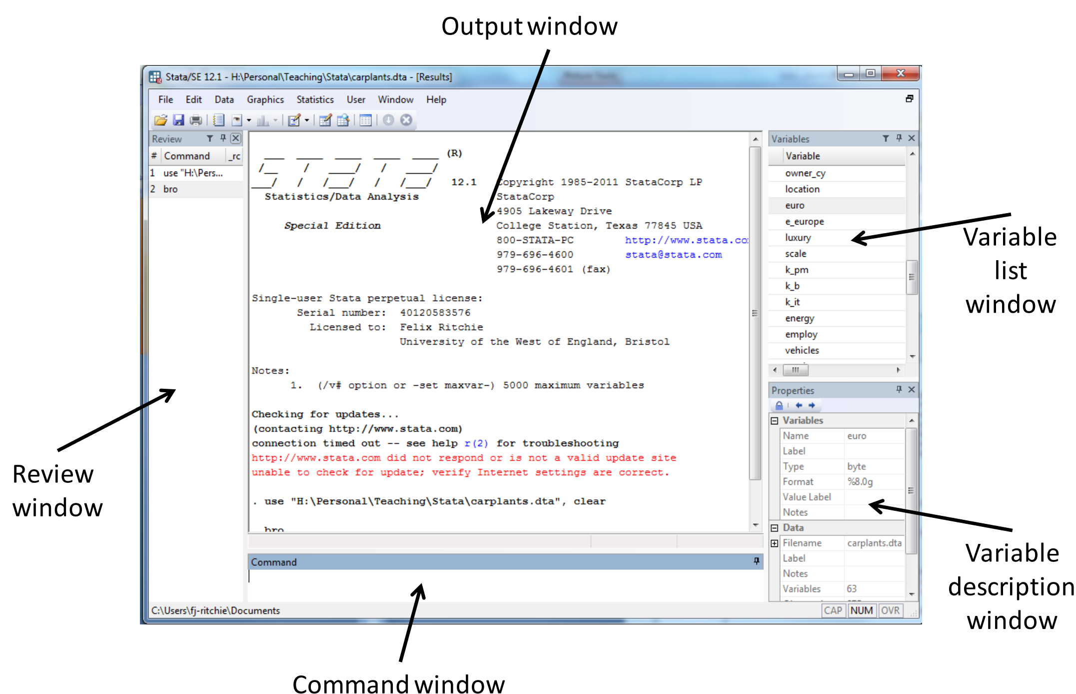
STATA Basics

This guide assumes some familiarity with concepts such as missing values, variables, observations, and datasets, and with simple statistical analysis.

1. The interface

1.1 The windows

When STATA is started five windows appear on the screen:



- The main window is the *output window* or *results window*, labelled *Stata Results*. This is where the effects of all operations are displayed.
- Below the results window is the *command window.* A command typed in here is executed immediately and the effects appear in the results window.
- To the left of the results window is the *review window*. This lists all the commands entered in the command window.
- To the right of the results window is the *variable list window*. This lists all the variables available in the dataset.
- Finally, the bottom right-hand window (the *variable description window*) shows more detailed information about any variable you have selected in the variable list window.

These windows can be resized by hovering over the edges of the windows until the arrow cursor turns into a double arrow, and then "pulling" the edge to the right size. Stata will remember these settings.

1.2 Using the windows

Load up one of the sample data files. From the top menus choose

file => open => auto.dta

where *auto.dta* is a Stata sample file found in the directory where Stata is installed.

Assuming Stata has been installed in C:\Program Files\Stata, the command

```
use "C:\Program Files\stata\auto.dta" , clear
```

has appeared in the results window along with a description of the file found ("1978 automobile data").

The command has also been placed in the review window. To run this command again, double-click on it; or use the "page-up" and "page-down" keys to move through the command list.

To enter a command, type in

```
summarize
```

and press the "return" key (↵). This gives a summary of the characteristics of the data sets. All of the Stata commands can be abbreviated, as long as those are not ambiguous. For example:

```
summarize
summa
sum
su
s
```

The first four are okay, but the last one could be any command beginning with "s".

To see the data for just a few variables, (for example, make and price) enter

```
summarize make price
```

Instead of typing in the variables names, click on the names in the variable windows. This will insert them into the command window.

Enter

```
list make price
```

Note that the display stops when it has a screen full. To continue displaying, press the space bar. To abandon the display, press "q" (for **quit**).

Although Stata displays one screen at a time, it keeps a record of previous outputs. Using the scroll bar on the right of the results window, it can be seen that earlier results are still visible. However, this is not an infinite record. Run the **list** command several times, then note how the results window drops the oldest output.

1.3 Logging results

The results window is dynamic and cannot be saved. To keep a permanent copy of results a *log file* needs to be created.

To create a log file, click on the log button or use the command

file => log => begin

This will open a dialogue box to find a log file. Note that Stata creates files with an "SMCL" extension. This is Stata's format for displaying log files, and it is not easily usable outside Stata.

When the file "test.smcl" (for example) is created, Stata records the command

```
log using "C:\stata\data\test.smcl"
```

as having been executed. This command could have been typed in as well as being a menu function. In Stata, almost all the menu commands have an equivalent that could be typed in. This becomes increasingly relevant as programs rather than interactive commands are used.

To close the log, enter

```
log close
```

Again, this could have been done from the menu. To review the log, on the menus choose

file => log => view…

and select the name of the log file. Unlike the results window, log files are not limited in size, and they will record all the output while the log file is open.

1.4 Amending the environment

To change the environment, the **set** command is used. This sets system parameters. One particularly useful command is:

```
set more off
```

This controls whether Stata stops at the end of each screen full of output and waits for a key to be pressed. If "more" is switched off, then Stata just displays results without waiting for user input.

Stata will not let certain things happen when it has a (changed) dataset loaded – for example, loading another dataset, or changing the amount of memory. To clear Stata, enter

```
clear
```

which removes all data from memory.

1.5 The working directory

Stata uses the concept of a "working directory" where files without a full path name are read from and stored. The current working directory can be seen on the bottom left-hand corner of the screen, or by entering

```
cd
```

which stands for “change directory”. To change to a particular directory, place the path after the *cd*:

```
cd “data”
cd “c:\stata\data”
```

The former moves to a directory “data” which is a subdirectory of the current working directory (that is, it is a *relative* path). The latter moves to the directory C:\Stata\Data irrespective of current location (in other words, it is an *absolute* path). The quote marks are not necessary unless there are spaces or other odd characters in the directory name, but they make the code more readable.

1.6 The help system

The Stata help system has three main paths to find information. Each is accessed by clicking on the “help” button. In addition, typing one of these

```
help item
search item
```

will display help on the item in the results window. However, this is rather less helpful than the pop-up windows which arise from using the menu command.

1.6.1 Help directory

This is largely a copy of the user guide, with help organised by functionality. This is the best place to explore the possibilities of Stata.

help => contents

1.6.2 Help search

This option allows a free-text word search through all the Stata help files, and will return all the help pages containing the search term. This sometimes seems to miss terms, and Stata is not good with synonyms, so if a term is not found it’s usually worth trying to find an alternative description.

help => search…

This is the best way to find information if a particular concept or tool is required.

1.6.3 Help on a specific command

This is the easiest route for finding information on a command or its parameters – if that command is known. To use it choose

help => Stata command…

Note however that the command help does not recognise abbreviations (eg “sum” for “summarize”).

1.7 Leaving Stata

To leave Stata type in

```
exit
```

or close the Stata window. If the data has been edited, Stata will prompt before exiting.

Section 1 Exercises

E1.1 Load up one of the sample data files.

E1.2 Experiment with **list**, **summarize** and using the variable window to add variables to commands.

E1.3 Use both the keys and the review window to rerun earlier commands

E1.4 Open a log file. Close and re-open it. Note the options that are available for appending and suspending.

E1.5 On the “file” menu, look for the command to view saved log files. Try the “append” and “suspend” options and view the results.

E1.6 Use the help system to

a. display text and do calculations interactively (use the “contents” option)

b. look for analysis of covariance operations (use the “search” open)

c. see what the options are for the estimation command **est** (use the “Stata command” option

2. The essential commands

For this section , use the **cd** command to change to the tutorial directory (which will be given out in class; for argument here, use C:\temp\tutorial):

```
cd "c:\temp\tutorial"
```

This should contain a copy of the sample *.dta* files found in the Stata home directory. If not, copy them over.

2.1 The command structure, and some warnings

Stata commands have the form

by …:	command	command parameters	if …	range …	, command options
selection criteria			*Select only some observations*	*selection criteria for 'if'*	

The *command* is necessary but everything else can be optional. The selection criteria are dealt with in the next section.

Some words of warning:

- Stata is not very **consistent** in the syntax of its commands. Moreover, some commands are extended with the syntax and options of other commands. There is no alternative but to learn the commands. Thinking in terms of the above blocks may help to clarify the command structure
- Stata is **case-sensitive**
- Stata allows **abbreviations** of commands, variables, and options where these are unambiguous. Hence, *su*, *sum*, *summar* are all valid abbreviations for "summarize" but *s* (ambiguous) and *summary* (not an abbreviation) are not.
- The **help system** does not consistently recognise abbreviations
- Stata **error messages** are not the most enlightening.

Taken together, these mean there is a lot of scope for problems, particularly in the syntax of commands. The good news is that these are relatively easy to identify.

2.2 Loading data

To open a data file, either use the menus

file => open…

or type in

```
use "filename" , clear
```

The **clear** option tells Stata to clear anything else out of memory. The quote marks are optional unless you have unusual characters in the file name like spaces, but they make the code more readable.

Stata datasets have the extension *.dta*. This does not need to be specified as Stata assumes it.

Stata only works on one dataset at a time. Therefore to work with more than one dataset it is necessary to **merge** or **append** them. To append

```
append using "filename"
```

This appends a file on disk to the one currently in memory. If there are variables in one file that don't exist in the other then the variables will be filled in with missing values. For example, observe the changes in the variable window when running the following code

```
use "auto"
count
append using "machine"
```

Now run

```
count
sum
```

count gives the number of observations in the file and **sum** is short for **summarize**. The first shows that there are 131 observations in the combined files. The second shows that there are 74 non-missing values for the variables which originate from the "auto" file and 57 non-missing values for variables which appeared in "machine" *(how do we know that the variable 'rep78' has 5 missing values?)*. The two datasets have formed two non-intersecting sets with 131 observations. **append** will always give the new number of observations as the sum of the observations in each of the two source datasets.

merge works in a similar fashion except that at least one variable is specified as a key field. If the key field is the same in both files, than that observation takes the variables from both files; otherwise a new observation is created. If however, there are several variables (which are not key fields) which have the same name, then the action that Stata takes depends upon the options set.

merge is a complex command and is not necessary for this course. However the exercises at the end of the section illustrate some of the issues.

2.3 Describing data

For this section load the files on Census housing figures:

```
use "hsng" , clear
```

The key descriptive commands are:

```
describe
summarize
tabulate
count
```

which can be shortened sensibly to

```
desc
sum
tab
```

describe gives general information about the dataset including number of observations and variables, dataset size, and details about the variables. It also notes whether the dataset is sorted or not.

In the *hsng* dataset, note the “storage type” column of **describe** reports that “division” and “region” are both integer variables **(int**). However, when working with these variables, they will usually be displayed with helpful names such as “NE” or “SW” rather than “region 1” or “region 2”. The way this is done is to associate a **label** with each integer value; Stata uses this whenever it makes sense to display this rather than the integer value. This information can be seen in the “value label” column. This course does not go into creating labels, but remember that these two fields are integers, not text fields – they only display as text.

To see how this works, look at the results fo these commands

```
tab division
tab division , nolabel
```

By default, Stata will use helpful labels. The second command tells Stata that we want to see the underlying coding.

summarize gives information about the characteristics of variables, including the range of values encountered. The column *obs* shows the number of non-missing values. Note however that text values show up as missing values. This is because Stata uses a special value to indicate missing values which looks like a piece of text, not a number. As it’s meaningless to talk about the median value for the name of a state, the sum command ignores text variables.

Both of these commands cane be augmented by variable names:

```
desc state pop
sum division region
```

tabulate can create one- or two-way tabulations of data. Try:

```
tab region
tab division region
```

It only works where the variables can be broken down into a small enough categories. A one-way tabulation allows for a greater range of values. Large values are allowed on some-two-way tabulations, depending upon how the data is displayed. Look at the effect of:

```
tab pop region
tab region pop
```

count simply sums up all the observations, including missing values.

2.4 Displaying variables

To examine the values of variables, use **list**:

```
list state pop
```

list on its own displays all variables for all observations, but the display format depends on how variables can be fitted into a window.

For more leisurely viewing, use the **browse** option

```
browse
```

which allows all variables to be examined in a table. Specific variables can be examined by adding variable names:

```
browse state region division pop popgrow
```

Note that some commands cannot be entered while the browse window is open.

For the value label fields "division" and "region", **browse** displays the text label. However, the underlying value is reported at the top of the screen. This is one way to identify the values underlying the displayed text, which will become important later.

2.5 Creating variables

The command **generate** can be used to create new variables:

```
generate rent_value_ratio = rent/hsngval
```

The usual arithmetic operations can be carried out. If the variable already exists, an error message will be sent.

A related command, **replace**, can be used to change existing variables:

```
replace rent_value_ratio = rent_value_ratio*12*100
```

For reasons best known to Stata, there is an "extension to generate" **egen** which has a wider range of functions not available in **generate**:

```
egen tot_pop = sum(pop)
generate pop_prop = pop/tot_pop
```

When using **generate** to create variables which are not floating-point numbers (that is, standard decimals), the variable type needs to be specified:

```
gen int pop_pp = pop_prop*100
gen str2 state_2l = substr(state, 1, 2)
```

The first creates an integer; the second, a string (a text field) two characters long from characters 1-2 of the "state" variable.

Stata has certain *system variables* which contain information about the data. Two of the most useful (although not in a whole-dataset context) are

_N The total number of observations

_n The number of the current observation, as the dataset is currently sorted

To see the effect, try

```
gen big_N = _N
gen small_N = _n
list state big_N small_N
```

They can be used to index variables:

```
gen prev_pop = pop[_n-1]
list pop prev_pop
```

These become more important later when dealing with subsets of the dataset. Other system variables include standard errors and coefficients generated by regressions.

2.6 Deleting variables

Variables can be removed by using the **drop** command:

```
drop state_21
```

will remove the variable permanently. Several variables can be removed at once.

Alternatively, Stata can be instructed to **keep** only certain variables:

```
keep region division pop popden
```

will drop all variables apart from the ones listed.

2.7 Sorting data

Sorting date is straightforward using **sort**. Datasets can be sorted on any number of variables.

```
sort region
sort pop
sort region pop
```

Once a dataset is sorted it remains in that order until Stata receives another sort command. The sort order is also remembered if the dataset is saved.

2.8 Saving data

To save a file, use the command **save**:

```
save filename
save filename , replace
```

If a file with that file name already exists than the first command fails. The latter will overwrite the file but will not ask before doing so. Which of these options is more appropriate depends upon the particular circumstances.

Section 2 Exercises

E2.1. Load up the file auto.dta. Use **sum**, **describe**, **browse** and **list** to familiarize yourself with the data.

E2.2. Use **generate** to create a dummy variable from the "foreign" variable. Create it as an "integer" data type

E2.3. Drop the original foreign variable

E2.4. Rename your new variable as "foreign". Use the **rename** command (see the Help system)

E2.5. Use **generate** and **egen** to create a new variable reflecting the average price of a car

3. Using data samples

Reload the datafile "hsng" for this section, remembering to use the "clear" option.

3.1 "If" clauses

Almost any command can have an *if-clause* attached to it. This is used to work on subsamples:

```
list state pop if pop<1000000
```

For equality tests, a double-equals sign is necessary:

```
gen small_state = pop<1000000
list state pop if small_state==1
```

"True" is recorded as a 1 and "False" as 0. The actual comparison could be omitted in the above example:

```
list state pop if small_state
```

If-clauses can also include many of the functions available for **generate** as long as they apply to individual entries:

```
list state pop if substr(state, 1, 2) == "Al"
```

which selects all the states beginning with the two-letter combination "Al".

In combination with **drop** and **keep**, the if-clause can be used to drop observations, not just variables:

```
drop if small_state
keep if rent>200
```

Stata also has logical operators allowing combinations of statements to be used

```
list state pop region if region == 1 & pop > 1000000
list state pop region if region == 1 | region == 2
```

Note how the comparison with "region" must be made in terms of the underlying integer value, not the displayed text label.

3.2 Creating temporary subsets using "by"

Prefixing commands with **by … :** allows Stata to act as if it were operating on a series of temporary subsamples. It is the creation of these which really make the **generate** and **egen** commands powerful:

```
sort region
by region: egen reg_pop = sum(pop)
```

This creates a variable called reg_pop which contains the total population for each region.

by requires that the dataset be sorted. If not, it can be sorted in the same command:

```
by region, sort: egen reg_pop2 = sum(pop)
```

and some (but not all) commands allow **by** as an additional option which does not sort the dataset:

```
egen reg_pop3 = sum(pop), by(region)
```

by comes into its own when combined with the Stata system variables. Compare the command in section 2.5 with these:

```
by region: gen r_big_N = _N
by region: gen r_small_N = _n
list region state r_big_N r_small_N
by region: list state r_big_N r_small_N
```

3.3 Missing values

Stata uses a special code for missing values. In the user interface, this is represented by a dot, and it can be used in expressions:

```
generate log_pop_growth = ln(popgrow)
replace log_pop_growth = -99 if log_pop_growth ==.
```

In combination with **generate** and **replace** this can be a way to create categorical variables:

```
generate expand = 1 if popgrow>0
replace expand = -1 if popgrow<0
replace expand = 0 if expand ==.
```

Section 3 Exercises

For this exercise, load the file *auto.dta*.

E3.1. List the models of cars which have an MPG over 30

E3.2. Use **generate** to create a new variable from "make" which contains the car manufacturer, not the model (hints: you want a new string variable which takes the text in 'make' up to the first space; **strpos** is a function which finds the position of one string in another; **substr** extracts one string from another**.** Remember to think about the data type in **generate**. What about the Subaru?)

E3.3. Use **by…:** and **egen** to create a variable which contains the average mpg per manufacturer.

E3.4. Use **generate**, **replace** and if-clauses to create a categorical variable which breaks average miles per gallon into x <20 < y <30 < z

E3.5. Save the file for use in the next section.

4. Analysis and graphing

For this section, reload the file *hsng.dta*.

4.1 Regression

Stata has a huge variety of regression commands, some of which are easy to find and some of which are not. All of the relevant panel data regression commands will be dealt with during this course.

The basic regression command is something like

```
regress rent pop hsngval
```

This will produce a standard statistical output. A constant will be included unless explicitly excluded, and called "_cons".

Regression commands can be combined with the other features of Stata encountered so far:

```
sort region
by region: regress rent pop hsngval
```

This pops up some results which don't agree with the pooled model estimated first (why?). A halfway house might be to introduce some dummies for the states. Noting that the region variables are coded as numbers (and only display with useful names – look at the dataset using **describe),** then:

```
gen int reg_1 = region==1
gen int reg_2 = region==2
gen int reg_3 = region==3
gen int reg_4 = region==4
```

Used in the regression (remembering to drop one dummy to avoid perfect collinearity)

```
regress rent pop hsngval reg_2 reg_3 reg_4
```

which seems to imply that the effect of separate regions is spurious as the dummies (which now measure deviation from the effect in region 4) are all insignificant. Maybe this is due to the small numbers in the separate-region estimate - or is it? Note that the slope coefficients in the separated models showed, relatively, far more variation than the intercept terms…

If you are not interested in keeping categories as separate variables,but just want them for the purposes of the regression, you can use the command **xi:** before the regression:

```
xi: regress rent pop hsngval i.region
```

This automatically generates dummy variables from any variable prefixed with "i.". Compare with the previous regress. Note that Stata automatically drops the first dummy to avoid collinearity (hence we used 2,3,4 for comparability in the previous example, not 1,2,3); you can set it to drop the largest category instead, which is often statistically more sensible.

To use the estimated values, the estimated coefficients are stored in a system variable called _b. This can be accessed by adding, in square brackets, the name of the variable whose coefficient is of interest:

```
display _b[pop]
```

and this can be used in further expressions, for example to generate the estimation errors:

```
gen err = rent
  -_b[_cons] -(_b[pop]* pop) -(_b[hsngval]* hsngval)
  -(_b[reg_2]*reg_2)-(_b[reg_3]*reg_3)-(_b[reg_4]*reg_4)
```

Note the above command should all be on one line. You can also access the standard errors for a variable by using **_se[ ]** in the same way.

All the summary statistics about the regression are stored in Stata, until you run another analytical command. Type

```
help regress
```

At the bottom it shows you "saved results". These saved results can be used in commands like any other variable. For example, one of those tells you that e(r2_a) is the adjusted R-squared. Enter

```
display e(r2_a)
```

This should be the same value as in the last regression you ran.

There are also what Stata calls *post-estimation commands*. These allow you to get more information about the predicted values and to carry out hypothesis tests. For example, to get the predicted values and errors, you could do this one of two ways usinghte **predict** command:

```
predict err , residuals
```

or

```
predict predicted_rent , xb
generate err = rent - predicted_rent
```

See **help predict** for details.

If you are running repeated regressions and want to store the results, look at the command **estimates store**. This is very helpful in programming but not covered here.

4.2 Graphs

Graphs in Stata are best done through the menus unless you are writing a program file as they have many options. However, you can copy the menu commands and see how these are put together. For example, enter

```
gen rent_est = rent-err
```

Then on the menus go to

Graphics => twoway plots=>

and select the scatter plot with rent as the X variable and rent_est as the Y variable. You willsee this reported as

```
twoway (scatter rent_est rent, sort)
```

in the command window.

Section 4 Exercises

For this exercise, load the file *auto.dta* saved at the end of the last exercise.

E4.1. Estimate the relationship between miles per gallon and the characteristics of the car.

E4.2. Add dummies to allow for car manufacturers. Does this help the regression?

E4.3. Create dummies for the country of origin. What is the effect now?

E4.4. Calculate the residuals using each of the three methods described.

E4.5. Plot the residuals using the ‘’histogram” function. How do the errors look?

5. Writing programs

Although the Stata command line is adequate for many operations, for more complex tasks (or where the task needs to be repeated, perhaps with slight amendments) a *program* is a more efficient way to write Stata code.

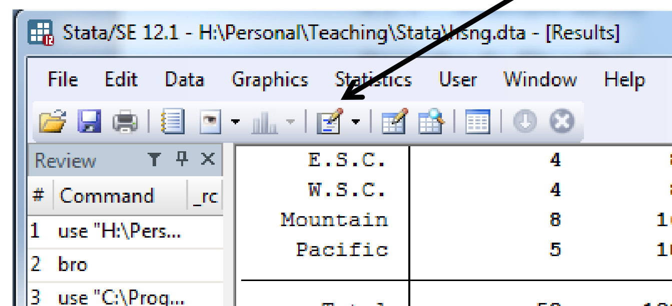
Stata recognises two basic programs: *do-files* and *ado-files*.

5.1 Do-files

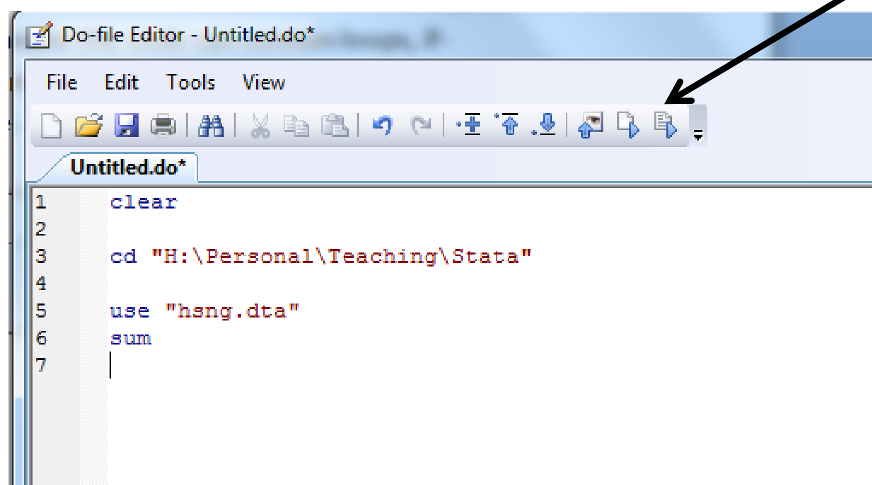
A do-file is a normal text file, which contains Stata code. Stata reads the file in and analyses the code before deciding what to do with it. This means the code can contain loops, if-statements to determine which commands are carried out, temporary variables, and comments to make the whole thing more readable.

5.1.1 A sample do-file

Open the do-file editor.



This will give you a window for the standard text-editing program. Enter some Stata code in just the same way as before, and then click on the “run the do-file” button:



Stata will run the code as if each command had been entered into the command window.

Alternatively, the file can be run by saving the file from the menu:

File => Save As…

then typing in **do** and the file name in the command window (assuming you saved it as "prog1.do":

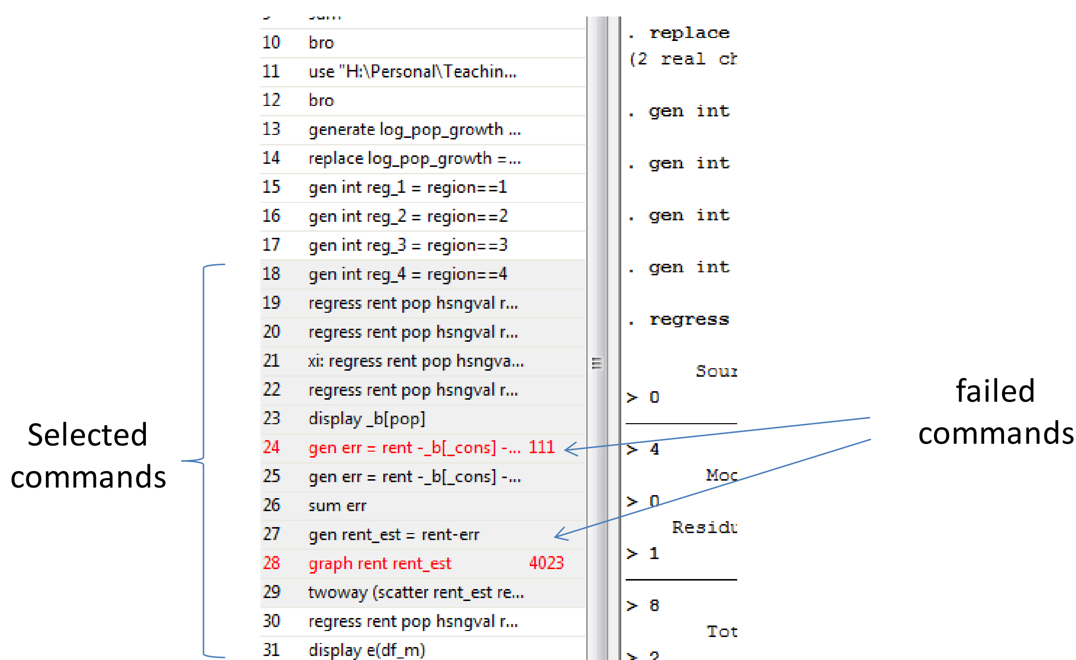
```
do prog1.do
```

Advantages of do-files include

- they can be permanently stored, and so can be run again the next time Stata is opened
- it is simple to make one change and then run a whole sequence of commands again
- there are more commands available in a do-file

5.1.2 Trialling code before using it

Often you'll find yourself trying out a bit of code in the command window before writing it into the do-file. You can scroll through previous commands to copy and select the bits you want, but you can also copy directly from the preview window. Highlight the commands you want, then right-click/copy, and then past into the do-file.



Note that the commands which didn't work are highlighted in red, so you can de-select those before copying by holding down the CTRL-key and clicking.

5.1.3 Adding comments

Comments can either consist of a single asterisk on a line, in which case everything following on the line is treated as a comment

```
* this is a comment
```

or they can consist of matched pairs of /* and */. This allows large blocks of text to be commented out easily:

```
/*
  Program: Simple_sum.do
  Author: FJR  03.8.13
  Loads file and creates a simple summary of data
*/
```

Comments make code much more readable:

```
* set up environment
clear
set more off

* move to correct directory and load data
cd "c:\stata"
use "hsng.dta"

* print quick summary
summ
```

5.1.4 Loops and macros

Stata has several confusingly similar looping commands. The simplest is probably **foreach** which carries out a sequence of operations for every variable in a list:

```
foreach var in 1 2 3 4 {
  sum if region == `var'
}
```

This is equivalent to

```
sum if region == 1
sum if region == 2
sum if region == 3
sum if region == 4
```

Other similar loop commands exist. See the help system for details (work through the examples, not the explanation).

The temporary variable “var” takes on the value of each of the variables in the list in order. Variables such as this, called *macros*, can be used generally in programs. They work by a direct text substitution. For example, these are equivalent:

```
local banana = 1
display `banana'
```

and

```
display 1
```

The keyword **local** tells Stata that the following name is to be substituted with the appropriate value whenever Stata comes across the name.

Macros declared using **local** have the same value throughout the program, unless they are changed by another **local** command. In contrast, macros used to control loops, sometimes called *loop variables* or *loop indexes*, have no value outside the loop. They can be redefined, but not relied upon unless re-set by a **local** command.

Macros require distinct left and right quote marks. On UK keyboards the quote marks are the top-left button (left quote) and below the @ symbol (right quote).

5.1.5 If-statements

If-statements can be used to alter the flow of a program. In the above example,

```
foreach var in 1 2 3 4 {
  if `var’ == 1 {
    display “North-east”
  }
  else {
    display “Somewhere else”
  }
  sum if region == `var'
}
```

alters the action depending upon the current value of the variable “var”. If-statements are commonly used in loop with loop variables (as above) to allow for variations in the data. Other places are where the action is not known in advance but depend upon the course of events. For example:

```
regress rent hsngval pop
if _b[pop]/_se[pop] > 1.96 {
  display "pop is significant at the 90% level"
}
else {
  display "pop is not significant"
```

```
}
```

5.1.6 *A warning about the do-file editor*

For both do-files and ado-files, these can be edited by any text-editor. Use of the Stata do-file editor is not necessary. However, if you do use the Stata editor, note that pressing the "run" button causes Stata to save a temporary copy on disk and run that temporary file – it does not automatically save the "real" file.

This can lead to confusion when using the **do** command and clicking on old commands in the review window. A common mistake is to re-run the temporary file while only the real one has been saved, or vice-versa.

5.2 Ado-files (for advanced users)

Ado-files are similar to do-files except that they contain a self-contain piece of code called by Stata (confusingly) a *program*. This program should have the same name as the ado-file.

When Stata comes across a command it does not recognise, for example

```
show_stuff(pop)
```

it looks for a file called show_stuff.ado containing a program called "show_stuff". If it finds the program, it will load it into memory and the code will be permanently available.

This is how a lot of Stata is implemented, and it means that although it has a huge array of commands the core of Stata is kept relatively small. Only a few key commands are available whenever Stata is running; the rest are loaded up as needed.

It also means that a lot of the core functions in Stata can be edited, as they are just text files. Do not edit any ado-files unless you know you or one of your colleagues has created it; you may be editing standard Stata commands.

Ado-files can be loaded into the Stata environment and run automatically. Each user has a "personal" ado file (like a start-up file) which can be used to set up your basic environment; for example, setting memory on opening the 10Mb instead of Stata's 1Mb default. The exact location of this file depends upon your Stata setup. It may not be accessible in some network implementations.

Section 5 Exercises

E5.1 Copy some of your commands from the review window into a do-file, remembering to leave out the failed commands.

E5.2 Clear the do-file, then repeat the exercise of section 4, but this time type the code into a do-file.