Introduction to Stata

Tebila Nakelse, Kansas State University

7 July 2017, Version 1

Introduction

Stata is a powerful statistical package with smart data-management facilities. Stata is a fast and easy to use software. In this tutorial I start with a quick introduction and overview, macros in Stata and some best practices in using and organising stata files .

1. Quick Tours

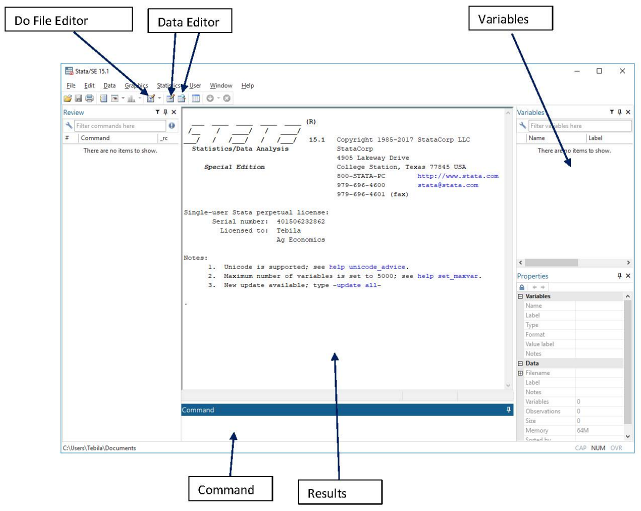
1.1 Quick tour of Stata capacity

- Stata is available for Windows, Unix, and Mac computers. This tutorial focuses on the Windows version, but most of the contents applies to the other platforms as well.
- The standard version is called Stata/IC (or Intercooled Stata) and can handle up to 2,047 variables.
- There is a special edition called Stata/SE that can handle up to 32,766 variables (and also allows longer string variables and larger matrices),and a version for multicore/multiprocessor computers called Stata/MP, allows larger datasets and is substantially faster.
- The number of observations is limited by your computer's memory, as long as it doesn't exceed about two billion in Stata/SE and about a trillion in Stata/MP.
- There are versions of Stata for 32-bit and 64-bit computers; the latter can handle more memory (and hence more observations) and tend to be faster.

1.2 Quick tour of User Interface

- The window labeled Command is where you type your commands.

- Stata then shows the results in the larger window immediately above, called appropriately enough Results.
- The window labeled Variables, on the top right, lists the variables in your dataset. The Properties window immediately below that, displays properties of your variables and dataset.
- Your command is added to a list in the window labeled Review on the left, so you can keep track of the commands you have used.
- You can resize or even close some of these windows. Stata remembers its settings the next time it runs. You can also save (and then load) named preference sets using the menu Edit|Preferences.
- You can also choose the font used in each window, just right click and select font from the context menu my own favorite on Windows is Lucida Console.
- Finally, it is possible to change the color scheme, selecting from seven preset or three customizable styles. One of the preset schemes is classic, the traditional black background used in earlier versions of Stata.



```
  ___  ____  ____  ____  ____ (R)
 /__    /   ____/   /   ____/
___/   /   /___/   /   /___/   15.1   Copyright 1985-2017 StataCorp LLC
  Statistics/Data Analysis            StataCorp
                                      4905 Lakeway Drive
     Special Edition                  College Station, Texas 77845 USA
                                      800-STATA-PC        http://www.stata.com
                                      979-696-4600        stata@stata.com
                                      979-696-4601 (fax)

Single-user Stata perpetual license:
       Serial number:  401506232862
         Licensed to:  Tebila
                       Ag Economics

Notes:
      1.  Unicode is supported; see help unicode_advice.
      2.  Maximum number of variables is set to 5000; see help set_maxvar.
      3.  New update available; type -update all-

.
```

Figure 1: User Interface

** Example commands **

```
.     display 2+2
4

.
.     display 2*ttail(20, 2.1)
.04861759
```

1.4 Stata Command Syntax

There are two ways that a Stata program can interpret what the user types:

- positionally, meaning first argument, second argument, and so on, or
- according to a grammar, such as standard Stata syntax.

The full stata command structure can be represented as follows:

```
[by varlist:] command [varlist|varname|namelist] [=exp] [if exp] [in range]
              [weight][using filename][,options]
```

In stata syntax grammar, the brackets indicate that specification is optional. As described in the above syntax, ***command*** name is the only required element

- **varlist** refers to a list of variables, e.g. *mpg weight length price*,
- **command** refers to the specific command you want to run,
- **exp** refers to a logical expression,
- **range** refers to a range of line numbers
- **options**, will depend on the command in question. The options must be specified at the end the command line, after a comma separator.

The by syntax:

The `[by varlist:]` formulation is optional and specifies that the command is to be repeated for each variable in the variable list. Not all commands can use this formulation.

- NB! To use `by`, the data should be sorted by the `by` -variables (sort varname(s)) or use `bysort`,
- When `by` is used in front of a command - `by` as a prefix - the command is repeated for group of observations for which the values of the variables in varlist are the same,
- When `by` is used after a command with a comma – `by()` as an option – informs the command what groups to use: `command varname [if exp] [in range] , by (groupvar) [options]`
- Some commands does not allow by in front of the command or as an option.

Examples:

- egen avprice1 = mean(price), by(rep78)
- bysort rep78: egen avprice2 = mean(price)
- bysort rep78: gen avprice=sum(price)/_N
- by rep78: gen avprice3=avprice[_N]

Note: _n and _N are useful in many difficult calculations.

Logical expressions:

If you decide to use the optional `[if exp]` specification you must use a special syntax for logical expressions.

- `==` equals to
- `!=` not equal to
- `>=` larger than or equal to, etc.
- `&` and `|` or

Examples:

- `tabulate make rep78 if foreign= =1`
- `tabulate make rep78 if foreign= =1&price<4000`
- `tabulate make rep78 if foreign= =1| price<4000`
- `list if make=="VW Rabbit"`
- `list make if rep78==3 & price !=.`
- `list displacement in 3`

Range:

- 1 gives observation 1
- -1 gives the last observation
- 1/5 gives the first five observations.
- -5/-1 gives the last 5 observations

  ###Options:

- The options are specific to each command. To look at the options for a command write –`help` command – and then the options will be listed.

Num(ber)lists:

Often you will find reference to numlist in the STATA syntax description. Numlist is simply a sequence of numbers, which can be specified in various ways. To get an overview of different ways to specify numlists, type help numlist. (NB! Better not to use commas in numlist as commas are not always allowed). Examples:

- 1 2 to 4 means four numbers, 1, 2, 3, 4;
- 10 15 to 30 means five numbers 10, 15, 20, 25, 30;
- 1/3 three numbers 1, 2, 3, (3/1 – the same in reverse order);

- 4 3:1 means the same as 4 3 to 1;
- -1/2 means four numbers -1, 0, 1, 2
- 2(2)10 means the sequence 2 4 6 8 10;
- -1(.5)2.5 means the numbers -1, -.5, 0, .5, 1, 1.5, 2, 2.5

1.3 Getting Help

- Stata has excellent online help (Statalist, etc.)
- To obtain help on a command (or function) type help command_name, which displays the help on a separate window called the Viewer
- You can also type chelp command_name , which shows the help on the Results window; but this is not recommended.
- If you don't know the name of the command you need, you can search for it. Type help search to learn more
- The help command reverts to a search if the argument is not recognized as a command. Try help Student's t . This will list all Stata commands and functions related to the t distribution. Among the list of "Stat functions" you will see t() for the distribution function and ttail() for right-tail probabilities.
- One of the nicest features of Stata is that, starting with version 11, all the documentation is available in PDF files. Moreover, these files are linked from the online help, so you can jump directly to the relevant section of the manual. To learn more about the help system type help help.

```
.     display 2+2
4

.
.     display 2*ttail(20, 2.1)
.04861759

.     display 5+5
10
```

1.3 Examples:

1.3.1 Loading data in stata

commands: `use`, `webuse`, `sysuse` and `import`

```
. *help use
.
. * Use dataset  a dataset from my folder where it is located
.
.  use "C:\Users\ksu\Documents\Math Review\auto.dta", clear
(1978 Automobile Data)
.
```

```
.  /* use dataset auto from stata memory*/
.
.  sysuse auto
(1978 Automobile Data)
.
.  // use auto dataset from the internet
.
.
.  webuse lifeexp, clear
(Life expectancy, 1998)
.
.
.  *or
.   use  "http://www.stata-press.com/data/r15/lifeexp", clear
(Life expectancy, 1998)
.
.  // Load a dataset from excel file
.
.   import excel "C:\Users\ksu\Documents\Math Review\lifeexp.xlsx", sheet("Shee
> t1") firstrow clear
.
.   *Type help import to learn the wide range of data files that can be importe
> d in stata.
.
```

1.3.2 Differents use of `by` prefix

```
sysuse auto

egen avprice1 = mean(price), by(rep78)

bysort rep78: egen avprice2 = mean(price)

bysort rep78: gen avprice=sum(price)/_N

by rep78: gen avprice3=avprice[_N]
```

2. Macros in Stata: `local` and `global`

A great deal of data management and statistical analysis involves doing the same task multiples times. You create and label many variables, fit a sequence of models, an run multiple tests. You create and label many variables, fit a sequence of models, and run multiple tests. By automating these tasks, you can save time and prevent errors, which are fundamental to an effective workflow. In this part, I discuss four tools for automation in stata.

- `Macros`: `Macros` are simply abreviations for a string of characters or a number. These abbreviations are amazingly useful!

- `Saved Results`: many Stata commands save their results in memory. This information can be retrieved and used to automate your work.
- `Loops`: Loops are ways to repeat a group of commands. By combining macros with loops, you can speed up tasks ranging from creating variables to fitting models.
- `Ado-files`: Ado-files let you write your own command to costomize Stata, automate your workflow, and speed up routines tasks.

2.1 Macros

Macros are the simplest tool for automating your work. A macro assigns a string of text or a number to an abbreviation. Here is a simple example. I want to fit the model

```
regress y var1 var2 var3
```

I can create the macro rhs with the names of the independent or right-hand-side variables:

```
local rhs "var1 var2 var3"
```

Then I can write the `regress` command as

regress y ‘rhs’

where the ‘and’ indicate that I want to insert the contents of the macro rhs. The command **regress y ‘rhs’** , works exactly the same as **regress y var1 var2 var3**. In the examples that follow, I show you many ways to use macros. For a more technical discussion see [P] macro.

2.1.1 Local macros

Local macros that contains a string of characters are defined as

local *local_name* “string”

For example,

```
local rhs "var1 var2 var3"
```

A local macro can also be equal to a numerical expression:

local *local_name* = expression

For example,

```
local ncases = 198
```

The content of a macro is inserted into your do-file or ado-file by entering ‘local-name’. For example, to print the contents of the local rhs, type:

```
local rhs "var1 var2 var3"

display "The local macro rhs contains: `rhs'"
```

2.1.2 Global macros

Global macros are defined much like local macros:

global global_name "string"

global global_name = expression

For example,

global rhs "var1 var2 var3"

global ncases = 198

The content of a macro is inserted into your do-file or ado-file by entering '$global_name'. For example :

global rhs "var1 var2 var3"

display "The global macro rhs contains: $rhs"

2.1.3 Specifying groups of variables and nested models

Macros can hold the names of variables that you are analyzing. Suppose that I want summary statistics and estimate for a linear regression of `lfp` on `k5, k618, age, wc, hc, lwg, inc`, and inc. Without macrosl I enter the commands like this :

```
summarize lfp k5 k618 age wc hc lwg inc

regress     lfp k5 k618 age wc hc lwg inc
```

If I change the variables, say , deleting *hc* and adding *agesquared*, I need to change both commands:

```
summarize lfp k5 k618 age wc agesquared  lwg inc

regress     lfp k5 k618 age wc agesquared lwg inc
```

Alternatively, I can define a macro with the variable names:

local myvars lfp k5 k618 age wc hc lwg inc

Then I compute the statistics like this :

`summarize` 'myvars'

`mean` 'myvars'

“‘

The idea of using a macro to hold variable names can be extended by using different macros for different groups of variables (e.g. demographic variables, health variables). These macros can be combined to specify a sequence of nested models. First, I create macrs; for four groups of independent variables:

```
. local set1_age   "age agesquared"

. local set2_educ  "wc hc"

. local set3_kids  "k5 k618"

. local set4_money "lwg inc"
```

To check that a local is correct, I display the content. For example,

```
.     display " set3_kids: `set3_kids´"
 set3_kids: k5 k618
```

Next I specify four nested models. The first model includes only the first set of variables and is specified as

```
. local model_1 "`set1_age´"
```

The macro *model_2* combines the content of the local *model_1* with the variable in *local set2_educ* :

```
. local model_2 "`model_1´ `set2_educ´"
```

The next two models are specified the same way:

```
. local model_3 "`model_2´ `set3_kids´"

. local model_4 "`model_3´ `set4_money´"
```

Next I check the variables in each model:

```
. display "model_1: `model_1´"
model_1: age agesquared

. display "model_2: `model_2´"
model_2: age agesquared wc hc

. display "model_3: `model_3´"
model_3: age agesquared wc hc k5 k618

. display "model_4: `model_4´"
model_4: age agesquared wc hc k5 k618 lwg inc
```

Using these locals, I estimate a series of logits:

```
. qui use wf-macros, clear

.
. qui logit lfp `model_1´

.
. qui logit lfp `model_2´

.
. qui logit lfp `model_3´

.
. qui logit lfp `model_4´

.
```

‘qui’ performs command but suppress terminal output.

There are several advantages to using locals to specify models. First, when specifying complex models, it is easy to make a mistake.

Second, locals make it easy to revise model specifications. Even if I am successful in initially defining a set of models by typing each variable name for each model, errors creep in when 1 change the models. For example, suppose that I do not need a quadratic term for age. Using locals, I need to make only one change:

```
. local set1_age   "age agesquared"
```

This change is automatically applied to the specifications of all models:

```
.
. local model_1 "`set1_age´"
. local model_2 "`model_1´ `set2_educ´"
. local model_3 "`model_2´ `set3_kids´"
. local model_4 "`model_3´ `set4_money´"
```

Using locals with graph

The options for the graph command can be very complicated. For example, here is a graph comparing the probability of tenure by the number of published articlces for male and female biochemists (** file wf4-macros-graph.do**):

```
.
. use wf-macros, clear
(Workflow data for illustrating macros \ 2008-04-02)
.
. //  complicated graph without using macros
.
. graph twoway ///
>   (connected pr_women articles, lpattern(solid) lwidth(medthick) ///
>   lcolor(black) msymbol(i)) ///
>   (connected pr_men articles,   lpattern(dash)  lwidth(medthick) ///
>   lcolor(black) msymbol(i)) ///
>   , ylabel(0(.2)1., grid glwidth(medium) glpattern(dash)) xlabel(0(10)50) ///
> ytitle("Probability of tenure") ///
> legend(pos(11) order(2 1) ring(0) cols(1))
. graph export wf4-macros-graph.eps, replace
(file wf4-macros-graph.eps written in EPS format)
.
.
```

Macros make it simpler to specify the options, to see which options are used, and to revise them. For this example, I can create macros that specify the line options for men and for women, the grid options, and the options for the legend:

```
. //  options defined in locals
. * line characteristics
. local opt_linF   "lpattern(solid) lwidth(medthick) lcolor(black) msymbol(i)"
. local opt_linM   "lpattern(dash)  lwidth(medthick) lcolor(black) msymbol(i)"
```

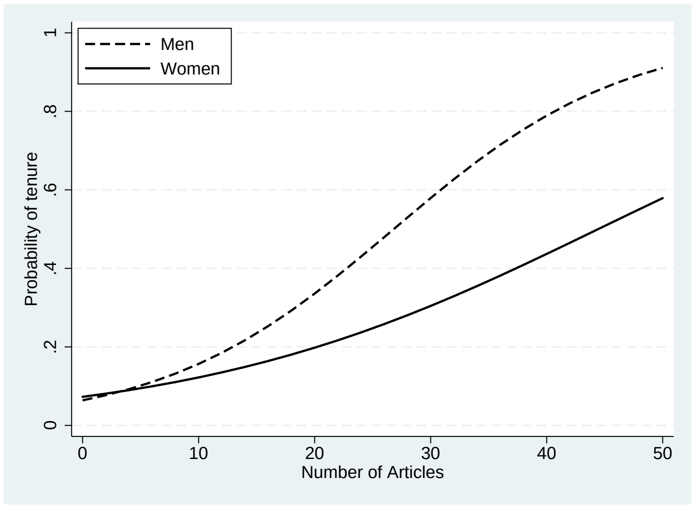


Figure 2: Probability of tenure by the number of published articlces for male and female biochemists

```
. * grid options
. local opt_ygrid  "grid glwidth(medium) glpattern(dash)"
. * legend options
. local opt_legend "pos(11) order(2 1) ring(0) cols(1)"
```

Using these macros, I create a graph command that I find easier to read:

```
. graph twoway ///
>   (connected pr_women articles,     `opt_linF´)   ///
>   (connected pr_men   articles,     `opt_linM´)   ///
>   , xlabel(0(10)50) ylabel(0(.2)1., `opt_ygrid´)  ///
> ytitle("Probability of tenure")               ///
> legend(`opt_legend´)
```

Moreover, if I have a series of similar graphs, I can use the same locals to specify options for all the graphs. If I want to change something, I only have to change the macros, not each graph command. For example, if I decide to use colored line to distinguish between men and women, I change the macros containing line options:

```
. * line characteristics
. local opt_linF   "lpattern(solid) lwidth(medthick) lcolor(green) msymbol(i)"
. local opt_linM   "lpattern(dash)  lwidth(medthick) lcolor(purple) msymbol(i)"
. * grid options
. local opt_ygrid  "grid glwidth(medium) glpattern(dash)"
. * legend options
. local opt_legend "pos(11) order(2 1) ring(0) cols(1)"
.
```

With these changes I can use the same graph twoway command as before.

```
. graph twoway ///
>   (connected pr_women articles,     `opt_linF´)   ///
>   (connected pr_men   articles,     `opt_linM´)   ///
>   , xlabel(0(10)50) ylabel(0(.2)1., `opt_ygrid´)  ///
> ytitle("Probability of tenure")               ///
> legend(`opt_legend´)

. graph export wf4-macros-graph2.eps, replace
(file wf4-macros-graph2.eps written in EPS format)
```

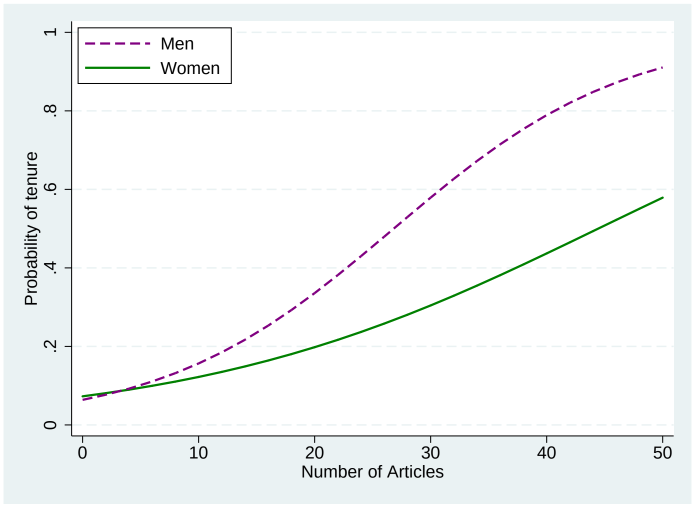


Figure 3: Probability of tenure by the number of published articlces for male and female biochemists

2.2 Information returned by Stata commands

When writing do-files you never want to type a number if Stata can provide lhe number for you. Fortunately, Stata can provide just about any number you need. To understand what this means, consider a simple example where I mean center the variable age. I could do this by first cornputing the mean (**file wf4-returned.do**):

```
.
. use wf-lfp, clear
(Workflow data on labor force participation \ 2008-04-02)

. summarize age

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         age |       753    42.53785    8.072574         30         60

. generate age_mean = age - 42.53785

. label var age_mean "age - mean(age)"

. summarize age_mean

    Variable |       Obs        Mean    Std. Dev.       Min        Max
```

```
-------------+--------------------------------------------------------
    age_mean |       753   -1.49e-06    8.072574  -12.53785   17.46215
.
```

Next I use the mean from summarize in the generate command:

```
.
. use wf-lfp, clear
(Workflow data on labor force participation \ 2008-04-02)
. summarize age
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         age |       753    42.53785    8.072574         30         60
. generate age_mean = age - 42.53785
. label var age_mean "age - mean(age)"
.
```

The average of the new variable is very close to 0 as it should be (within .000001):

```
.
. summarize age_mean
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
    age_mean |       753   -1.49e-06    8.072574  -12.53785   17.46215
.
```

I can do the same thing without typing the mean. The summarize command both sends output to the Results window and saves this information in memory. In Stata's terminology, summarize returns this information. To see the information returned by the last command, I use the return list command. For example,

```
. summarize age
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         age |       753    42.53785    8.072574         30         60
. return list
scalars:
                  r(N) =  753
              r(sum_w) =  753
               r(mean) =  42.53784860557769
                r(Var) =  65.16645121641095
                 r(sd) =  8.072574014303674
                r(min) =  30
                r(max) =  60
                r(sum) =  32031
```

The mean is returned to a scalar named `r(mean)`, I use this value to subtract the mean from age:

```
. generate age_meanV2 = age - r(mean)
```

When I compare the two mean-centered variables, I find that the variable created using `r(mean)` is slightly closer to zero:

```
. summarize age_mean age_meanV2
```

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
    age_mean |       753   -1.49e-06    8.072574  -12.53785   17.46215
  age_meanV2 |       753    6.29e-08    8.072574  -12.53785   17.46215
```

I could get even closer to zero by creating a variable using double precision:

```
. summarize age
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         age |       753    42.53785    8.072574         30         60
. generate double age_meanV3 = age - r(mean)
. label var age_meanV3 "age - mean(age) using double precision"
. summarize age_mean age_meanV2 age_meanV3
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
    age_mean |       753   -1.49e-06    8.072574  -12.53785   17.46215
  age_meanV2 |       753    6.29e-08    8.072574  -12.53785   17.46215
  age_meanV3 |       753    3.14e-15    8.072574  -12.53785   17.46215
.
```

This example illustrates the first reason why you never want to enter a number by hand if the information is stored in memory. Values are returned with more numerical precision than shown in the output from the Results window. Second, using returned results prevents errors when typing a number. Finally, using a returned value is more robust. If you type the mean based on the output from summarize and later change the sample being analyzed, it is easy to forget to change the generate command where you typed the mean. Using `r(mean)` automatically inserts the correct quantity.

Most Stata commands that compute numerical quantities return those quantities and often return additional information that is not in the output. To look at the returned results from commands that are not fitting a model use `return list`. For estimation commands use `ereturn list`. To find out what each return contains, enter help `command-name` and look at section on saved results:

2.3 Loops: foreach

Loops let you execute a group of commands multiple lines. Here is a simple example that illustrates the key features of loops. I have a four-category ordinal variable *y* with values from 1 to 4. I want to create the binary variables *y_lt2*, *y_lt3*, and *y_lt4* that equal 1 if *y* is less than indicated value, else 0. I can create the variables with three generate commands (**file: wf4-loops. do**):

```
. use wf-loops, clear
(Workflow data for illustrating loops \ 2008-04-02)
. * without loop
. generate y_lt2 = y<2 if !missing(y)
. generate y_lt3 = y<3 if !missing(y)
. generate y_lt4 = y<4 if !missing(y)
```

where the `if` condition `!missing(y)` selects cases where *y* is not missing. I could create the same generate command with a foreach loop:

```
. * First drop these generated variables I can create them another way
. drop y_lt2 y_lt3 y_lt4

.
. * with foreach loop
. foreach cutpt in 2 3 4 {
  2. generate y_lt`cutpt´ = y<`cutpt´ if !missing(y)
  3. }
```

Let's look at each part of this loop. *Line 1* starts the loop with the foreach command. *cutpt* is the name I chose for a macro to hold the cutpoint used to dichotomize *y*. Each time through the loop, the value of cutpt changes in signals the start of a list of values that will be assigned in sequence to the local cutpt. The numbers 2 3 4 are the values to be assigned to cutpt. `{` indicates that the list has ended. *Line 2* is the command that I want to run multiple times. Notice that it uses the `macro` *cutpt* that was created in *line 1*. Line 3 ends the foreach loop. Here is what happens when the loop is executed. The first time through foreach the local cutpt is assigned the first value in the list. This is equivalent to the command `local cutpt "2"` Next the generate command is run, where , cutpt, is replaced by the value assigned to cutpt. The first time through the loop, line 2 is evaluated as

`generate y_lt2 = y<2 if !missing(y)`

Next the closing brace `}` is encountered: which sends us back to the foreach command in *line 1*. In the second pass, foreach assigns cutpt to the second value in the list, which means that the generate command is evaluated as:

`generate y_lt3 = y<3 if !missing(y)`

This continues once more, assigning cutpt to 4. When the foreach loop ends, three variables have been generated.

```
. //  estimate models using a loop
.
. local rhs "yr89 male white age ed prst"
```

To run the logits, I could use the commands

logit y_lt2 'rhs'

logit y_lt3 'rhs'

logit y_lt4 'rhs'

Or I could do the same thing with a loop:

```
. foreach lhs in y_lt2 y_lt3 y_lt4 {
  2.    qui logit `lhs´ `rhs´
  3. }
```

Using foreach to fit three models is probably more trouble than it is worth. Suppose that I also want to compute the frequency distribution of the dependent variable and fit a probit model. I can add two lines to the loop:

```
. foreach lhs in y_lt2 y_lt3 y_lt4 {
  2.    qui tabulate `lhs´
  3.    qui logit    `lhs´ `rhs´
  4.    qui probit   `lhs´ `rhs´
  5. }
```

If I want to change a command, say, adding the missing option to tabulate, I have to make the change in only one place and it applies to all three outcomes. I hope this simple example gives you some ideas a.bout how useful loops can be. There are two other types of loops: *forvalue* and *while*. They are as useful as *foreach* loop. or further information1 use help or check [P] foreach, [P] forvalues and [P] while.

3. Use and Best Practice of organizing stata files

3.1. A simple example of a dofile structure

(See File **Example__Of__Organizing__a__Dofile.do**)

3.2. Organizing a Stata Project

(In Class using projmanager – Stata Project Manager)