Chapter 12

Introduction to Advanced Programming in STATA

Aim: Upon completing the chapter, the learner should be able to develop short programs that will make the existing Stata commands run more efficiently.

12.1 Introduction

Stata provides an editor window to save Stata commands and user-defined commands. These files can be executed within this editor or they can be called for execution within another do-file. In this chapter, we will present an introduction about how to prepare do-files and the structure to define program commands (Juul, 2014).

12.2 do-files

The do-file editor tool can be used for data management and to create programs. There are four ways to open a new do-file: (1) the Windows menu (Window → do-file editor); (2) the keyboard (press Crtl+9); (3) the Windows icon (using the new do-file editor); and (4) using the command line *doedit*.

Example 1

Open a new do-file editor using the command window by typing "doedit." A do-file editor page will open. Create the following do-file:

```
noisily display "Introduction to STATA Programming"
noisily display "Example 1: New Do-file"
noisily display "END"
```

Then, save it under "\Users\Documents\students\example1.do."

To run the example1.do file, use the command window. To do this, you can type

```
cd "\Users\Documents\students"
do "example1.do",
```

and the following will appear:

```
. noisily display "Introduction to STATA Programming"
Introduction to STATA Programming
. noisily display "Example 1: New Do-file"
Example 1: New Do-file
. noisily display "END"
END
```

Or you can type

```
run example1,
```

and the following will appear:

```
Introduction to STATA Programming
Example 1: New Do-file
END
```

The commands ***noisily*** and ***quietly*** are special commands that turn the output on and off. The first, ***noisily***, performs the command subsequently written and ensures terminal output. The second, ***quietly***, performs the command subsequently written but suppresses terminal output. As you can see in the example above, if you type "do" before typing "example1," all the information in the do-file will be displayed in the Stata results window. If you type "run," only the information after the ***noisily display*** commands will be shown in the output window.

12.3 *program* Command

The *program* command is used to do data management and to run analyses. The following is the basic structure to use the program command:

```
program program name
{a series of STATA commands}
end
```

When you are writing a program, the program name has to be unique and cannot be the same as any other command name. For example, you cannot use the name "ttest" because it is a built-in command in Stata. To be able to find out whether a

name is already in use by Stata, you can use the *which* command. In the command window type *which program name*. Doing so will result in the following:

```
. which ttest
C:\Program Files (x86)\Stata\ado\base\t\ttest.ado
*! version 4.1.1  30dec2004
. which example
command example not found as either built-in or ado-file
r(111);
```

As you can see in the previous example, a program named "ttest" is being used by Stata, and the program name "example" is not being used. *Example2.do* illustrates the use of the *program* command using the command window. Type the following lines in the command field:

```
program example2
display "Example 2: How to use the command program"
display "STATA commands"
display "End of the Example"
end
```

To execute the program, type "example2" in the command line; the output after having done so will be:

```
Example 2: How to use the command program
STATA commands
End of the Example
```

If you want to change or edit a program, you will need to first delete that program from the memory. If you try to use the name "example2" again, you will get the following error:

```
program example2
example3 already defined
r(110);
```

To delete this error or cause it to be ignored, you can use the commands *drop* and *capture.* The *drop* command deletes the program from the memory, and the *capture* command causes the errors associated with the command that follows the *capture* command to be ignored. Type the following example in the command line, or create a do-file with these commands:

```
capture program drop example3
program example3
display "Example 3: How to use the commands drop and capture"
display "STATA commands"
display "End of the Example"
end
```

Then, type "example3" in the command window.

```
Example 3: How to use the commands drop and capture
STATA commands
End of the Example
```

12.4 Log Files

It is important to keep track of your work while you are programming. If you keep a log of your work, you will find it easier to go back and check what you have already done. To create a log, use the following steps:

1. To start:
   Menu toolbar → File – Log – Begin → Save as *filename*
2. To finish:
   Menu toolbar → File – Log – Close

Or, write the following in the command window:

```
log using "C:\MyPrograms\mylog", replace
{a series of STATA commands and their outputs}
log close
```

For example, prepare a do-file, named "example4," as follows:

```
log using "example4", replace text
use "/Users/Documents/bmi.dta"
tab bmig
       bmig |      Freq.     Percent        Cum.
------------+-----------------------------------
     Normal |          4       40.00       40.00
 Overweight |          3       30.00       70.00
      Obese |          3       30.00      100.00
------------+-----------------------------------
      Total |         10      100.00
log close
```

After *log using* command, all the commands and their outputs will be saved in example4.txt until *log close* command.

12.5 *trace* Command

While you are programming, sometimes you will get errors after executing a given program. The *trace* command is a useful tool for finding these errors. You can turn this command on or off at any time while you are programming. The following do-file, named "example5," illustrates how to use the *trace* command:

```
capture log close

log using "example5.log", replace
```

```
capture program drop _all
 program example5
      display "Example #5"
      display "Runs the error when you run the program"
      display "Stops and displays the error when executing the
      program"
      display "End of the Example"
      ERRORRRRRR
 end
log close
set trace on
set more off
example5
```

If you run the do-file named "example5," you will get the following error:

```
. example5
  ------------------------------------------- begin example5 ---
  - display "Example #5"
Example #5
  - display "Runs the error when you run the program"
Runs the error when you run the program
  - display "Stops and displays the error when executing the
program"
Stops and displays the error when executing the program
  - display "End of the Example"
End of the Example
  - ERRORRRRRR
unrecognized command:  ERRORRRRRR
  ------------------------------------------- end example5 ---
r(199);
end of do-file
```

12.6 Delimiters

Stata reads each line as a complete command line, but sometimes the commands are long. To be able to use more than one line as your command line you can use delimiters. There are two types of delimiters, one you can use in each line (**///**), and one you set up before running your Stata commands (**#d ;**). The following examples show how each of the two delimiters is used to create a two-way graph:

```
twoway (scatter bmi age if sex==0, sort mcolor(navy)              ///
         msymbol(circle_hollow))                                  ///
     (scatter bmi age if sex==1, sort mcolor(maroon)  msymbol(circle)) ///
     (line    bmi age if sex==0, sort lcolor(navy)   lwidth(thick))  ///
```

```
     (line    bmi age if sex==1, sort lcolor(maroon) lwidth(thick)),///
     legend(position(10) ring(0) col(1) order(1 "Males" 2 "Females")///
     region(fcolor(none) lcolor(none)))                              ///
ylab(,angle(horizontal))ytitle("BMI")xtitle("Age")graphregion(fcolor
(white))
```

And

```
#d ;
twoway (scatter bmi age if sex==0, sort mcolor(navy) msymbol(circle_hollow))
(scatter bmi age if sex==1,   sort mcolor(maroon) msymbol(circle))
(line    bmi age if sex==0,   sort lcolor(navy)   lwidth(thick))
(line    bmi age if sex==1,   sort lcolor(maroon) lwidth(thick)),
legend(position(10) ring(0) col(1) order(1 "Males" 2 "Females")
region(fcolor(none) lcolor(none))) ylab( , angle(horizontal))
ytitle("BMI") xtitle("Age") graphregion(fcolor(white));
#d cr
```

As you can see above, you need to open with **#d ;** and then close with **#d cr** for the next command lines. If you do not close the delimiter, Stata will continue to read all the lines continuously.

12.7 Indexing

When you execute a Stata command, the command will loop across each line of the dataset. For example, if you generate a new variable, Stata will work in line 1, then line 2, and so on. The use of indexing will help the user to run only Stata commands in certain observations. The following are examples of indexing:

1. Generate a new variable, *x,* that contains the number of the current observation:

   ```
   gen x=_n
   ```

   Output

   ```
   .  list x

      +---+
      | x |
      |---|
   1. | 1 |
   2. | 2 |
   3. | 3 |
   4. | 4 |
   5. | 5 |
      |---|
   6. | 6 |
   7. | 7 |
      +---+
   ```

2. Generate a new variable, *y,* which contains the total number of observations in the dataset, assuming the last dataset:

```
gen y=_N

. list x y

   +-------+
   | x   y |
   |-------|
1. | 1   7 |
2. | 2   7 |
3. | 3   7 |
4. | 4   7 |
5. | 5   7 |
   |-------|
6. | 6   7 |
7. | 7   7 |
   +-------+
```

3. To check for duplicates in your dataset, assuming every subject has an id, which is identified in this dataset by ID and it is a sequential set of numbers starting with 1, use the following command line:

```
bysort ID: gen duplicates = _n
```

4. To create a variable with the total number of subjects in a group, where these groups are identified by *groupid*, use the following command line:

```
bysort groupid: gen subjects = _N
```

5. Generate two new variables, *z* and *w*. Variable *z* contains the current observation minus 1. The first observation will be missing. Variable *w* contains the current observation plus 1. The last observation will be missing. Observe:

```
gen z=x[_n-1]
gen w=x[_n+1]

list  x y z w
   +---------------+
   | x   y   z   w |
   |---------------|
1. | 1   7   .   2 |
2. | 2   7   1   3 |
3. | 3   7   2   4 |
4. | 4   7   3   5 |
5. | 5   7   4   6 |
6. | 6   7   5   7 |
7. | 7   7   6   . |
   +---------------+
```

12.8 Local Macros

Local macros are temporary variables in the memory for loops and programs. A local macro can be a number or a string of characters (in either case, up to 31 characters can be used). To exemplify these macros, let's assume the following database:

```
. list
```

	age	bmi	hgb	smoke
1.	18	18	11.3	1
2.	19	24	14	1
3.	23	27	14.5	1
4.	25	24	14.7	0
5.	37	28	15	0
6.	56	29	13	0
7.	78	32	12	0
8.	52	23	11	1
9.	21	24	14	1
10.	45	20	11.5	1
11.	25	24	14.7	0
12.	34	20	12	0
13.	59	29	13	0
14.	78	32	12	0

If we are interested in using age and hemoglobin (hgb) levels as predictors of bmi, we could define the list of predictors and then run a multivariate linear regression model, as follows:

```
local list = "age hgb"
reg bmi 'list'
```

Output

```
  Source |       SS           df       MS      Number of obs   =        14
---------+------------------------------       F(2, 11)        =     43.77
   Model | 221.078339         2  110.539169    Prob > F        =    0.0000
Residual | 27.7788042        11  2.52534584    R-squared       =    0.8884
---------+------------------------------       Adj R-squared   =    0.8681
   Total | 248.857143        13  19.1428571    Root MSE        =    1.5891

------------------------------------------------------------------------------
     bmi |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
---------+--------------------------------------------------------------------
     age |   .2179857   .0239641     9.10   0.000      .165241    .2707304
     hgb |   2.241635   .3550485     6.31   0.000     1.460178    3.023091
   _cons |  -12.84275   5.193205    -2.47   0.031    -24.27292   -1.412584
------------------------------------------------------------------------------
```

12.9 Scalars

Scalars are temporary results that are saved in the memory after a command is run. After you run a command, you can review which scalars were saved using the *return list* command. For example, let's assume that we have the variable *smoke* from the previous database, and we want to run a Student's *t*-test to compare the expected bmi by smoke. The following is what that would look like:

```
ttest bmi, by(smoke)
```

Output

```
Two-sample t test with equal variances
------------------------------------------------------------------------------
   Group | Obs       Mean  Std. Err.  Std. Dev.  [95% Conf.  Interval]
---------+--------------------------------------------------------------------
       0 |   8     27.25  1.497021   4.234214   23.71011      30.78989
       1 |   6  22.66667  1.308094   3.204164    19.3041      26.02923
---------+--------------------------------------------------------------------
combined |  14  25.28571  1.169336   4.375255   22.75952      27.81191
---------+--------------------------------------------------------------------
    diff |       4.583333   2.07317              .0662847      9.100382
------------------------------------------------------------------------------
   diff = mean(0) - mean(1)                                      t =    2.2108
Ho: diff = 0                     degrees of freedom =        12

   Ha: diff < 0                 Ha: diff != 0                 Ha: diff > 0
 Pr(T < t) = 0.9764    Pr(|T| > |t|) = 0.0472          Pr(T > t) = 0.0236
```

After Student's *t*-test, we use the *return* command, as follows:

```
return list
```

After doing so, the following results should appear:

```
scalars:
              r(level) =  95
                 r(sd) =  4.375255094603872
               r(sd_2) =  3.204163957519444
               r(sd_1) =  4.234214381508266
                 r(se) =  2.073169652345752
                r(p_u) =  .0236068853559555
                r(p_l) =  .9763931146440445
                  r(p) =  .047213770711911
                  r(t) =  2.210785464733855
               r(df_t) =  12
               r(mu_2) =  22.66666666666667
               r(N_2) =  6
               r(mu_1) =  27.25
               r(N_1) =  8
```

Scalars are useful for displaying only the results you want, instead of displaying all the results. Here is an example:

```
. noisily display "Pr(|T| > |t|) =" %9.3f 'r(p)'
Pr(|T| > |t|) =    0.047
```

In addition, you can create new scalars to calculate results not included in the saved results. In the following example, using the previous database, the mean difference between two groups is calculated:

```
capture program drop example6
program example6, rclass
  summarize '1' if '2'==0, meanonly
  scalar mean1 = r(mean)
  summarize '1' if '2'==1, meanonly
  scalar mean2 = r(mean)
  return scalar diff = mean1 - mean2
end
```

After running the program named "example6," the following is returned:

```
. example6 bmi smoke

. return list

scalars:

r(diff) =  4.583333333333332
```

12.10 Loops (*foreach* and *forvalues*)

The command *foreach* repeatedly executes the commands enclosed inside the braces, as can be seen in the following:

```
foreach lname {in|of listtype} list {
commands referring to 'lname'
}
```

Here is an example that uses the previous database with the following do-file:

```
foreach var of var bmi age hgb {
mean 'var'
}
```

Output

```
Mean estimation                   Number of obs    =         14
--------------------------------------------------------------
             |       Mean   Std. Err.      [95% Conf. Interval]
-------------+------------------------------------------------
         bmi |   25.28571   1.169336       22.75952    27.81191
--------------------------------------------------------------

Mean estimation                   Number of obs    =         14

--------------------------------------------------------------
             |       Mean   Std. Err.      [95% Conf. Interval]
-------------+------------------------------------------------
         age |   40.71429   5.614374      28.58517      52.8434
--------------------------------------------------------------

Mean estimation                   Number of obs    =         14

--------------------------------------------------------------
             |       Mean   Std. Err.      [95% Conf. Interval]
-------------+------------------------------------------------
         hgb |      13.05   .3789444      12.23134     13.86866
--------------------------------------------------------------
```

In addition, you can use the *local* command in the do-file, as can be seen in the following:

```
local variables = "bmi age hgb"
foreach var of local variables {
sum 'var'
}
```

After doing so, the following results will appear:

```
  Variable |       Obs        Mean    Std. Dev.       Min        Max
-----------+---------------------------------------------------------
       bmi |        14    25.28571    4.375255        18         32

  Variable |       Obs        Mean    Std. Dev.       Min        Max
-----------+---------------------------------------------------------
       age |        14    40.71429    21.00706        18         78

  Variable |       Obs        Mean    Std. Dev.       Min        Max
-----------+---------------------------------------------------------
       hgb |        14       13.05     1.41788        11         15
```

The command *forvalues* loops over consecutive values, using the following structure:

```
forvalues lname = range {
commands referring to 'lname'
}
```

For example, assuming we want to generate two random variables with uniform distribution between the numbers 1 and 14, and assuming we are using the previous bmi database, the do-file will be composed of the following commands:

```
forvalues i = 1(1)2  {
generate x`i' = 1+ int(runiform()*14)
}
```

Once the above *forvalues* command is run, the variables *x*1 and *x*2 are generated. To explore the values of these variables, we use *list*, as is demonstrated in the following:

```
. list x1 x2
     +---------+
     | x1   x2 |
     |---------|
  1. |  7   13 |
  2. | 11   13 |
  3. | 13   11 |
  4. |  2   13 |
  5. |  7   10 |
     |---------|
  6. | 13    4 |
  7. | 11   12 |
  8. |  4    1 |
  9. |  3   13 |
 10. | 11    5 |
     |---------|
 11. | 14   11 |
 12. | 11    9 |
 13. | 13    5 |
 14. |  4    3 |
     +---------+
```

Assuming we would like to select those persons for whom *x*1 is greater than *x*2 for further assessment, we would use the following commands:

```
gen id=_n
gen selec=(x1 > x2)
list id age bmi hgb smoke if selec==1
```

Output

```
     +-------------------------------+
     | id   age   bmi    hgb   smoke |
     |-------------------------------|
 3.  |  3    23    27   14.5       1 |
 6.  |  6    56    29     13       0 |
 8.  |  8    52    23     11       1 |
10.  | 10    45    20   11.5       1 |
11.  | 11    25    24   14.7       0 |
     |-------------------------------|
12.  | 12    34    20     12       0 |
13.  | 13    59    29     13       0 |
14.  | 14    78    32     12       0 |
     +-------------------------------+
```

Therefore, the individuals to be selected will be those with id*s* 3,6,8,10,11,12,13, and 14.

12.11 Application of *matrix* and *local* Commands for Prevalence Estimation

If we want to estimate the prevalence of one particular event, there are different Stata commands for performing this process, which include *proportion* and *glm*. The *proportion* command uses a normal approach (Rosner, 2010), and the *glm* command uses a logistic regression model (Hosmer and Lemeshow, 2000). For example, assuming we are interested in estimating the prevalence of women from the previous database who have a hemoglobin level below 12, the syntaxes with the *proportion* command will be as follows:

```
gen nhgb=hgb < 12
proportion nhgb
```

Output

```
Proportion estimation             Number of obs    =         14
--------------------------------------------------------------
             | Proportion   Std. Err.      [95% Conf. Interval]
-------------+------------------------------------------------
nhgb         |
           0 |   .7857143   .1138039      .4598449     .9404495
           1 |   .2142857   .1138039      .0595505     .5401551
--------------------------------------------------------------
```

The prevalence estimate of a hemoglobin level below 12 is 21.4% (95% CI: 5.9, 54.0%).

If we want to use the *glm* command for this estimation, we will use the logistic regression model with no predictor variables, as follows:

$$\text{Prevalence} = \frac{1}{1+e^{-\beta_0}}$$

where β_0 is the intercept. The syntaxes for a prevalence estimate of hemoglobin levels below 12, after running the *glm* command, will be based on the *matrix* and *local* commands and are seen in the following:

```
quietly: glm nhgb , fam(bin)
matrix def b=e(b)
matrix def v=e(V)
local c=b[1,1]
local es=sqrt(v[1,1])
gen prev=100/(1+exp(-`c'))
gen previnf=100/(1+exp(-(`c'-1.96*`es')))
gen prevsup=100/(1+exp(-(`c'+1.96*`es')))
collapse (mean) prev previnf prevsup
list
```

The output of the above will be:

```
    +---------------------------------+
    |     prev    previnf    prevsup |
    |---------------------------------|
 1. | 21.42857   7.070517   49.43356 |
    +---------------------------------
```

The point estimates of this prevalence are the same, but the confidence limits are different, probably because of the small sample size for the normal approach used in the *proportion* command.

The other option for prevalence estimation is to use the *adjust* command after the *logit* command, as is demonstrated in the following:

```
logit nhgb
adjust ,pr ci
```

Output

```
Logistic regression                 Number of obs     =         14
                                    LR chi2(0)        =       0.00
                                    Prob > chi2       =          .
Log likelihood = -7.2741177         Pseudo R2         =     0.0000
```

```
------------------------------------------------------------------------------
  nhgb |   Coef.    Std. Err.   z    P>|z|   [95% Conf. Interval]
-------+----------------------------------------------------------------------
 _cons | -1.299283 .6513389  -1.99 0.046  -2.575884  -.0226821
------------------------------------------------------------------------------

. adjust ,pr ci

------------------------------------------------------------------------------
  Dependent variable: nhgb   Equation: nhgb     Command: logit
------------------------------------------------------------------------------


------------------------------------------------
   All |            pr           lb           ub
-------+----------------------------------------
       |       .214286    [.070707     .49433]
------------------------------------------------
  Key:   pr           =  Probability
         [lb , ub]    =  [95% Confidence Interval]
```

The results are the same as those obtained with the *glm* command. That is, the prevalence estimate of hemoglobin levels below 12 is 21.4% (95% CI: 7.07%, 49.43%).

When the logistic regression model includes predictors, prevalence estimation can be performed setting the value of only one of the predictors. For example, if we run the previous logistic model with *age* as the predictor, the prevalence can be estimated at *mean bmi* and at *bmi equal to 20,* as follows:

```
logit nhgb bmi
adjust , pr ci
adjust bmi=20, pr ci
```

Output

```
Logistic regression                             Number of obs     =         14
                                                LR chi2(1)        =       7.09
                                                Prob > chi2       =     0.0078
Log likelihood =  -3.729784                     Pseudo R2         =     0.4873

------------------------------------------------------------------------------
    nhgb |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
---------+--------------------------------------------------------------------
     bmi |  -.7007166   .4025497    -1.74   0.082    -1.489699    .0882662
   _cons |   14.81156   8.899236     1.66   0.096    -2.630626    32.25374
------------------------------------------------------------------------------

.  adjust , pr ci
```

```
------------------------------------------------------------------------------
     Dependent variable: nhgb     Equation: nhgb     Command: logit
    Variable left as is: bmi
------------------------------------------------------------------------------

------------------------------------------------
      All |         pr           lb           ub
----------+-------------------------------------
          |     .05183     [.00225    .569917]
------------------------------------------------
     Key:  pr         =  Probability
           [lb , ub]  =  [95% Confidence Interval]

.  adjust bmi=20, pr ci

------------------------------------------------------------------------------
     Dependent variable: nhgb     Equation: nhgb     Command: logit
 Covariate set to value: bmi = 20
------------------------------------------------------------------------------

------------------------------------------------
      All |         pr           lb           ub
----------+-------------------------------------
          |     .68938    [.165612    .961265]
------------------------------------------------
     Key:  pr         =  Probability
           [lb , ub]  =  [95% Confidence Interval]
```

The prevalence estimate of hemoglobin levels below 12 set at mean bmi is 5.2% (95% CI: 0.22%, 57.0%). The prevalence estimate of hemoglobin levels below 12 for those subjects with bmi equal to 20 is 68.9% (95% CI: 16.56%, 96.13%). Although the bmi predictor in the model is marginally significant (*P*-value = .082), the prevalence estimates at different bmi values are quite different.

There are other options of programming that can be explored in Stata, including different procedures for matrix operations using *Mata* functions. These topics are beyond the scope of this book, so we recommend checking out the books by Acock (*A Gentle Introduction to Stata,* 4th edition, 2014) and by Baum (*An Introduction to Stata Programming,* 2009).