro variables. We can use the **%put** statement again to display the values of these system-defined macro variables.

```
%put _automatic_;
```

Below is a partial output from the log window. The first column indicates the type of macro variable, the second indicates the name of the macro variable and the third contains the value of the macro variable. For example, **SYSDSN** (system data source name) is in the WORK directory and the last data set created was **hsb2**.

```
92   %put _automatic_;
AUTOMATIC AFDSID 0
AUTOMATIC AFDSNAME
AUTOMATIC AFLIB
AUTOMATIC AFSTR1
AUTOMATIC AFSTR2
AUTOMATIC FSPBDV
AUTOMATIC SYSBUFFR
AUTOMATIC SYSCC 0
AUTOMATIC SYSCHARWIDTH 1
AUTOMATIC SYSCMD
AUTOMATIC SYSDATE 17JUN03
AUTOMATIC SYSDATE9 17JUN2003
AUTOMATIC SYSDAY Tuesday
AUTOMATIC SYSDEVIC
AUTOMATIC SYSDMG 0
AUTOMATIC SYSDSN WORK    HSB2
```

These macro variables can be used in the same way as ordinary macro variables. For example, in the following example, we use two of the system-defined macro variables in the **title** statement.

```
title "today's date is &SYSDATE9 and today is &SYSDAY";
proc means data = hsb2;
  var &indvarS;
run;
today's date is 17JUN2003 and today is Tuesday.
The MEANS Procedure
```

```
Variable     N            Mean         Std Dev         Minimum
Maximum
-----------------------------------------------------------------------------
-
write       18      53.2222222       7.7273811      33.0000000
65.0000000
math        18      51.6666667       7.1373088      41.0000000
71.0000000
female      18       0.2777778       0.4608886               0
1.0000000
socst       18      55.3888889       9.6536423      31.0000000
71.0000000
-----------------------------------------------------------------------------
-
```

Notice that in the **title** statement we used double quotation marks around the title. Normally, we can use either single quotes or double quotes. When macro variables are embedded in the **title** statement, only double quotes will work. The following example shows some of the problems that might occur when using single quotes with macro variables.

```
title 'The date is &SYSDATE9 and today is &SYSDAY';
proc means data = hsb2;
  var &indvarS;
run;
The date is &SYSDATE9 and today is &SYSDAY.
The MEANS Procedure
Variable     N            Mean         Std Dev         Minimum
Maximum
-----------------------------------------------------------------------------
-
write       18      53.2222222       7.7273811      33.0000000
65.0000000
math        18      51.6666667       7.1373088      41.0000000
71.0000000
female      18       0.2777778       0.4608886               0
1.0000000
socst       18      55.3888889       9.6536423      31.0000000
71.0000000
-----------------------------------------------------------------------------
-
```

We can also display macro variables defined by a user. The macro variable **indvar,** which was defined earlier, is an example of a user defined macro variable. Since **indvar** was defined outside a macro program it is by default a global macro variable. A global macro variable can be use in any SAS procedure or data step whereas a local macro variable can only be used inside the macro program in which it was defined.

```
%put _user_;
127  %put _user_;
GLOBAL INDVARS write math female socst
```

Summary:

In this section, we have mentioned the following.

- defining a macro variable by using **%let** statement;
- displaying macro variable values as text in the SAS log by using **%put** statement;
- System-defined automatic macro variables
    - **%put _automatic_**;
- User-defined macro variables
    - **%put _user_**;
- Substituting the value of a macro variable in a program;
    - use of &;
    - double quotes vs. single quotes;

Macro functions

There are many functions that are related to macro variables. They include string functions, evaluation functions and others. In the this section we will show some examples of these different types of functions..

Some of the most commonly used string functions include **%upcase**, **%substr** and **%scan**. The function **%scan** takes a string and an integer *i* as arguments and returns the *i*th word in the string. The **%substr** function will pick out a subcomponent of a string variable; this function takes three arguments where the first argument is the string variable (a macro variable), the second is the start position of the substring and the third argument is the length of the substring. The **%upcase** function creates a new variable which contains the upper case version of a string variable.

```
%put &indvars;
938  %put &indvars;
write math female socst
%let newind = %upcase(&indvars);
%put &newind;
940  %let newind = %upcase(&indvars);
941  %put &newind;
WRITE MATH FEMALE SOCST
%let word2 = %scan(&indvars, 2);
%put &word2;
943  %let word2 = %scan(&indvars, 2);
944  %put &word2;
math
%let subword = %substr(&indvars, 5, 3);
%put &subword;
946  %let subword = %substr(&indvars, 5, 3);
947  %put &subword;
e m
```

The evaluation functions evaluate arithmetic and logical expressions. The following are examples of very basic arithmetic and logical evaluation functions.

```
%let k = 1;
%let tot = &k + 1;
%put &tot;
989   %let k = 1;
990   %let tot = &k + 1;
991   %put &tot;
1 + 1
%let tot = %eval(&k + 1);
%put &tot;
992   %let tot = %eval(&k + 1);
993   %put &tot;
2
```

The %eval function uses integer arithmetic. That means we will get an error message when any part of the expression is not an integer nor a logic statement. For example,

```
%let tot = %eval(&k + 1.234);
995   %let tot = %eval(&k + 1.234);
ERROR: A character operand was found in the %EVAL function or %IF condition
where a numeric
       operand is required. The condition was: 1 + 1.234
```

Instead, we can use %sysevalf function as shown in the following example.

```
%let tot = %sysevalf(&k + 1.234);
%put &tot;
996   %let tot = %sysevalf(&k + 1.234);
997   %put &tot;
2.234
```

Summary:

In this section, we have mentioned the following.

- string functions;
  - **%upcase**;
  - **%substr**;
  - **%scan**;
- evaluation functions;
  - **%eval**;
  - **%sysevalf**;

Symput and symget function to pass information to and from a data step

There are two functions that are particularly useful when we want to get information in and out of a data step. These are **symput** and **symget**. You use **symput** to get information from a data step into a macro variable and **symget** is used when we want to get information from a macro variable into a data step.

The syntax used is **CALL SYMPUT**(argument1, argument2), where **argument1** is the macro variable that we are creating which will store the value that is being passed out of the data step and **argument2** is the value in string format. Notice that the new macro variable has to be in single quotes.

```
proc means data = hsb2 n;
  var write;
  where write>=55;
  output out=w55 n=n;
run;
proc print data = w55;
run;
data _null_;
  set w55;
  call symput('n55', n);
run;
%put &n55 Observations have write >=55;
118  %put &n55 Observations have write >=55;
9 Observations have write >=55
```

The syntax for **symget** is **symget**(argument) where **argument** can be the name of a macro variable, a string variable or a character expression. Suppose that we want to create a new variable in the **hsb2** data set that is constant across the entire data set and the value for this variable is the number of students who have a writing score 55 or higher. We have already stored the number in the macro variable **n55** so this will be the argument for the **symget** function. Notice that even though **n55** is a macro variable we do not use the ampersand sign preceding **n55**, instead we use single quotes.

```
data hsb2_55;
  set hsb2;
  w55 = symget('n55')+0;
run;
proc print data = hsb2_55;
  var write w55;
run;
```

```
Obs    write    w55
  1      52      9
  2      59      9
  3      33      9
  4      44      9
  5      52      9
  6      52      9
  7      59      9
  8      46      9
  9      57      9
 10      55      9
 11      46      9
 12      65      9
 13      60      9
 14      63      9
 15      57      9
 16      49      9
 17      52      9
 18      57      9
```

Summary:

In this section, we have mentioned the following.

- **symput** — **call symput**(‘new_macro_variable’, value_in_string_format)
- **symget** —**symget**(‘macro_variable’)

Creating macro variables using proc sql

Another way of creating macro variables is through **proc sql**. SQL stands for Structured Query Language and is a standardized database language. **Proc sql** can create SAS macro variables that contains values from a query result. In the following example we create a macro variable called **w55**, which contains the number of students whose writing scores are higher than or equal to 55.

```
proc sql;
  select sum(write>=55) into :w55
  from hsb2;
quit;
%put w55 is &w55;
35   %put w55 is &w55;
w55 is        9
```

The example below shows how to create group means for each level of the variable **ses** and store them in three macro variables called **write1**, **write2** and **write3**. We make use of the **%put** function to display the values of the macro variables in the log file.

```
proc sql;
  select mean(write) into :write1 - :write3
  from hsb2
  group by ses;
quit;
%put write1 to write3 are &write1, &write2 and &write3;
311  %put write1 to write3 are &write1, &write2 and &write3;
write1 to write3 are 52.66667, 54.8 and 50.4
```

Summary:

In this section, we have mentioned the following.

- proc **sql** with **select into** statement to create macro variable(s);

Creating a list of file names for a data step using a macro program

Thus far we have gained familiarity with macro variables. Now we will use this knowledge to write some macro programs. A macro program always starts with the **%macro** statement including the user defined program name and it ends with a **%mend** statement. When SAS is going to compile a SAS program it first sends the program to a word scanner which intercepts the macro syntax before it can reach the compiler. The macro processor translates the macro syntax into standard SAS syntax which is then compiled. Thus, the macro language serves as a dynamic editor for SAS programs.

Let's first create some exercise data sets. In the following data step, we create four data files: **file1** – **file4**.

```
data file1 file2 file3 file4;
  input a @@;
  if _n_ <= 3 then output file1;
  if 3 < _n_<=  6 then output file2;
  if 6 < _n_ <= 9 then output file3;
  if 9 < _n_ <=12 then output file4;
cards;
1 2 3 4 5 6 7 8 9 10 11 12
;
run;
```

In the following program the goal is to stack a number of data sets together into one data set. Suppose we have four data sets that are named **file1**, **file2** and so forth. In a standard SAS program we would have to write out the names of all the files in the **set** statement. In the macro program we will demonstrate how the program will write the names of the files in the **set** statement for us.

In general, it is always a good idea to write a regular SAS program first, test it and then turn it into a macro program. For example, the following data step will be our base program for stacking the four files together.

```
data all;
  set
   file1
   file2
   file3
   file4
  ;
run;
```

How do we turn this piece of SAS program into a SAS macro program? We need to start with a **%macro** statement where we specify the name of the macro; then we write the program and finally we end the macro program with a **%mend** statement. The only part from the SAS program that we need to modify substantially is the **set** statement. Consider the macro program called **combine** in the following example. We need to create a **do** loop in the **set** statement in order to create the list of file names automatically rather than writing them out one by one.

```
%macro combine;
  data all_1;
    set
    %do i = 1 %to 4;
      file&i
    %end;
    ;
  run;
%mend;
```

We submit the macro program in the same way as we submit a SAS program. The program can then be executed by submitting the following code which consists of a percent sign followed by the name of the macro program. Note that macro programs are called in a statement, which unlike all standard SAS programs, does NOT end in a semicolon. Another point of interest is that our macro does not take any arguments. In order to see what is going on behind the scene, we turn on a SAS system option called **mprint** (for macro print). It will print out SAS statements generated by macro execution.

```
*executing the combine program;
options mprint;
%combine
```

Here is what has happened in the log window:

```
167  %combine
MPRINT(COMBINE):   data all_1;
MPRINT(COMBINE):   set file1 file2 file3 file4 ;
MPRINT(COMBINE):   run;
NOTE: There were 3 observations read from the data set WORK.FILE1.
NOTE: There were 3 observations read from the data set WORK.FILE2.
NOTE: There were 3 observations read from the data set WORK.FILE3.
NOTE: There were 3 observations read from the data set WORK.FILE4.
NOTE: The data set WORK.ALL_1 has 12 observations and 1 variables.
NOTE: DATA statement used:
      real time           0.02 seconds
      cpu time            0.02 seconds
```

Ideally we would like to be able to stack any number of data sets into one long data set. The current macro program stacks exactly four data sets together, no more and no less. So, we would like to generalize the program to take an argument which will specify how many data sets we are stacking in any specific execution of the program.

When a macro program takes arguments we list the names of the arguments in parenthesis after the name of the program in the **%macro** statement. In the following example we include an argument called **num** in the new version of the **combine** program. Inside the macro program we use **&num** to refer to the value passed by the argument. **&num** is now a local macro variable which only “lives” inside the combine macro program. If we refer to **&num** outside the **combine** program SAS will have no idea what we are talking about and we will get an error indicating that the reference to **&num** was unresolved.

The only other change to the program is that instead of executing the **do** loop exactly four times we now execute it **&num** number of times. At the end of the code when we finally execute the new version of the **combine** program we specify that we want to execute the **do** loop three times thus stacking together **file1**, **file2** and **file3**.

```
%macro combine(num);
  data big;
    set
    %do i = 1 %to &num;
      file&i
    %end;
    ;
  run;
%mend;

*executing the macro program;
%combine(3)
180  %combine(3)
MPRINT(COMBINE):   data big;
MPRINT(COMBINE):   set file1 file2 file3 ;
MPRINT(COMBINE):   run;
NOTE: There were 3 observations read from the data set WORK.FILE1.
NOTE: There were 3 observations read from the data set WORK.FILE2.
NOTE: There were 3 observations read from the data set WORK.FILE3.
NOTE: The data set WORK.BIG has 9 observations and 1 variables.
```

A macro program for repeating a procedure multiple times

Suppose that we have a number of binary dependent variables and two independent variables. Our task is to fit a logistic model for each of the dependent variables on the same two independent variables. We could simply write a **proc logistic** for each model but this would be tedious and typing intensive. Instead we choose to write a macro program which will automatically cycle through all the dependent variables and fit a logistic model to each one of the dependent variables.

Let's first create a data set which consists of the dependent variables **v1** to **v5** and predictors **ind1** and **ind2**.

```
data xxx;
  input v1-v5 ind1 ind2;
  cards;
1 0 1 1 0 34 23
0 0 1 0 1 22 32
1 1 1 0 0 12 10
0 1 0 1 1 56 90
0 1 0 1 1 26 80
1 1 0 0 0 46 45
```

```
0 0 0 1 1 57 53
1 1 0 0 0 22 77
0 1 0 1 1 44 45
1 1 0 0 0 41 72
;
run;
```

To get a better idea of how we will write the macro program let us first write a standard SAS program for fitting the logistic model to **v1**.

```
proc logistic data = xxx descending;
  model v1 = ind1 ind2;
run;
```

What part of the program do we have to change? The key change will be in the **model** statement. The following program will demonstrate one way of changing it. We create a **do** loop which will iterate through each of the dependent variables and fit a logistic model for each one. We include a number argument, called **num**, which will specify how many dependent variables we will be using. The **do** loop takes advantage of the naming convention of the dependent variables.

```
%macro mylogit(num);
   %do i = 1 %to &num;
      title "dependent variable is v&i";
      proc logistic data=xxx des;
        model v&i = ind1 ind2;
      run;
  %end;
%mend;
*executing the macro using 5 dependent variable;
%mylogit(5)
```

This was merely the first attempt to automate the repetitive process. We can further modify the program in many different ways.

Debugging a macro program

Before modifying our macro program, let's pause for a second. When we write SAS macro programs, SAS actually will try to help us to detect errors in the program. Two SAS options are particularly useful: **mprint** and **mlogic**. We have seen how **option mprint** helps us to see the translation process from a macro program to regular SAS statements. Let's add these two options along with other SAS options. Notice that, SAS spills out all the relevant information related to a macro program or macro variable to log window. The other way to debug is to use the %put statement manually inside our macro program. For example, in the example below, %put is used after the looping. We can see if the looping stops correctly this way.

```
options mprint mlogic;
%macro mylogit(num);
  %do i = 1 %to &num;
    proc logistic data=xxx des;
      model v&i = ind1 ind2;
    run;
  %end;
  %put &i;
%mend;

*executing the macro using 5 dependent variable;
%mylogit(5)
...
...
...
MLOGIC(MYLOGIT):  %DO loop index variable I is now 6; loop will not iterate
again.
MLOGIC(MYLOGIT):  %PUT &i
6
MLOGIC(MYLOGIT):  Ending execution.
```

Specifying dependent variable names

There are some limitation to the **mylogit** macro program in its current form; it only works iteratively when the dependent variable names are of the form **v1**, **v2** and so forth. We would like to modify the **mylogit** macro to be able to take any type of dependent variable names and we would like to be able to simply pass the macro a variable list as an argument and then the macro will fit a model to every variable in that list. To accomplish this goal we make use of the macro function **%scan** which will scan the list of dependent variables one at a time. The name of the dependent variable will then be stored in the local macro variable **dep** which is then passed in to the logistic procedure. The **while** loop works in that we are asking SAS to iterate the process until **dep** is equal to missing, in other words, the loop iterates until the end of the list. We increment the local macro variable **k** for each iteration of the while loop because **&k** is the position indicator in the variable list. Thus, for the first iteration of the **while** loop **&k**=1, and the **scan** function stores the first variable in the dependent variable list in the local macro variable **dep**. Then SAS fits a logistic model using the first variable in the list as the dependent variable and then it increments **&k**=2. Now **scan** stores the second variable in the dependent variable list in **dep** and this variable is used as the dependent variable in the logistic procedure. This continuous until the dependent variable list has been exhausted at which point **dep** will be equal to missing and SAS will exit the **while** loop.

```
%macro mylogit1(all_deps);
  %let k=1;
  %let dep = %scan(&all_deps, &k);
  %do %while("&dep" NE "");
    title "dependent variable is &dep";
    proc logistic data=xxx des;
      model &dep = ind1 ind2;
    run;
    %let k = %eval(&k + 1);
    %let dep = %scan(&all_deps, &k);
  %end;
%mend;

*run the program for the first three v's;
%mylogit1(v1 v2 v3)
```

Saving the estimates to a data set

The next generalization that we would like to implement is to be able to save the estimates from each logistic model in a data set. So, the macro program will now take two arguments: **all_dep** which is the dependent variable list and **outest** which is the name of the data set containing the estimates of all the logistic models. The **mylogit1** macro program takes uses the **outest** option in the **proc logistic** statement to create a data set containing the parameter estimates for all the model fitted. The parameter estimates for the first model fitted will be stored in the data set called **_est1**, the estimates for the second model in the data set **_est2** and so forth. If we specify a name for the **outest** argument then the program tells SAS to stack all the data sets containing the parameter estimates in a data set with this name. If we do not specify a name then the program uses a **proc datasets** to delete all the data sets containing the parameter estimates.

```
%macro mylogit1(all_deps, outest);
  %let k=1;
  %let dep = %scan(&all_deps, &k);
  %do %while("&dep" NE "");
    title "dependent variable is &dep";
    proc logistic data=xxx des outest=_est&k;
      model &dep = ind1 ind2;
    run;
    %let k = %eval(&k + 1);
    %let dep = %scan(&all_deps, &k);
  %end;
  %if "&outest" NE "" %then
  %do;
    data &outest;
      set
      %do i = 1 %to &k - 1;
        _est&i
      %end;
      ;
    run;
    %let k = %eval(&k - 1);
    proc datasets;
      delete _est1 - _est&k;
    run;
  %end;
  %else
  %do;
     %put no dataset name was provided, files are not combined;
  %end;
%mend;
%mylogit1(v1 v2 v3)

%mylogit1(v1 v2 v3, a)
proc print data = a;
  var intercept ind1 ind2;
run;
```

```
Obs    Intercept      ind1         ind2

 1       2.4570     -0.04282    -0.01709
 2       0.3278     -0.09480     0.09078
 3      33.3421     -0.50434    -0.40122
```

A more flexible version of the same macro program

We can generalize the macro program even more. The new version of the macro program called mylogita will allow the user to specify an input data file, a list of dependent variables, a list of predictors and an output data set containing the parameter estimates. This macro program actually contains two types of arguments: positional arguments and non-positional arguments. The non-positional arguments are followed by an equal sign and possibly a default value. The argument indvars is an example of a non-positional argument which does not have a defaults value and the argument outest is a non-positional argument with the default value of _out. The arguments indata and all_deps are both examples of positional arguments. The difference between these types of arguments occur when we want to execute the macro program. Positional arguments must appear in the code executing the macro in the exact same order they appear in the macro program. In other words, the name of the input data set has to be the first argument in the code, the list of dependent variables has to be the second argument. The order of the list of independent variables and the name of the data set containing the parameter estimates is not fixed. We can change the order of these arguments, all we have to do is specify which argument we are giving the value for by including the name of the argument and an equal sign and the value. Thus, in the first example where we execute the mylogita macro we declare that the list of independent variables should include ind1 and ind2 (by specifying indvars = ind1 ind2) and that the name of the data set containing the parameter estimates should be myparms (by specifying outest = myparms). In the second example we switch the order of these two arguments without any problems since we use the argument name, equal sign and value syntax.

```
%macro mylogita(indata, all_deps, indvars =, myout =_out );
  %let k=1;
  %let dep = %scan(&all_deps, &k);
  %do %while(&dep NE);
    title "The dependent variable is &dep";
    title2 "The independent variables are &indvars";
    proc logistic data=&indata des outest=est&k;
      model &dep = &indvars;
    run;
    %let k = %eval(&k + 1);
    %let dep = %scan(&all_deps, &k);
  %end;
  data &myout;
    set
    %do i = 1 %to &k - 1;
      est&i
    %end;
    ;
  run;
%mend;
*run the program;
%mylogita(xxx, v1 v2 v3, indvars = ind1 ind2, myout = myparms)

title;
proc print data = myparms;
  var _name_ intercept ind1 ind2;
run;
```

```
Obs    _NAME_    Intercept      ind1        ind2
 1       v1        2.4570     -0.04282    -0.01709
 2        v2         0.3278     -0.09480     0.09078
 3        v3        33.3421     -0.50434    -0.40122
* run the program again: unpositional arguments can be reordered;
%mylogita(hsb2,female, myout = myparm1, indvars = write math)

title;
proc print data = myparm1;
  var _name_ intercept write math;
run;
Obs     _NAME_     Intercept       write         math
 1      female      -3.49607    0.068307    -0.022230
```

Summary:

In this long section, we have mentioned the following.

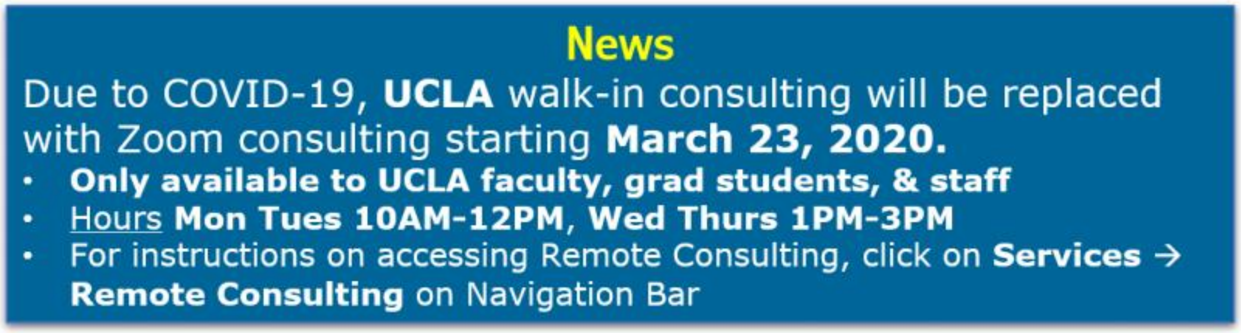
- defining a SAS macro program with %macro and %mend;
- making use of %let statement to create macro variables inside a macro program;
- making use of macro functions such as %scan and %eval;
- how to call a SAS macro program (executing a macro program);
- how to debug a SAS macro program;
- positional vs. non-positional arguments;

Reference

1. SAS Code Fragments: Combining Multiple SAS Files Using a Macro with the “set” command
2. SAS Macro Language Documentation

Primary Sidebar

Footer



News

Due to COVID-19, **UCLA** walk-in consulting will be replaced with Zoom consulting starting **March 23, 2020.**

- **Only available to UCLA faculty, grad students, & staff**
- Hours **Mon Tues 10AM-12PM, Wed Thurs 1PM-3PM**
- For instructions on accessing Remote Consulting, click on **Services → Remote Consulting** on Navigation Bar