Introduction to Stata

London School of Economics
Michaelmas Term 2008

Alexander C. Lembcke

eMail: a.c.lembcke@lse.ac.uk
Homepage: http://personal.lse.ac.uk/lembcke

This is an updated version of Michal McMahon's Stata notes. He taught this course at the Bank of England (2008) and at the LSE (2006, 2007). It builds on earlier courses given by Martin Stewart (2004) and Holger Breinlich (2005). Any errors are my sole responsibility.

Full Table of contents

Course Outline

This course is run over 5 weeks during this time it is not possible to cover everything – it never is with a program as large and as flexible as Stata. Therefore, I shall endeavour to take you from a position of complete novice (some having never seen the program before), to a position from which you are confident users who, through practice, can become intermediate and onto expert users.

In order to help you, the course is based around practical examples – these examples use macro data but have no economic meaning to them. They are simply there to show you how the program works. There will be some optional exercises, for which data is provided on my website – http://personal.lse.ac.uk/lembcke. These are to be completed in your own time, there should be some time at the end of each meeting where you can play around with Stata yourself and ask specific questions.

The course will follow the layout of this handout and the plan is to cover the following topics.

Week	**Time/Place**	**Activity**
Week 1	We, 17:30 – 19:30 (S169)	Getting started with Stata
Week 2	We, 17:30 – 19:30 (S169)	Database Manipulation and graphs
Week 3	We, 17:30 – 19:30 (S169)	Regression and post-regression analysis
Week 4	We, 17:30 – 19:30 (S169)	Advanced estimation methods in Stata
Week 5	We, 17:30 – 19:30 (S169)	Programming in Stata

I am very flexible about the actual classes, and I am happy to move at the pace desired by the participants. But if there is anything specific that you wish you to ask me, or material that you would like to see covered in greater detail, I am happy to accommodate these requests.

Getting to Know Stata and Getting Started

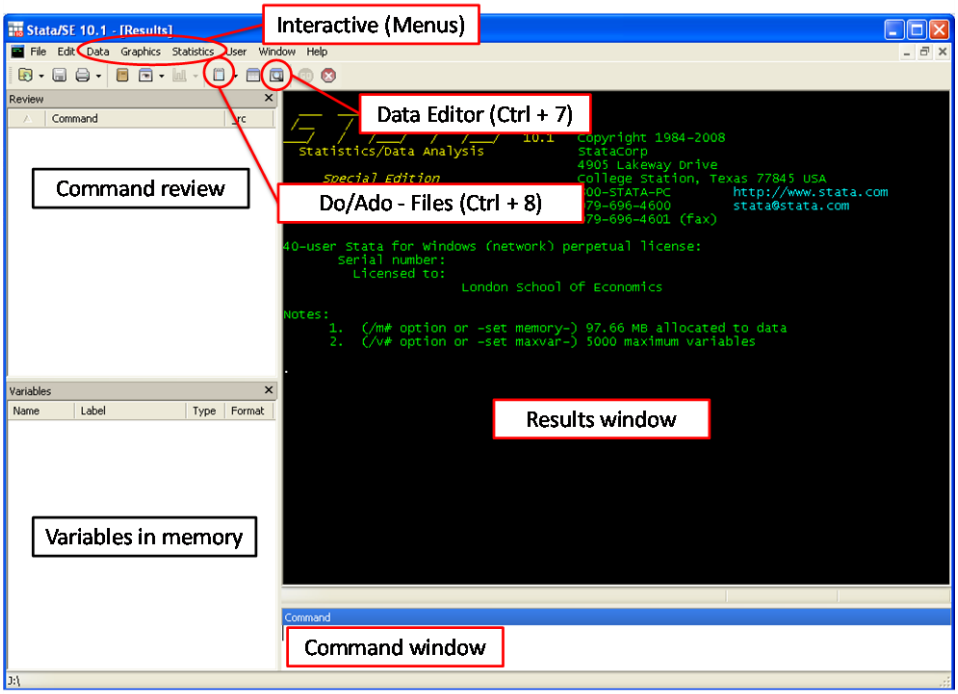
Why Stata?

There are lots of people who use Stata for their applied econometrics work. But there are also numerous people who use other packages (SPSS, Eviews or Microfit for those getting started, RATS/CATS for the time series specialists, or R, Matlab, Gauss, or Fortran for the really hardcore). So the first question that you should ask yourself is why should I use Stata?

Stata is an integrated statistical analysis packaged designed for research professionals. The official website is http://www.stata.com/. Its main strengths are handling and manipulating large data sets (e.g. millions of observations!), and it has ever-growing capabilities for handling panel and time-series regression analysis. The most recent version is Stata 10 and with each version there are improvements in computing speed, capabilities and functionality. It now also has pretty flexible graphics capabilities. It is also constantly being updated or advanced by users with a specific need – this means that even if a particular regression approach is not a standard feature, you can usually find someone on the web who has written a programme to carry-out the analysis and this is easily integrated with your own software.

What Stata looks like

The Stata package is located on a software server and can be started by either going through the Start menu (Start – Programs – Statistics – Stata10) or by double clicking on wsestata.exe in the W:\Stata10 folder. The current version is Stata 10.

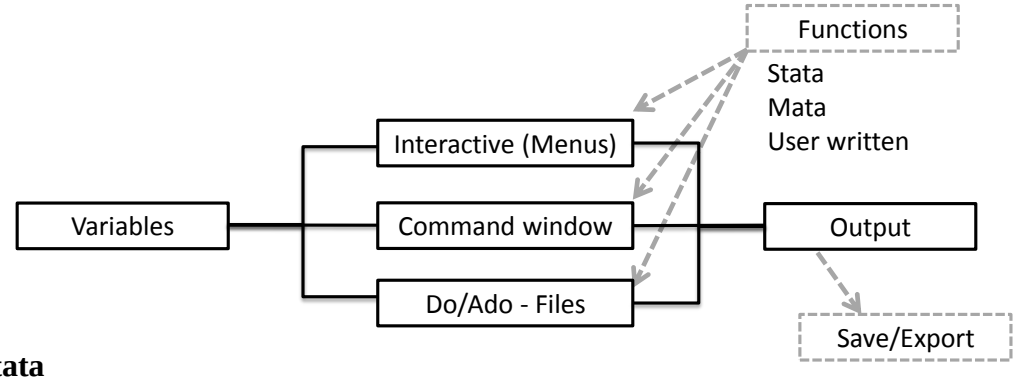


There are 4 different packages available: Stata MP (multi-processor) which is the most powerful, Stata SE (special edition), Intercooled STATA and Small STATA. The main difference between these versions is the maximum number of variables, regressors and observations that can be handled (see http://www.stata.com/order/options-e.html#difference-sm for details). The LSE is currently running the SE-version, version 10.

Stata is a command-driven package. Although the newest versions also have pull-down menus from which different commands can be chosen, the best way to learn Stata is still by typing in the commands. This has the advantage of making the switch to programming much easier which will be necessary for any serious econometric work. However, sometimes the exact syntax of a command is hard to get right –in these cases, I often use the menu-commands to do it once and then copy the syntax that appears.

You can enter commands in either of three ways:
- Interactively: you click through the menu on top of the screen
- Manually: you type the first command in the command window and execute it, then the next, and so on.
- Do-file: type up a list of commands in a "do-file", essentially a computer programme, and execute the do-file.

The vast majority of your work should use do-files. If you have a long list of commands, executing a do-file once is a lot quicker than executing several commands one after another. Furthermore, the do-file is a permanent record of all your commands and the order in which you ran them. This is useful if you need to "tweak" things or correct mistakes – instead of inputting all the commands again one after another, just amend the do-file and re-run it. Working interactively is useful for "I wonder what happens if …?" situations. When you find out what happens, you can then add the appropriate command to your do-file. To start with we'll work interactively, and once you get the hang of that we will move on to do-files.



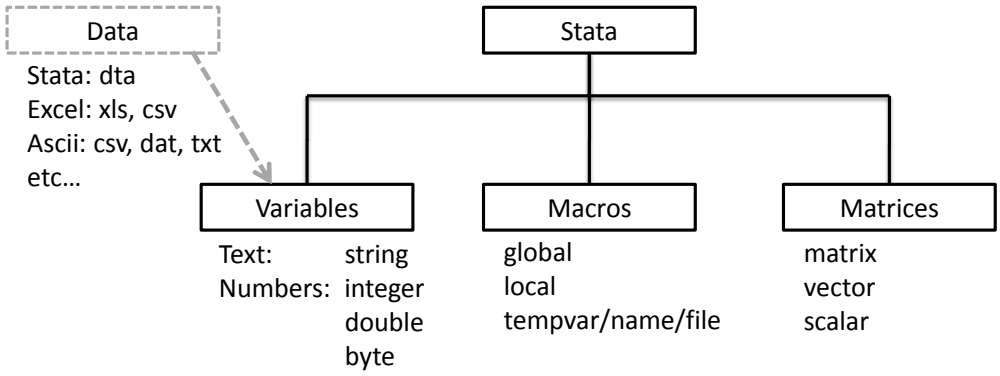
Data in Stata

Stata is a versatile program that can read several different types of data. Mainly files in its own dta format, but also raw data saved in plain text format (ASCII format). Every program you use (i.e. Excel or other statistical packages) will allow you to export your data in some kind of ASCII file. So you should be able to load all data into Stata.

When you enter the data in Stata it will be in the form of variables. Variables are organized as column vectors with individual observations in each row. They can hold numeric data as well as strings. Each row is associated with one observation, that is the 5^{th} row in each variable holds the information of the 5^{th} individual, country, firm or whatever information you data entails.

Information in Stata is usually and most efficiently stored in variables. But in some cases it might be easier to use other forms of storage. The other two forms of storage you might find useful are matrices and macros. Matrices have rows and columns that are not associated with any observations. You can for example store an estimated coefficient vector as a $k \times 1$ matrix (i.e. a column vector) or the variance matrix which is $k \times k$. Matrices use more memory then variables and the size of matrices is limited 11,000 (800 in Stata/IC), but your memory will probably run out before you hit that limit. You should therefore use matrices sparingly.

The third option you have is to use macros. Macros are in Stata what variables are in other programming languages, i.e. named containers for information of any kind. Macros come in two different flavours, local or temporary and global. Global macros stay in the system and once set, can be accessed by all your commands. Local macros and temporary objects are only created within a certain environment and only exist within that environment. If you use a local macro in a do-file it, you can only use it for code within that do-file.



Getting help

Stata is a command driven language – there are over 500 different commands and each has a particular syntax required to get any various options. Learning these commands is a time-consuming process but it is not hard. At the end of each class notes I shall try to list the commands that we have covered but there is no way we will cover all of them in this short introductory course. Luckily though, Stata has a fantastic options for getting help. In fact, most of your learning to use Stata will take the form of self-teaching by using manuals, the web, colleagues and Stata's own help function.

Manuals

The Stata manuals are available in MA – many people have them on their desks. The User Manual provides an overall view on using Stata. There are also a number of Reference Volumes, which are basically encyclopaedias of all the different commands and all you ever needed to know about each one. If you want to find information on a particular command or a particular econometric technique, you should first look up the index at the back of any manual to find which volumes have the relevant information. Finally, there is a separate Graphics Manual, panel data manual (cross-sectional time-series) and one on survey data.

Stata's in-built help and website

Stata also has an abbreviated version of its manuals built-in. Click on Help, then Contents. Stata's website has a very useful FAQ section at http://www.stata.com/support/faqs/. Both the in-built help and the FAQs can be simultaneously searched from within Stata itself (see menu Help>Search). Stata's website also has a list of helpful links at http://www.stata.com/links/resources1.html.

The web

As with everything nowadays, the web is a great place to look to resolve problems. There are numerous chat-rooms about Stata commands, and plenty of authors put new programmes on their websites. Google should help you here.

Colleagues

The other place where you can learn a lot is from speaking to colleagues who are more familiar with Stata functions than you are – the LSE is littered with people who spend large parts of their days typing different commands into Stata, you should make use of them if you get really stuck.

Directories and folders

Like Dos, Linux, Unix and Windows, Stata can organise files in a tree-style directory with different folders. You should use this to organise your work in order to make it easier to find things at a later date. For example, create a folder "data" to hold all the datasets you use, sub-folders for each dataset, and so on. You can use some Dos and Linux/Unix commands in Stata, including:

```
. cd "H:\ECStata"          - change directory to "H:\ECStata"
. mkdir "FirstSession"     - creates a new directory within the current one (here, H:\ECStata)
. dir                      - list contents of directory or folder (you can also use the linux/unix command: ls)
```

Note, Stata is case sensitive, so it will not recognise the command `CD` or `Cd`. Also, quotes are only needed if the directory or folder name has spaces in it – `"h:\temp\first folder"` – but it's a good habit to use them all the time.

Another aspect you want to consider is whether you use absolute or relative file paths when working with Stata. Absolute file paths include the complete address of a file or folder. The `cd` command in the previous example is followed by an absolute path. The relative file path on the other hand gives the location of a file or folder relative to the folder that you are currently working in. In the previous example `mkdir` is followed by a relative path. We could have equivalently typed:

```
. mkdir "H:\ECStata\FirstSession"
```

Using relative paths is advantageous if you are working on different computers (i.e. your PC at home and a library PC or a server). This is important when you work on a larger or co-authored project, a topic we will come back to when considering project management. Also note that while Windows and Dos use a backslash "\" to separate folders, Linux and Unix use a slash "/". This will give you trouble if you work with Stata on a server (ARC at the LSE). Since Windows is able to understand a slash as a separator, I suggest that you use slashes instead of backslashes when working with relative paths.

```
. mkdir "/FirstSession/Data"      - create a directory "Data" in the folder H:\ECStata\FirstSession
```

Reading data into Stata

When you read data into Stata what happens is that Stata puts a copy of the data into the memory (RAM) of your PC. All changes you make to the data are only temporary, i.e. they will be lost once you close Stata, unless you save the data. Since all analysis is contacted within the limitations of the memory, this is usually the bottle neck when working with large data sets. There are different ways of reading or entering data into Stata:

use

If your data is in Stata format, then simply read it in as follows:

```
. use "H:\ECStata\G7 less Germany pwt 90-2000.dta", clear
```

The `clear` option will clear the revised dataset currently in memory before opening the other one.

Or if you changed the directory already, the command can exclude the directory mapping:

```
. use "G7 less Germany pwt 90-2000.dta", clear
```

insheet

If your data is originally in Excel or some other format, you need to prepare the data before reading it directly into Stata. You need to save the data in the other package (e.g. Excel) as either a csv (comma separated values ASCII text) or txt (tab-delimited ASCII text) file. There are some ground-rules to be followed when saving a csv- or txt-file for reading into Stata:

- The first line in the spreadsheet should have the variable names, e.g. series/code/name, and the second line onwards should have the data. If the top row of the file contains a title then delete this row before saving.
- Any extra lines below the data or to the right of the data (e.g. footnotes) will also be read in by Stata, so make sure that only the data itself is in the spreadsheet before saving. If necessary, select all the bottom rows and/or right-hand columns and delete them.
- The variable names cannot begin with a number. If the file is laid out with years (e.g. 1980, 1985, 1990, 1995) on the top line, then Stata will run into problems. In such instances you can for example, place an underscore in front of each number (e.g. select the row and use the spreadsheet package's "find and replace" tools): 1980 becomes _1980 and so on.
- Make sure there are no commas in the data as it will confuse Stata about where rows and columns start and finish (again, use

"find and replace" to delete any commas before saving – you can select the entire worksheet in Excel by clicking on the empty box in the top-left corner, just above 1 and to the left of A).

- Some notations for missing values can confuse Stata, e.g. it will read double dots (`..`) or hyphens (`-`) as text. Use find & replace to replace such symbols with single dots (`.`) or simply to delete them altogether.

Once the csv- or txt-file is saved, you then read it into Stata using the command:

```
. insheet using "H:\ECStata\G7 less Germany pwt 90-2000.txt", clear
```

Note that if we had already changed to `H:\ECStata` using the `cd` command, we could simply type:

```
. insheet using "G7 less Germany pwt 90-2000.txt", clear
```

There are a few useful options for the insheet command ("options" in Stata are additional features of standard commands, usually appended after the command and separated by a comma – we will see many more of these). The first option is `clear` which you can use if you want to insheet a new file while there is still data in memory:

```
. insheet using "H:\ECStata\G7 less Germany pwt 90-2000.txt", clear
```

Alternatively, you could first erase the data in memory using the command `clear` and then insheet as before.

The second option, `names`, tells Stata that the file you insheet contains the variable names in the first row. Normally, Stata should recognise this itself but sometimes it simply doesn't – in these cases `names` forces Stata to use the first line in your data for variable names:

```
. insheet using "F:\Stata classes\G7 less Germany pwt 90-2000.txt", names clear
```

Finally, the option `delimiter("char")` tells Stata which delimiter is used in the data you want to insheet. Stata's `insheet` automatically recognises tab- and comma-delimited data but sometimes different delimiters are used in datasets (such as ";"):

```
. insheet using "h:\wdi-sample.txt", delimiter(";")
```

infix

While comma separated or tab delimited data is very common today, older data is often saved in a fixed ASCII format. The data cannot be read directly but a codebook is necessary that explains how the data is stored. An example for data that is stored this way is the U.S. National Health Interview Survey (NHIS). The first two lines of one of the 1986 wave look like this:

```
10861096028901   05 011  1  02130103000000000000000000001
10861096028902   05 011  1  02140103000000000000000000001
```

The codebook (usually a pdf or txt file) that accompanies the data tells you that the first 2 numbers code the record type, the following 2 numbers are the survey year (here 1986), the fifth number is the quarter (here the first quarter) of the interview and so on.

To read this type of data into Stata we need to use the `infix` command and provide Stata with the information from the codebook.

```
. infix rectype 1-2 year 3-4 quarter 5 […] using "H:\ECStata\NHIS1986.dat", clear
```

Since there are a lot of files it my be more convenient to save the codebook information in a separate file, a so called "dictionary file". The file would look like this:

```
infix dictionary using NHIS1986.dat {
       rectype       1-2
       year   3-4
       quarter       5
       […]
}
```

After setting up this file we would save it as NHIS1986.dct and use it in the infix command. Note that we used a relative path in the dictionary file, i.e. by not stating a file path for NHIS1986.dat we assume that the raw data is located in the same directory as the dictionary file. With the dictionary file we do not need to refer to the data directly anymore:

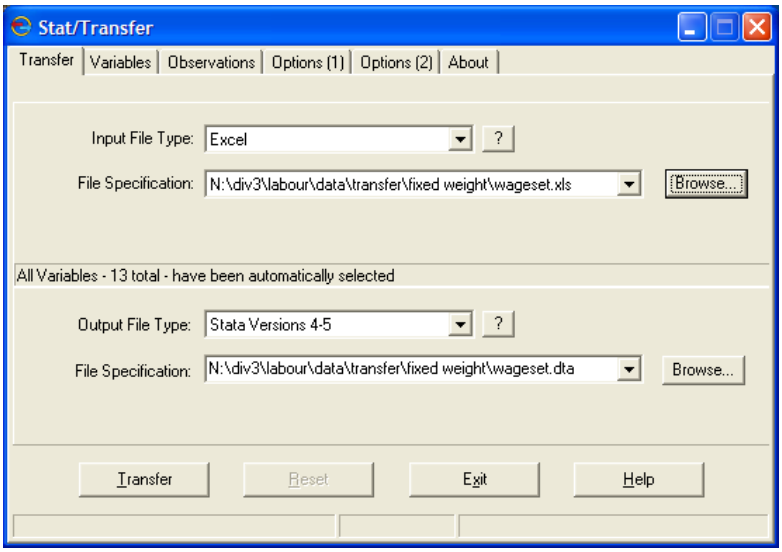
```
. infix using "H:\ECStata\NHIS1986.dct", clear
```

Since setting up dictionary files is a lot of work, we are lucky that for the NHIS there exists already a dictionary file that can be read with SAS (a program similar to Stata). After reading the data into SAS and saving it we can use a tool called Stat/Transfer to convert the file into the Stata data format.

Stat/Transfer program

This is a separate package that can be used to convert a variety of different file-types into other formats, e.g. SAS or Excel into Stata or vice versa. You should take great care to examine the converted data thoroughly to ensure it was converted properly.

It is used in a very user-friendly way (see screen shot below) and is useful for changing data between lots of different packages and format.



Manual typing or copy-and-paste

If you can open the data in Excel, you can usually copy and paste the data into the Stata data editor. All you need to do is select the columns in Excel; copy them; open the Stata data editor; and paste. This works usually quite well but entails certain pitfalls. The data format might not turn out to be correct, missing values might not be accounted for properly and in some cases language issues might arise (in some countries a comma rather than a decimal point is used).

Manually typing in the data is the tedious last resort – if the data is not available in electronic format, you may have to type it in manually. Start the Stata program and use the edit command – this brings up a spreadsheet-like where you can enter new data or edit existing data.

This can be done directly by typing the variables into the window, or indirectly using the input command.

Variable and data types

Indicator or data variables

You can see the contents of a data file using the `browse` or `edit` command. The underlying numbers are stored in "data variables", e.g. the `cgdp` variable contains national income data and the `pop` variable contains population data. To know what each data-point refers to, you also need at least one "indicator variable", in our case countryisocode (or country) and year tell us what country and year each particular gdp and population figure refers to. The data might then look as follows:

country	countryisocode	year	pop	cgdp	openc
Canada	CAN	1990	27700.9	19653.69	51.87665
France	FRA	1990	58026.1	17402.55	43.46339
Italy	ITA	1990	56719.2	16817.21	39.44491
Japan	JPN	1990	123540	19431.34	19.81217
United Kingdom	GBR	1990	57561	15930.71	50.62695
United States	USA	1990	249981	23004.95	20.61974

This layout ensures that each data-point is on a different row, which is necessary to make Stata commands work properly.

Numeric or string data

Stata stores or formats data in either of two ways – numeric or string. Numeric will store numbers while string will store text (it can also be used to store numbers, but you will not be able to perform numerical analysis on those numbers).

Numeric storage can be a bit complicated. Underneath its Windows platform, Stata, like any computer program, stores numbers in binary format using 1's and 0's. Binary numbers tend to take up a lot of space, so Stata will try to store the data in a more compact format. The different formats or storage types are:

byte : integer between -127 and 100 e.g. dummy variable

int : integer between -32,767 and 32,740 e.g. year variable

long : integer between -2,147,483,647 and 2,147,483,620 e.g. population data

float : real number with about 8 digits of accuracy e.g. production output data

double : real number with about 16 digits of accuracy

The Stata default is "float", and this is accurate enough for most work. However, for critical work you should make sure that your data is "double". Note, making all your numerical variables "double" can be used as an insurance policy against inaccuracy, but with large data-files this strategy can make the file very unwieldy – it can take up lots of hard-disk space and can slow down the running of Stata. Also, if space is at a premium, you should store integer variables as "byte" or "int", where appropriate.

The largest 27 numbers of each numeric format are reserved for missing values. For byte the standard missing value is 101, which will be represented by a dot (.) in Stata. Later when we evaluate logic expressions we need to account for this.

String is arguably more straightforward – any variable can be designated as a string variable and can contain up to 244 characters, e.g. the variable `name` contains the names of the different countries. Sometimes, you might want to store numeric variables as strings, too. For example, your dataset might contain an indicator variable `id` which takes on 9-digit values. If `id` were stored in float format (which is accurate up to only 8 digits), you may encounter situations where different `id` codes are rounded to the same amount. Since we do not perform any calculations on `id` we could just as well store it in string format and avoid such problems.

To preserve space, only store a variable with the minimum string necessary – so the longest named `name` is "United Kingdom" with 14 letters (including the space). A quick way to store variables in their most efficient format is to use the `compress` command – this goes through every observation of a variable and decides the least space-consuming format without sacrificing the current level of accuracy in the data.

```
. compress
```

Missing values

Missing numeric observations are denoted by a single dot (`.`), missing string observations are denoted by blank double quotes (`""`). For programming purposes different types of missing values can be defined (up to 27) but this will rarely matter in applied work.

Examining the data

It is a good idea to examine your data when you first read it into Stata – you should check that all the variables and observations are there and in the correct format.

List

As we have seen, the `browse` and `edit` commands start a pop-up window in which you can examine the raw data. You can also examine it within the results window using the `list` command – although listing the entire dataset is only feasible if it is small. If the dataset is large, you can use some options to make `list` more useable. For example, list just some of the variables:

```
. list country* year year pop
     +-----------------------------------------+
     |        country   countr~e   year    pop |
     |-----------------------------------------|
  1. |         Canada        CAN   1990   27700.9 |
  2. |         France        FRA   1990   58026.1 |
  3. |          Italy        ITA   1990   56719.2 |
  4. |          Japan        JPN   1990    123540 |
  5. | United Kingdom        GBR   1990     57561 |
     |-----------------------------------------|
  6. |  United States        USA   1990    249981 |
  7. |         Canada        CAN   1991   28030.9 |
  8. |         France        FRA   1991   58315.8 |
  9. |          Italy        ITA   1991   56750.7 |
 10. |          Japan        JPN   1991    123920 |
     |-----------------------------------------|
```

The star after "country" works as a place holder and tells Stata to include all variables that start with "country".

Alternatively we could focus on all variables but list only a limited number of observations. For example the observation 45 to 49:

```
. list in 45/49
```

Or both:

```
. list country countryisocode year pop in 45/49
     +-----------------------------------------+
     |        country   countr~e   year    pop |
     |-----------------------------------------|
 45. |          Italy        ITA   1997   57512.2 |
 46. |          Japan        JPN   1997    126166 |
 47. | United Kingdom        GBR   1997     59014 |
 48. |  United States        USA   1997    268087 |
 49. |         Canada        CAN   1998     30248 |
     |-----------------------------------------|
```

Subsetting the data (if and in qualifiers)

In the previous section we used the "in" qualifier. The qualifier ensures that commands apply only to a certain subset of the data. The "in" qualifier is followed by a range of observations.

```
. list in 45/49
. list in 50/l
. list in -10/l
```

The first command lists observations 45 to 49, the second the observations from 50 until the last observation (lower case l) and the last command lists the last ten observations.

A second way of subsetting the data is the "if" qualifier (more on this later on). The qualifier is followed by an expression that evaluates either to "true" or "false" (i.e. 1 or 0). We could for example list only the observations for 1997:

```
. list if year == 1997
```

Browse/Edit

We have already seen that `browse` starts a pop-up window in which you can examine the raw data. Most of the time we only want to view a few variables at a time however, especially in large datasets with a large number of variables. In such cases, simply list the variables you want to examine after `browse`:

```
. browse name year pop
```

The difference with edit is that this allows you to manually change the dataset.

Assert

With large datasets, it often is impossible to check every single observation using `list` or `browse`. Stata has a number of additional commands to examine data which are described in the following. A first useful command is `assert` which verifies whether a certain statement is true or false. For example, you might want to check whether all population (`pop`) values are positive as they should be:

```
. assert pop>0
. assert pop<0
```

If the statement is true, `assert` does not yield any output on the screen. If it is false, `assert` gives an error message and the number of contradictions.

Describe

This reports some basic information about the dataset and its variables (size, number of variables and observations, storage types of variables etc.).

```
. describe
```

Note that you can use the `describe` command for a file that hasn't yet been read into Stata:

```
. describe using "H:\wdi-sample.dta"
```

Codebook

This provides extra information on the variables, such as summary statistics of numerics, example data-points of strings, and so on. Codebook without a list of variables will give information on all variables in the dataset.

```
. codebook country
```

Summarize

This provides summary statistics, such as means, standard deviations, and so on.

```
. summarize
```

Variable	Obs	Mean	Std. Dev.	Min	Max
country	0				
countryiso~e	0				
year	66	1995	3.18651	1990	2000
pop	66	98797.46	79609.33	27700.9	275423
cgdp	66	22293.23	4122.682	15930.71	35618.67
openc	66	42.54479	18.64472	15.91972	86.80463
csave	66	24.31195	5.469772	16.2536	37.80159
ki	66	23.52645	4.634476	17.00269	35.12778
grgdpch	66	1.582974	1.858131	-3.981008	5.172524

Note that code and name are string variables with no numbers, so no summary statistics are reported for them. Also, year is a numeric, so it has summary statistics. Additional information about the distribution of the variable can be obtained using the `detail` option:

```
. summarize, detail
```

Tabulate

This is a versatile command that can be used, for example, to produce a frequency table of one variable or a cross-tab of two variables.

```
. tab name
          Name |      Freq.     Percent        Cum.
---------------+-----------------------------------
        Canada |         10       14.29       14.29
        France |         10       14.29       28.57
       Germany |         10       14.29       42.86
         Italy |         10       14.29       57.14
         Japan |         10       14.29       71.43
United Kingdom |         10       14.29       85.71
 United States |         10       14.29      100.00
---------------+-----------------------------------
         Total |         70      100.00
```

We can use the `tabulate` command combined with the `sum(varname)` option to gain a quick idea of the descriptive statistics of certain subgroups. For example the average population of all G7 countries (except Germany) in all years:

```
. tab year, sum(pop)
           |           Summary of POP
      year |        Mean   Std. Dev.       Freq.
-----------+------------------------------------
      1990 |   95588.034   81969.389           6
      1991 |     96250.4   82904.603           6
      1992 |   96930.667   83847.404           6
      1993 |    97603.95   84778.438           6
      1994 |   98222.017   85639.914           6
      1995 |   98872.683   86510.583           6
      1996 |   99462.834    87354.77           6
      1997 |   100083.63   88256.404           6
      1998 |   100676.85   89128.951           6
      1999 |   101246.45   89981.306           6
      2000 |   101834.58   90824.442           6
-----------+------------------------------------
     Total |   98797.464   79609.332          66
```

There are also options to get the row, column and cell percentages as well as chi-square and other statistics – check the Stata manuals or on-line help for more information.

Inspect

This is another way to eyeball the distribution of a variable, including as it does a mini-histogram. Also useful for identifying outliers or unusual values, or for spotting non-integers in a variable that should only contain integers.

```
. inspect cgdp

cgdp:                                                 Number of Observations
-------                                                                    Non-
                                                      Total   Integers    Integers
|      #                        Negative                -          -           -
|  #   #                        Zero                    -          -           -
|  #   #                        Positive               66          -          66
|  #   #                                              -----      -----       -----
|  #   #   #                    Total                  66          -          66
|  #   #   #   .   .            Missing                 -
+----------------------                               -----
15930.71       35618.67                               66

  (66 unique values)
```

Graph

Stata has very comprehensive graphics capabilities (type "help graph" for more details). You can graph a simple histogram with the command:

```
. graph twoway histogram cgdp
```

Or a two-way scatterplot using:

```
. graph twoway scatter cgdp pop
```

While graphs in Stata 9 and Stata 10 have the advantage of looking quite fancy, they are also very slow. Often, you just want to visualise data without actually using the output in a paper or presentation. In this case, it is useful to switch to version 7 graphics which are much faster:

```
. graph7 cgdp pop
```

Saving the dataset

The command is simply `save`:

```
. save "H:\ECStata\G7 less Germany pwt 90-2000.dta", replace
```

The `replace` option overwrites any previous version of the file in the directory you try saving to. If you want to keep an old version as back-up, you should save under a different name, such as "new_G7". Note that the only way to alter the original file permanently is to save the revised dataset. Thus, if you make some changes but then decide you want to restart, just re-open the original file.

Preserve and restore

If you are going to make some revisions but are unsure of whether or not you will keep them, then you have two options. First, you can save the current version, make the revisions, and if you decide not to keep them, just re-open the saved version. Second, you can use the `preserve` and `restore` commands; `preserve` will take a "photocopy" of the dataset as it stands and if you want to revert back to that copy later on, just type `restore`.

Keeping track of things

Stata has a number of tools to help you keep track of what work you did to datasets, what's in the datasets, and so on.

Do-files and log-files

Instead of typing commands one-by-one interactively, you can type them all in one go within a do-file and simply run the do-file once. The results of each command can be recorded in a log-file for review when the do-file is finished running.

Do-files can be written in any text editor, such as Word or Notepad. Stata also has its own editor built in – click the icon along the top of the screen with the pad-and-pencil logo (although it looks like an envelope to me). Most do-files follow the following format:

```
clear
cd "c:\projects\project1\"
capture log close
log using class.log, replace
set more off
set memory 100m

LIST OF COMMANDS

log close
```

To explain the different commands:

`clear` – clears any data currently in Stata's memory. If you try opening a datafile when one is already open, you get the error message: `no; data in memory would be lost`

`cd c:\projects\project1\` - sets the default directory where Stata will look for any files you try to open and save any files you try to save. So, if you type `use wdi-sample.dta`, Stata will look for it in this folder. If, during the session, you want to access a different directory, then just type out its destination in full, e.g. `use "c:\data\production.dta"` will look for the file in the `c:\data` folder. Note again that if you use spaces in file or directory names, you must include the file path in inverted commas.

`capture log close` – closes any log-files that you might have accidentally left open. If there were no log-file actually open, then the command `log close` on its own would stop the do-file running and give the error message: `no log file open`. Using `capture` tells Stata to ignore any error messages and keep going.

`log using class1.log, replace` – starts a log-file of all the results. The `replace` option overwrites any log file of the same name, so if you re-run an updated do-file again the old log-file will be replaced with the updated results. If, instead, you want to add the new log-file to the end of previous versions, then use the `append` option.

`set more off` – when there are a lot of results in the results window, Stata pauses the do-file to give you a chance to review each page on-screen and you have to press a key to get more. This command tells Stata to run the entire do-file without pausing. You can then review the results in the log file.

`set memory 100m` – Stata's default memory may not be big enough to handle large datafiles. Trying to open a file that is too large returns a long error message beginning: `no room to add more observations`. You can adjust the memory size to suit. First check the size of the file using the describe command (remember that you can use `describe` for a file that hasn't yet been read into Stata). This reports the size of the file in bytes. Then set memory just a bit bigger. Note, setting it too large can take the PC's memory away from other applications and slow the computer down, so only set it as large as necessary. For example, `describe using "c:\data\WDI-sampe.dta"` reports the size of the file to be 2,730 bytes, so `set memory 1m` should be sufficient.

`log close` – closes the log file.

It is good practice to keep extensive notes within your do-file so that when you look back over it you know what you were trying to achieve with each command or set of commands. You can insert comments in two different ways:

`//`

Stata will ignore a line if it starts with two consecutive slashes (or with an asterisk `*)`, so you can type whatever you like on that line. Note, comments are also useful for getting Stata to temporarily ignore commands – if you decide later to re-insert the command into your do-file, just delete the slashes or the asterisk.

`/* */`

You can place notes after a command by inserting it inside these pseudo-parentheses, for example:

```
. use "c:\data\WDI-sample.dta", clear /* opens 1998 production data */
```

These pseudo-parentheses are also useful for temporarily blocking a whole set of commands – place `/*` at the beginning of the first command, `*/` at the end of the last, and Stata will just skip over them all.

Labels

You can put labels on datasets, variables or values – this helps to make it clear exactly what the dataset contains.

A dataset label of up to 80 characters can be used to tell you the data source, it's coverage, and so on. This label will then appear when you `describe` the dataset. For example, try the following:

```
. label data "  Data from Penn World Tables 6.1"

. describe
```

Variable names tend to be short – you can use up to 32 characters, but for ease of use it's best to stick to about 8 or 10 as a maximum. This can give rise to confusion about what the variable actually represents – what exactly is `gdp` and in what units is it measured? Which is where variable labels, with a capacity of 80 characters, come in.

```
. label variable cgdp "GDP per capita in constant international dollars"
```

It can also be helpful to label different values. Imagine countries were coded as numbers (which is the case in many datasets). In this case, a tabulation may be confusing – what country does 1 represent, or 2 or 3?

```
. tabulate code

       code |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         10       33.33       33.33
          2 |         10       33.33       66.67
          3 |         10       33.33      100.00
------------+-----------------------------------
      Total |         30      100.00
```

It might be better to label exactly what each value represents. This is achieved by first "defining" a label (giving it a name and specifying the mapping), then associating that label with a variable. This means that the same label can be associated with several variables – useful if there are several "yes/no/maybe" variables, for example. The label name itself can be up to 32 characters long (e.g. `countrycode`), and each value label must be no more than 80 characters long (e.g. "France" or "Italy").

```
. label define countrycode 1 "Canada" 2 "Germany" 3 "France"

. label values code countrycode
```

Now, the tabulation should make more sense:

```
. tabulate code

       code |      Freq.     Percent        Cum.
------------+-----------------------------------
     Canada |         10       33.33       33.33
    Germany |         10       33.33       66.67
     France |         10       33.33      100.00
------------+-----------------------------------
      Total |         30      100.00
```

see what each code represents, use `codebook` or:

```
. label list countrycode
countrycode:
           1 Canada
           2 Germany
           3 France
```

Notes

You can also add Post-it notes to your dataset or to individual variables to, for example, remind you of the source of the data, or to remind you of work you did or intend to do on a variable.

```
. note: data from PWT
. note cgdp: This is per capita variable
```

You can also time-stamp these notes:

```
. note cgdp: TS need to add Germany to complete the G7
```

Review your notes by simply typing `notes`:

```
. notes

_dta:
  1.  data from PWT

cgdp:
  1.  This is per capita variable
  2.  15 Feb 2006 13:01 need to add Germany to complete the G7
```

Stata will also tell you that there are notes when you use `describe`:

```
. describe
```

You can also delete notes. To drop all notes attached to a variable:

```
. note drop cgdp
```

To drop just one in particular:

```
. note drop cgdp in 2
```

Review

One final tool for keeping track is reviewing a list of previous commands. To see the last four, for example:

```
. #review 4
```

This is especially useful if you are working in interactive mode on a "what happens if…". When you are happy with the sequence of commands you've tried, you can `review`, then cut and paste into your do-file.

Some shortcuts for working with Stata

- Most commands can be abbreviated, which saves some typing. For example: `summarize` to `sum`, `tabulate` to `tab`, `save` to `sa`. The abbreviations are denoted by the underlined part of the command in Stata help or the Stata manuals.
- You can also abbreviate variable names when typing. This should be used with caution, as Stata may choose a variable different to the one you intended. For example, suppose you have a dataset with the variables `pop`, `popurban` and `poprural`. If you want summary statistics for `popurban`, the command `sum pop` will actually give statistics for the `pop` variable. An alternative is to type in part of the variable name and then hit the tabulator key. Stata will fill in the rest of the variable name until ambiguity arises. In this example typing in `po` and hitting the tabulator key results in Stata putting in `pop`, typing in `popr` and hitting the tab key will give `poprural`.
- Stata's default file type is `.dta`, so you don't need to type that when opening or saving Stata files:
  `sa "G7 less Germany pwt 90-2000"` is the same as `sa "G7 less Germany pwt 90-2000.dta"`
- You can save retyping commands or variable names by clicking on them in the review and variable windows – they will then appear in the command window. You can also cycle back and forth through previous commands using the PageUp and PageDown keys on your keyboard. Similarly, variable names can be easily entered by clicking on them in the Variables Window (bottom-left of the screen).
- Over time, you will find yourself using the same commands or the same sequence of commands again and again, e.g. the list of commands at the beginning of a log-file. Save these in a "common commands" text file from which you can cut and paste into your do-files.

A note on working empirical projects.

When you start working on an empirical project you will quite quickly accumulate a large number of do files, data sets, log files and other output. To keep track of things you should use comments throughout your do files that remind you of what the do file does, when you created it, when you last changed it, what links it has to other do files, etc. When saving files add the date they were created to them (e.g. 20081005 for the 5th of October 2008) and sort files into different folders. I keep one folder for do files, another for data and a third folder to export results and save log files.

If you are working with large data sets, the UK Labour Force Survey, the U.S. Current Population Survey, etc. memory, or rather the lack thereof becomes a problem rather quickly. The memory problem is aggravated by a curious (and apparently unsolvable) hard limit on the amount of memory that can be allocated to Stata when using Windows XP (32-bit). The limit is around 1.2 gb of memory, no matter how much actual memory your PC has. But two or even three gigabytes of memory might not suffice for your projects. The first thing you should do when running into the memory threshold is to drop everything from the dataset that you do not need for your analysis. You can always reload the original data set once you ran a particular regression (though this might be a fairly slow procedure). Use the `compress` command. You will not lose any information by doing so, but potentially save some space. If everything else fails, you can apply for an ARC account. ARC is a Unix server that runs Stata (among other statistical programs). But disk space and computation time is scarce so you should only use ARC as a last resort. You can find more information on ARC at the LSE itservice website: http://www.lse.ac.uk/itservices/help/itsupport/ARC/default.htm.

And last but not least: Never forget to backup your work!

If you use Stata regularly you might want to think about integrating Stata with an external editor. An excellent choice for Windows is WinEdt (http://www.winedt.com). WinEdt is mainly used for writing Latex documents but allows Stata command highlighting by installing an add on (http://www.winedt.org/Config/modes/Stata.php). If you want to use Stata and Latex with WinEdt I recommend you install another add on called "Auto Mode" (see my website for details). If you do not want to pay the $30 license fees, you can also use free alternatives such as Notepad ++ (again see my website for details).

Database Manipulation

Now we are going to take the data that is in a form that Stata understands and we will organise those datasets by combining many together into a single large dataset, deleting unwanted variables, and also creating some new variables. Finally, we will learn a few techniques to close gaps in your data (extrapolation, splicing).

Organising datasets

Rename

You may want to change the names of your variables, perhaps to make it more transparent what the variable is:

```
. rename countryisocode country_code
. ren grgdpch gdp_growth
```

Note, you can only rename one variable at a time.

Recode and Replace

You can change the values that certain variables take, e.g. suppose 1994 data actually referred to 1924:

```
. recode year (1994 = 1924)
```

This command can also be used to recode missing values to the dot that Stata uses to denote missings. And you can recode several variables at once. Suppose a dataset codes missing population and gdp figures as –999:

```
. recode pop cgdp (-999 = .)
```

`recode` can not only several variables but several changes at the same time. We could for example use recode to generate a new variable with categorical population values, 1 for countries with less than 50 million inhabitants, 2 for 50 to 100 million and 3 for more than 100 million inhabitants.

```
. recode pop (0 / 50000 = 1)  (50001 / 100000 = 2) (100000 / 7000000 = 3)
```

With string variables, however, you need to use the replace command (see more on this command below):

```
. replace country="United Kingdom" if country_code =="GBR"
```

Keep and drop (including some further notes on if-processing)

The original dataset may contain variables you are not interested in or observations you don't want to analyse. It's a good idea to get rid of these first – that way, they won't use up valuable memory and these data won't inadvertently sneak into your analysis. You can tell Stata to either `keep` what you want or `drop` what you don't want – the end results will be the same. For example, we can get rid of unwanted variables as follows:

```
. keep country year pop cgdp
```

or

```
. drop country_code openc csave ki grgdpch
```

or

```
. drop country_code openc - gdp_growth
```

Each of these will leave you with the same set of variables. Note that the hyphen sign (-) is a useful shortcut, e.g. the first one indicates all the variables between `openc` and `gdp_growth` are to be dropped. However, you must be careful that the order of the variable list is correct, you don't want to inadvertently drop a variable that you thought was somewhere else on the list. The variable list is in the variables window or can be seen using either the `desc` or `sum` commands.

You can also drop or keep observations, such as those after or before 1995:

```
. keep if year >= 1995
```

or

```
. drop if year < 1995
```

Note that missing values of numeric variables are treated as a large positive number, so both commands would keep not only all observations for 1995 and after but also all observations with missing values in the `year` variable.

The different relational operators are:

== equal to

!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Keeping observations for the years 1990 to 1995 only:

```
. keep if (year>=1990 & year<=1995)
```

or

```
. drop if (year<1990 | year>1995)
```

Or, to get really fancy, keep the observations for 1990-95 and 1997-99:

```
. keep if ((year>=1990 & year<=1995) | (year>=1997 & year<=1999))
```

Note, the different logical operators are:

&	and
|	or
~	not
!	not

You may want to drop observations with specific values, such as missing values (denoted in Stata by a dot):

```
. drop if pop == .
```

Sometimes it is convenient to use a shorthand notation and leave out the operators:

```
. drop if pop
```

is short for

```
. drop if pop~=0
```

that is we drop all the observations where a population of zero is reported.

You may want to keep observations for all countries other than those for Italy:

```
. drop if country_code != "ITA"
```

Note, with string variables, you must enclose the observation reference in double quotes. Otherwise, Stata will think that ITA refers to a variable and claim not to be able to find what you are referring to.

If you know the observation number, you can selectively keep or drop different observations. Dropping observations 1 to 10:

```
. drop if _n <= 10
```

Dropping the last observation (number _N) in the dataset:

```
. drop if _n == _N
```

Both `_n` and `_N` are inbuilt system variables. The upper case N refers to the last number of observation. Combined with the `by` (see below) this can be the number of the last observation in a subset rather than the whole data. Lower case n always refers to the number of each observation (combined with by, this can be again a relative relation).

Finally, you may want to keep only a single occurrence of a specific observation type, e.g. just the first observation of each country code we can use Stata's indexing capabilities. A variable name followed by square brackets means that we want to refer to a certain observation, this can be an absolute value `[1]` would mean the first observation or `[_N]` the last observation or a relative index `[_n]` means the current and `[_n-1]` the observation before the current observation. To keep only the first occurrence of each country we can use:

```
. keep if country[_n]~=country[_n-1]
```

or simply

```
. keep if country~=country[_n-1]
```

Stata starts at observation number one and applies the command, then moves onto observation two and applies the command again, then onto three and so on. So, starting at one _n=1 but there is no observation _n-1 = 0, so the country in one cannot equal the country in zero (which is missing: "") and the observation will be kept. Moving on to two: the country in two equals the country in one (both AGO), so the observation will be dropped. Each subsequent observation with country AGO will also be dropped. When we get to an observation with a different country (which will be ALB), the two countries will be different (AGO~=ALB) and the observation will be kept. Thus, we will end up being left with just the first observation for each country.

Sort

From the previous example, hopefully you will have realised the importance of the order of your observations. If the country codes had started out all jumbled up, then we would have ended up with a completely different set of observations. Suppose we applied the above command to the following dataset:

Number in dataset	country	Result
1	AGO	Kept since _n=0 does not exist
2	AGO	Dropped since country==country[_n-1]
3	ALB	Kept
4	ALB	Dropped
5	AGO	Kept
6	ALB	Kept
7	BEL	Kept

We would actually end up with numerous occurrences of some country codes. This shows how sorting the data first is important:

```
. sort country
```

If you wanted to make sure the observation that was kept was the earliest (i.e. 1950), then first:

```
. sort country year
```

This command first sorts the data by country, and then within each country code it sorts the data by year. This ensures that the first observation for every country (the one that is kept) will be 1950.

Note that sorting is in ascending order (A,B,C or 1950, 1951, 1952). To sort in descending order, you need to use the `gsort` command:

```
. gsort –country
```

This gives ZWE first, then ZMB, ZAR, ZAF, YEM and so on. Note that you need to place a minus sign before every variable you want to sort in descending order. This command allows you to sort in complicated ways, e.g. to sort country codes in descending order but then years in ascending order:

```
. gsort –country year
```

By-processing

You can re-run a command for different subsets of the data using the `by` prefix. For example, to get summary statistics of population broken down by year:

```
. sort year
. by year: sum pop
```

Note that you have to either sort the data first or use the `bysort` prefix:

```
. bysort year: sum pop
```

The by prefix causes the sum command to be repeated for each unique value of the variable year. The result is the same as writing a list of sum commands with separate if statements for each year:

```
. sum pop if year==1990
. sum pop if year==1991
. sum pop if year==1992
. sum pop if year==1993
. sum pop if year==1994
```

By-processing can be useful when organising your dataset. In our sort examples above we asked Stata to keep only the first observation for each country. The `by` command makes this selection a lot easier:

```
. bysort country: keep in 1
```
or equivalently
```
. bysort country: keep if _n == 1
```

Both commands will keep the first observation of each subset, i.e. the first observation for each country. But this is not necessarily the earliest observation. To ensure that we select the first year for each country we need to sort within the by-group (country) we selected:

```
. bysort country (year): keep in 1
```

The parentheses tell Stata to sort within country rather than opening up a different by group:

bysort country		bysort country (year)		bysort country year	
AGO	2000	AGO	1990	AGO	2000
AGO	1990	AGO	2000	AGO	1990
ALB	1990	ALB	1990	ALB	1990
ALB	2000	ALB	2000	ALB	2000

Append, merge and joinby

You can combine different datasets into a single large dataset using the `append`, `merge` and `joinby` commands. `append` is used to add extra observations (rows). Suppose you have two datasets containing the G7 less Germany PWT data for different countries and/or different years. The datasets have the same `variables country / year / pop / etc`, but one dataset has data for 1970-1990 (called "`G7 less Germany pwt 70-90.dta`") and the other has data for 1975-1998 (called "`G7 less Germany pwt 90-2000.dta`").

```
. use "H:\ECStata\G7 less Germany pwt 90-2000.dta", clear
. append using "H:\ECStata\G7 less Germany pwt 70-90.dta"
. save "H:\ECStata\G7 less Germany pwt.dta", replace
```

Append

CTRY	YEAR	C-GDP
USA	1990	23,004
GBR	1990	15,930

CTRY	YEAR	C-GDP
USA	2000	35,618
GBR	2000	24,253

CTRY	YEAR	C-GDP
USA	1990	23,004
GBR	1990	15,930
USA	2000	35,618
GBR	2000	

`append` is generally very straightforward. There is one imp ortant exception, however, if the two datasets you want to append have stored their variables in different formats (meaning string vs. numeric – having different numeric formats, for example byte vs. float, does not matter). In this case, Stata converts the data in the file to be appended to the format of the original file and in the process replaces all values to missing! To detect such problems while using `append`, watch out for messages like:

```
. (note: pop is str10 in using data but will be float now)
```

This indicates that a variable (here: pop) has been transformed from string to float – and contains all missing values now for the appending dataset (here: all years 1970-1990). It is thus very important to check via `describe` that the two files you intend to append have stored all variables in the same broad data categories (string/numeric). If this is not the case, you will need to transform them first (see the commands `destring` and `tostring` below).

`merge` is used to add extra variables (columns). Suppose we now also have a second dataset containing the same indicator variables `country / year`, but one dataset has data for GDP per capita and other variables, and the second has data for shares in GDP per capita of consumption and investment.

Merge (1-to-1)

CTRY	YEAR	C-GDP
USA	1990	23,004
GBR	1990	15,930

CTRY	YEAR	POP
USA	1990	250
GBR	1990	58

CTRY	YEAR	C-GDP	Pop
USA	1990	23,004	250
GBR	1990	15,930	58

Merge (1-to-n)

CTRY	YEAR	C-GDP	Pop
USA	1990	23,004	250
GBR	1990	15,930	58
USA	2000	35,618	275
GBR	2000	24,253	

YEAR	UN-M
1990	159
2000	189

CTRY	YEAR	C-GDP	Pop	UN-M
USA	1990	23,004	250	159
GBR	1990	15,930	58	159
USA	2000	35,618	275	189
GBR	2000	24,253		

You must first ensure that both datasets are sorted by their common indicator variables, then `merge` according to these variables.

```
. use "H:\ECStata\G7 less Germany pwt.dta", clear
. sort country year
. save "H:\ECStata\G7 less Germany pwt.dta", replace
. use "H:\ECStata\G7 extra data.dta", clear /* "master" data */
. sort country year
. merge country year using "H:\ECStata\G7 less Germany pwt.dta" /*"using" data */
. tab _merge /* 1= master, 2= using, 3= both */
```

Stata automatically creates a variable called _merge which indicates the results of the merge operation. It is crucial to tabulate this variable to check that the operation worked as you intended. The variable can take on the values:

1 : observations from the master dataset that did not match observations from the using dataset
2 : observations from the using dataset that did not match observations from the master dataset
3 : observations from the both datasets that matched

Ideally, all observations will have a _merge value of 3. However, it may be possible, for instance, that the master dataset has observations for extra countries or extra years. If so, then some observations will have a _merge value of 1. You should tabulate these to confirm what the extra observations refer to:

```
. tab country if _merge==1
. tab year if _merge==1
. tab _merge
```

```
     _merge |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |         31       10.95       10.95
          3 |        252       89.05      100.00
------------+-----------------------------------
      Total |        283      100.00
```

`tab country if _merge==1` then reveals that the these extra observations are for the country "GER" or Germany. Now see if you can successfully incorporate the data on Germany between 1970-2000 for all of these variables? Look at help for how to do it.

Finally, `joinby` joins, within groups formed by the variables list behind the command, observations of the dataset in memory with another Stata-format dataset. "join" means "form all pairwise combinations". For example, you might have an industry classification (both codes and names) in one file and corresponding tariff rates in another (with only codes and tariff rates). Tariff rates vary across time but the industry classification does not. Now, you would like to match every industry with a time series of tariffs and also know what the different industry codes stand for. Since the classification data does not contain a year variable, you cannot use `merge` (unless you create a year variable and `expand` the data first which we will learn how to do later on). However, if you type

```
. joinby indclass using tariffs.dta
```

this will create all possible combinations between `indclass` (the variable that contains the different classification categories) and `year`. If the master and using dataset contain common variables, `joinby` will use the master contents. Also, observations unique to one or the other datasets are ignored, unless you overrule this using the option `unmatched` (see `help joinby` for details).

Collapse

This command converts the data into a dataset of summary statistics, such as sums, means, medians, and so on. One use is when you have monthly data that you want to aggregate to annual data:

```
. collapse (sum) monthpop, by(country year)
```

or firm-level data that you want to aggregate to industry level:

```
. collapse (sum) firmoutput, by(industry year month)
```

`by()` excludes the indicator variable that you are collapsing or summing over (`month` in the first example, `firm` in the second) – it just contains the indicator variables that you want to collapse by. Note that if your dataset contains other variables beside the indicator variables and the variables you are collapsing, they will be erased.

One possible problem that arises in the use of collapse is in its treatment of missings. It returns the summary statistic of missing values as zero. If, for example, when using the PWT Afghanistan ("AFG") contains all missing values for pop. If you wanted to aggregate population data over time (for whatever reasons), collapse would report aggregate population for Afghanistan as zero, not missing. If , instead, you want aggregate population figures to be missing if any or all of the year data is missing, then use the

following coding (the technicalities of it will become clearer later, after you learn how to create dummy variables):

```
. gen missing=(pop==.)
. collapse (sum) pop missing, by(countrygroup)
. replace firmoutput=. If missing>0
. rename pop aggpop
. drop missing
```

Note, if you are running this command on a large dataset, it may be worthwhile to use the `fast` option – this speeds things up skipping the preparation of a backup if the command is aborted by the user pressing BREAK, but this is really only useful for when you are working interactively).

Order, aorder, and move

These commands can be used to do some cosmetic changes to the order of your variable list in the variables window, e.g. if you want to have the indicator variables on top of the list. `aorder` alphabetically sorts variables and `order` brings them in a user-specified order:

```
. aorder
. order countrycode year pop
```

If you do not list certain variables after order, they will remain where they are. m`ove` is used if you simply want to swap the position of two variables, e.g. bringing year to the top:

```
. move year countrycode
```

Creating new variables

Generate, egen, replace

The two most common commands for creating new variables are `gen` and `egen`. We can create a host of new variables from the existing data with the `gen` command:

```
. gen realgdp=(pop*1000)*cgdp           /* real GDP in current prices */
. gen lpop=ln(pop)                      /* log population */
. gen popsq=pop^2                       /* squared population */
. gen ten=10                            /* constant value of 10 */
. gen id=_n                             /* id number of observation */
. gen total=_N                          /* total number of observations */
. gen byte yr=year-1900                 /* 50,51,etc instead of 1950,1951 */
. gen str6 source="PWT6.1"              /* string variable */
. gen largeyear=year if pop>5000 & pop!=.
```

A couple of things to note. First, Stata's default data type is float, so if you want to create a variable in some other format (e.g. byte, string), you need to specify this. Second, missing numeric observations, denoted by a dot, are interpreted by Stata as a very large positive number. You need to pay special attention to such observations when using `if` statements. If the last command above had simply been `gen largeyear=year if pop>5000`, then `largeyear` would have included observations `1950-1959` for `AGO`, even though data for those years is actually missing.

The `egen` command typically creates new variables based on summary measures, such as sum, mean, min and max:

```
. egen totalpop=sum(pop), by(year)     /* world population per year */
. egen avgpop=mean(pop), by(year)      /* average country pop per year */
. egen maxpop=max(pop)                 /* largest population value */
. egen countpop=count(pop)             /* counts number of non-missing obs */
. egen groupid=group(country_code)     /* generates numeric id variable for countries */
```

The `egen` command is also useful if your data is in long format (see below) and you want to do some calculations on different observations, e.g. `year` is long, and you want to find the difference between 1995 and 1998 populations. The following routine will achieve this:

```
. gen temp1=pop if year==1995
. egen temp2=max(temp1), by(country_code)
. gen temp3=pop-temp2 if year==1998
. egen diff=max(temp3), by(country)
. drop temp*
```

Note that both `gen` and `egen` have `sum` functions. `egen` generates the total sum, and `gen` creates a cumulative sum. The running cumulation of `gen` depends on the order in which the data is sorted, so use it with caution:

```
. egen totpop=sum(pop)                  /* sum total of population = single result*/
. gen cumpop=sum(pop)                   /* cumulative total of population */
```

To avoid confusion you can use the `total` function rather than `sum` for `egen`. It will give you the same result.

As with `collapse`, `egen` has problems with handling missing values. For example, summing up data entries that are all missing yields a total of zero, not missing (see `collapse` below for details and how to solve this problem).

The `replace` command modifies existing variables in exactly the same way as `gen` creates new variables:

```
. gen lpop=ln(pop)
. replace lpop=ln(1) if lpop==.   /* missings now ln(1)=0 */
. gen byte yr=year-1900
. replace yr=yr-100 if yr >= 100  /* 0,1,etc instead of 100,101 for 2000 onwards */
```

Converting strings to numerics and vice versa

As mentioned before, Stata cannot run any statistical analyses on string variables. If you want to analyse such variables, you must first `encode` them:

```
. encode country, gen(ctyno)
. codebook  ctyno                 /* Tells you the link with the data* /
```

This creates a new variable `ctyno`, which takes a value of 1 for `CAN`, 2 for `FRA`, and so on. The labels are automatically computed, based on the original string values – you can achieve similar results but without the automatic labels using `egen`

```
ctyno=group(country).
```

You can go in the other direction and create a string variable from a numerical one, as long as the numeric variable has labels attached to each value:

```
. decode ctyno, gen(ctycode)
```

If you wanted to convert a numeric with no labels, such as `year`, into a string, the command is:

```
. tostring year, generate(yearcode)
```

And if you have a string variable that only contains numbers, you can convert them to a numeric variable using:

```
. destring yearcode, generate(yearno)
```

This last command can be useful if a numeric variable is mistakenly read into Stata as a string. You can confirm the success of each conversion by:

```
. desc country ctyno ctycode year yearcode yearno
```

Combining and dividing variables

You may wish to create a new variable whose data is a combination of the data values of other variables, e.g. joining country code and year to get `AGO1950`, `AGO1951`, and so on. To do this, first convert any numeric variables, such as `year`, to string (see earlier), then use the command:

```
. gen str7 ctyyear=country_code+yearcode
```

If you want to create a new numeric combination, first convert the two numeric variables to string, then create a new string variable that combines them, and finally convert this string to a numeric:

```
. gen str4 yearcode=string(year)
. gen str7 popcode=string(pop)
. gen str11 yearpopcode=yearcode+popcode
. gen yearpop=real(yearpopcode)

. sum yearpopcode yearpop /* displays the result */
```

To divide up a variable or to extract part of a variable to create a new one, use the `substr` function. For example, you may want to reduce the `year` variable to `70`, `71`, `72`, etc. either to reduce file size or to `merge` with a dataset that has `year` in that format:

```
. gen str2 yr=substr(yearcode,3,2)
```

The first term in parentheses is the string variable that you are extracting from, the second is the position of the first character you want to extract (--X-), and the third term is the number of characters to be extracted (--XX). Alternatively, you can select your starting character by counting from the end (2 positions from the end instead of 3 positions from the start):

```
. gen str2 yr=substr(yearcode,-2,2)
```

Things can get pretty complicated when the string you want to divide isn't as neat as `yearcode` above. For example, suppose you have data on city population and that each observation is identified by a single variable called `code` with values such as `"UK London"`, `"UK Birmingham"`, `"UK Cardiff"`, `"Ireland Dublin"`, `"France Paris"`, `"Germany Berlin"`, `"Germany Bonn"`, and so on. The `code` variable can be broken into `country` and `city` as follows:

```
. gen str10 country=substr(code,1,strpos(code," ")-1)
. gen str10 city=trim(substr(code, strpos(code," "),11))
```

The `strpos()` function gives the position of the second argument in the first argument, so here it tells you what position the blank space takes in the `code` variable. The `country` substring then extracts from the `code` variable, starting at the first character, and extracting a total of `3-1=2` characters for `UK`, `8-1=7` characters for `Ireland` and so on. The `trim()` function removes any leading or trailing blanks. So, the `city` substring extracts from the `code` variable, starting at the blank space, and extracting a total of 11 characters including the space, which is then trimmed off. Note, the `country` variable could also have been created using `trim()`:

```
. gen str10 country=trim(substr(code,1,strpos(code," ")))
```

Dummy variables

You can use `generate` and `replace` to create a dummy variable as follows:

```
. gen largepop=0
. replace largepop=1 if (pop>=5000 & pop!=. )
```

Or you can combine these in one command:

```
. gen largepop=(pop>=5000 & pop~=.)
```

Note, the parenthesis are not strictly necessary, but can be useful for clarity purposes. It is also important to consider missing values when generating dummy variables. With the command above a missing value in `pop` results in a `0` in `largepop`. If you want to keep missing values as missing, you have to specify an `if` condition:

```
. gen largepop=(pop>=5000 & pop~=.) if pop~=.
```

The `& pop~=.` part is not strictly necessary in this case, but it doesn't hurt to keep it there either.

You may want to create a set of dummy variables, for example, one for each `country`:

```
. tab country, gen(cdum)
```

This creates a dummy variable `cdum1` equal to 1 if the country is "`CAN`" and zero otherwise, a dummy variable `cdum2` if the country is "`FRA`" and zero otherwise, and so on up to `cdum7` for "`USA`". You can refer to this set of dummies in later commands using a wild card, `cdum*`, instead of typing out the entire list.

A third way to generate dummy variables is by using the `xi` prefix as a command.

```
. xi i.country
i.country          _Icountry_1-6       (_Icountry_1 for country==Canada omitted)
```

The command generates 5 dummy variables, omitting the sixth. This is useful to avoid a dummy variable trap (i.e. perfect multicolinearity of the intercept and a set of dummy variables) in a regression. We can control which category should be omitted and also specify that all dummy variables should be generated (see `help xi`). But the main use of `xi` is as a prefix, if we want to include a large set of dummy variables or dummy variables and interactions with dummy variables, we can use `xi` to save us the work of defining every variable by itself. And we can easily drop the variables after we used them with `drop _*`.

```
. xi: sum i.country*year
i.country        _Icountry_1-6      (_Icountry_1 for country==Canada omitted)
i.country*year   _IcouXyear_#       (coded as above)

    Variable |      Obs       Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
 _Icountry_2 |       66   .1666667    .3755338         0          1
 _Icountry_3 |       66   .1666667    .3755338         0          1
 _Icountry_4 |       66   .1666667    .3755338         0          1
 _Icountry_5 |       66   .1666667    .3755338         0          1
 _Icountry_6 |       66   .1666667    .3755338         0          1
-------------+--------------------------------------------------------
        year |       65   1933.846    347.2559         0       2000
_IcouXyear_2 |       65        307    725.5827         0       2000
_IcouXyear_3 |       65        307    725.5827         0       2000
_IcouXyear_4 |       65   337.6154     753.859         0       2000
_IcouXyear_5 |       65   337.6154     753.859         0       2000
-------------+--------------------------------------------------------
_IcouXyear_6 |       65   337.6154     753.859         0       2000
```

Lags and leads

To generate lagged population in the G7 dataset:

```
. so countrycode year
. by countrycode: gen lagpop=pop[_n-1] if year==year[_n-1]+1
```

Processing the statement country-by-country is necessary to prevent data from one `country` being used as a lag for another, as could happen with the following data:

country	Year	pop
AUS	1996	18312
AUS	1997	18532
AUS	1998	18751
AUT	1950	6928
AUT	1951	6938
AUT	1952	6938

The `if` argument avoids problems when there isn't a full panel of years – if the dataset only has observations for 1950, 1955, 1960-1998, then lags will only be created for 1961 on. A lead can be created in similar fashion:

```
. so country year
. by country: gen leadpop=pop[_n+1] if year==year[_n+1]-1
```

Cleaning the data

This section covers a few techniques that can be used to fill in gaps in your data.

Fillin and expand

Suppose you start with a dataset that has observations for some years for one country, and a different set of years for another country:

country	Year	pop
AGO	1960	4816
AGO	1961	4884
ARG	1961	20996
ARG	1962	21342

You can "rectangularize" this dataset as follows:

```
. fillin country year
```

This creates new missing observations wherever a `country-year` combination did not previously exist:

country	Year	pop
AGO	1960	4816
AGO	1961	4884
AGO	1962	.
ARG	1960	.
ARG	1961	20996
ARG	1962	21342

It also creates a variable `_fillin` that shows the results of the operation; 0 signifies an existing observation, and 1 a new one. If no country had data for 1961, then the `fillin` command would create a dataset like:

country	Year	pop
AGO	1960	4816
AGO	1962	.
ARG	1960	.
ARG	1962	21342

So, to get a proper "rectangle", you would first have to ensure that at least one observation for the year 1961 exists:

```
. expand 2 if _n==1
. replace year=1961 if _n==_N
. replace pop=. if _n==_N
```

`expand 2` creates 2 observations identical to observation number one `(_n==1)` and places the additional observation at the end of the dataset, i.e observation number `_N`. As well as recoding the year in this additional observation, it is imperative to replace all other data with missing values – the original dataset has no data for 1961, so the expanded dataset should have missings for 1961. After this has been done, you can now apply the `fillin` command to get a complete "rectangle".

These operations may be useful if you want to estimate missing values by, for example, extrapolation. Or if you want to replace all missing values with zero or some other amount.

Interpolation and extrapolation

Suppose your population time-series is incomplete – as with some of the countries in the PWT (e.g. `STP` which is Sao Tome and Principe). You can linearly interpolate missing values using:

```
. so country
. by country: ipolate pop year, gen(ipop)
```

country	Year	pop
STP	1995	132
STP	1996	135.29
STP	1997	.
STP	1998	141.7
STP	1999	144.9
STP	2000	148

Note, first of all, that you need to interpolate by country, otherwise Stata will simply interpolate the entire list of observations irrespective of whether some observations are for one country and some for another. The first variable listed after the `ipolate` command is the variable you actually want to interpolate, the second is the dimension along which you want to interpolate. So, if you believe population varies with time, you can interpolate along the time dimension. You then need to specify a name for a new variable that will contain all the original and interpolated values – here `ipop`. You can use this cleaned-up version in its entirety in subsequent analysis, or you can select values from it to update the original variable, e.g. to clean values for `STP` only:

```
. replace pop=ipop if country=="STP"
```

Linear extrapolation can be achieved with the same command, adding the `epolate` option, e.g. to extrapolate beyond 2000:

```
. so country
. by country: ipolate pop year, gen(ipop) epolate
```

Note, however, that Stata will fail to interpolate or extrapolate if there are no missing values to start with. No 2001 or 2002 observations actually exist, so Stata will not actually be able to extrapolate beyond 2000. To overcome this, you will first have to create blank observations for 2001 and 2002 using `expand` (alternatively, if these observations exist for other countries, you can rectangularise the dataset using `fillin`).

Splicing data from an additional source

It is also possible to fill gaps with data from another source, as long as the series from both sources are compatible. For example, one source may provide data for 1950-92 and another for 1970-2000 with data for the overlapping years being identical. In such cases, you can simply replace the missing years from one source with the complete years from the other.

It is more common, however, for the data in the overlapping years to be similar but not identical, as different sources will often use different methodologies and definitions. In such instances, you can splice the data from one source on to that from the other. For example, the latest version of PWT has data for the unified Germany going back to 1970 while the earlier PWT5.6 has data for West Germany going all the way back to 1950. It is arguably reasonable to assume that the trends in the total German data were similar to those in the West German data and to splice the early series onto the up-to-date version. To do this, you must first merge the old series into the new one, making sure to rename the variables first, e.g. rename pop in PWT6.1 to pop61, and to ensure that both Germany's are coded identically, e.g. GER.

```
. gen temp1=pop61/pop56 if country=="GER" & year==1970
. egen diff=mean(temp1), by(country)
. replace pop61=pop56*diff if pop61==. & year<1970
```

country	year	pop56	pop61	temp1	diff
GER	1968	59499			1.281248
GER	1969	60069			1.281248
GER	1970	60651	77709	1.281248	1.281248
GER	1971	61303	78345		1.281248
GER	1972	61675	78715		1.281248

Panel Data Manipulation: Long versus Wide data sets

Reshape

Datasets may be laid out in wide or long formats. Suppose we keep population data for 1970-75 only:

```
. keep country country_code year pop
. keep if year<=1975
```

In **long** format, this looks like:

country	country_code	year	pop
Canada	CAN	1970	21324
Canada	CAN	1971	21962.1
Canada	CAN	1972	22219.6
Canada	CAN	1973	22493.8
Canada	CAN	1974	22808.4
Canada	CAN	1975	23142.3
France	FRA	1970	52040.8
France	FRA	1971	52531.8
France	FRA	1972	52993.1
France	FRA	1973	53420.5
France	FRA	1974	53771
France	FRA	1975	54016

And the same data in **wide** format looks like:

country	country_code	pop1970	pop1971	pop1972	pop1973	pop1974	pop1975
Canada	CAN	21324	21962.1	22219.6	22493.8	22808.4	23142.3
France	FRA	52040.8	52531.8	52993.1	53420.5	53771	54016
United Kingdom	GBR	55632	55928	56097	56223	56236	56226
Germany	GER	77709	78345	78715	78956	78979	78679
Italy	ITA	53821.9	54073.5	54381.3	54751.4	55110.9	55441
Japan	JPN	103720	104750	106180	108660	110160	111520
United States	USA	205089	207692	209924	211939	213898	215981

The vast majority of Stata commands work best when the data is in long format. In any case, to convert formats from long to wide:

```
. reshape wide pop, i(country_code) j(year)
```

or from wide to long:

```
. reshape long pop, i(country_code) j(year)
```

The variable(s) immediately behind `long` or `wide` is the one that contains the data we want to reshape (the "data variable", in our case `pop`). Note that in the `reshape long` case, Stata will reshape all variables that start with the letters you put behind long. Here, there are actually six of them (pop1970-pop1975, all starting with pop). The i() specifies the variable(s) whose unique values denote a logical observation in wide format. In our case, this is `country`. It uniquely identifies every data entry in wide format (here: `pop`). The j() specifies the variable whose unique values denote a sub-observation, in our case `year`. That is, within every group of countries, `year` uniquely identifies observations. In long format, i() and j() together completely identify each observation.

If there are more than two indicator variables in wide format, then be careful to include the correct list in i(). For example, if there were also an agegroup indicator variable, so that pop actually referred to population in a given age group, then we could reshape the data from country / agegroup / year / pop to country / agegroup / pop1960 / pop1961 / etc using:

```
. reshape wide pop, i(country agegroup) j(year)
```

If there is more than one data variable, first `drop` the variables you are not interested in, and then make sure to include the full list you are interested in reshaping within the command:

```
. reshape wide pop cgdp pi, i(country) j(year)
```

This will create new variables pop1970-1975, cgdp1970-1975 and pi1970-1975. Note if you had not dropped all other variables beforehand, you would get an error message. For example, if you had forgotten to delete `cc`:

```
. cc not constant within country
. Type "reshape error" for a listing of the problem observations.
```

As Stata suggests, "reshape error" will list all observations for which country does not uniquely identify observations in wide

format (here, these are actually all observations!). More generally, any variable that varies across both i() and j() variables either needs to be dropped before `reshape wide` or be included in the data variable list. Intuitively, Stata would not know where to put the data entries of such variables once `year` has gone as an identifier.

We could also have reshaped the original long data to have the country variable as wide:

```
. reshape wide pop, i(year) j(country) string
```

Note, you need to specify the string option when j() is a string variable. Browsing the resulting data:

year	popCAN	popFRA	popGBR	popGER	popITA	popJPN	popUSA
1970	21324	52040.8	55632	77709	53821.9	103720	205089
1971	21962.1	52531.8	55928	78345	54073.5	104750	207692
1972	22219.6	52993.1	56097	78715	54381.3	106180	209924
1973	22493.8	53420.5	56223	78956	54751.4	108660	211939
1974	22808.4	53771	56236	78979	55110.9	110160	213898
1975	23142.3	54016	56226	78679	55441	111520	215981

To create variables named CANpop / FRApop / GBRpop instead of popCAN/popFRA/popGBR, use:

```
. reshape wide @pop, i(year) j(country) string
```

The @ is useful when, for example, you start with a dataset that has the dimension you want to reshape written the "wrong" way around. It denotes where the suffix/prefix is affixed. Suppose you are given a dataset with country / youngpop / oldpop. You can reshape the pop variable to long to give country / agegroup / pop using:

```
. reshape long @pop, i(country) j(agegroup) string
```

Estimation

We now move on from the manipulation of databases to the more exciting material of running regressions. In this tutorial, we shall use data from Francesco Caselli's "Accounting for Cross-Country Income Differences" which is published in the Handbook of Economic Growth (2005, Ch. 9). There is a link to these data on my website. But before we start looking at the basics of regression commands, let us look at Stata's graph capabilities in more detail.

Descriptive graphs

Stata allows a large variety of graphs and options to customize them (see http://www.ats.ucla.edu/stat/stata/topics/graphics.htm for a more indepth overview). Here we will only consider some very basic graphs. We already introduced the graph command to plot a simple histogram and a scatter plot. We will extend these examples here and focus on formatting graphs via the graph options. A very useful command that helps us manage very long code in do-files is the `#delimit` command.

```
#delimit ;
```

or short

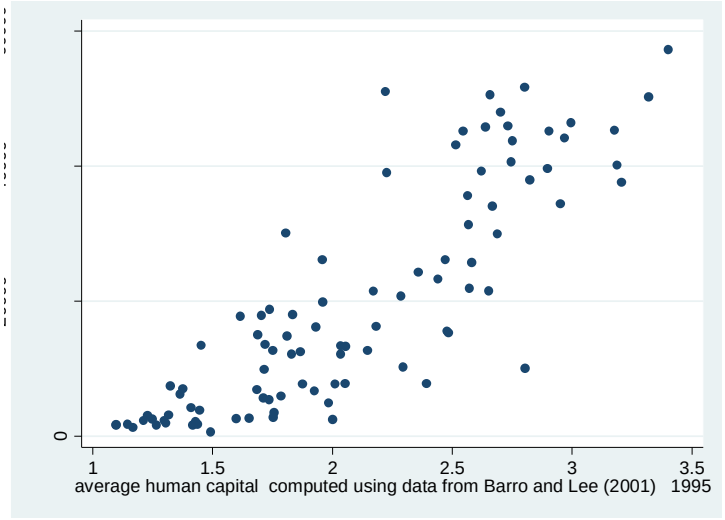
```
#d;
```

The delimit command changes what Stata perceives as the end of a command. Normally we write one command per line, that is the end of a line denotes the end of a command as well. `#delimit` introduces a character – the semicolon (;) – to denote the end of a command. This way we can either write several commands on one line, or (more importantly) use several lines for the same command. To revert back to the end of line as the end of a command we write:

```
#d cr
```

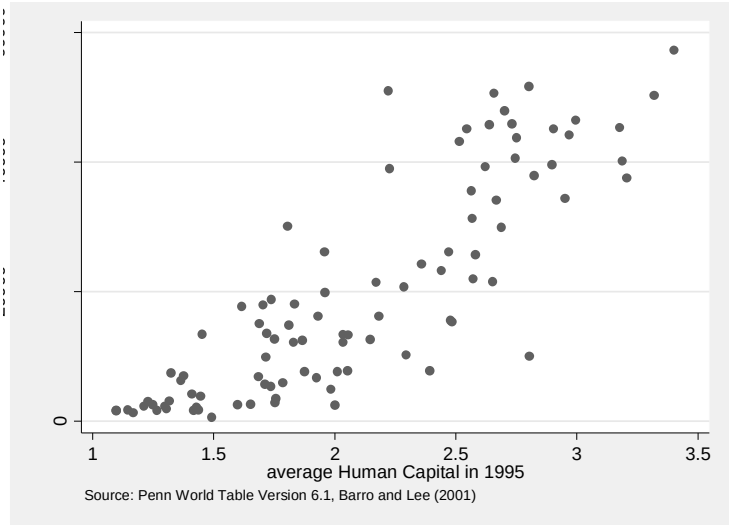
Let us start with a simple scatter plot, a command we have used already.

```
graph twoway scatter y h
```



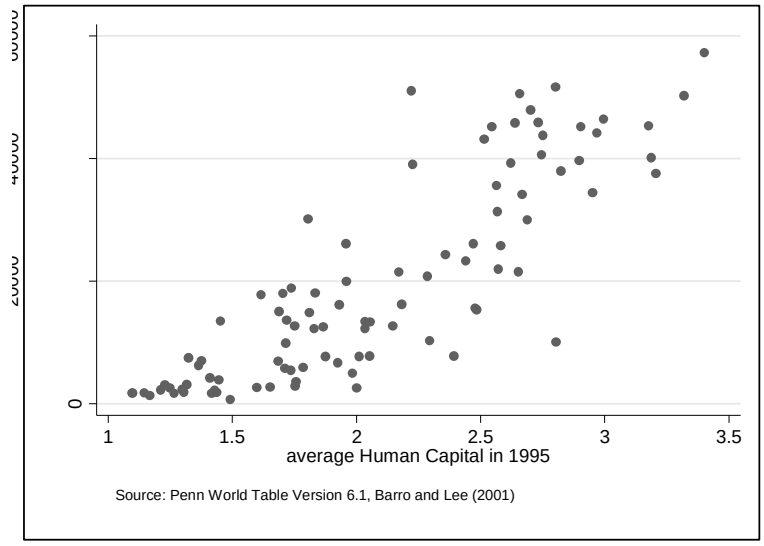
The graph depicts real per worker GDP on the vertical and average human capital on the horizoantal axis, for all countries in the sample. We left the options all at standard ans so we have a coloured graph with the variable labels as names for the axes. The graph command has some plots that do not fall into the `twoway` category, e.g. box-and-whiskers plots, but most of the plots you see in published articles are twoway-type of plots. The first thing we should change about this graph is the naming of the axes. We could either change the label of the underlying variables, or use the `graph` commands axis options. The second thing we want to change is that Stata prodcues a coloured graph. While colours are good for presentations, they will not appear in any paper.

```
#d;
graph twoway scatter y h,
      scheme(s2mono) ytitle("real GDP per worker")
      xtitle("average Human Capital in 1995")
      note("Source: Penn World Table Version 6.1, Barro and Lee (2001)");
#d cr
```



Stata has some inbuild colour definitions (like the standard colour scheme of the first graph or this black-and-white scheme) which are available via the `scheme` option. The grey shading around the graph is not really a nice feature and so we will make some changes to the standard scheme we chose. Also the footnote about the source of the data could be a little further apart from the axis label. The title, axis names, the note field, etc. are

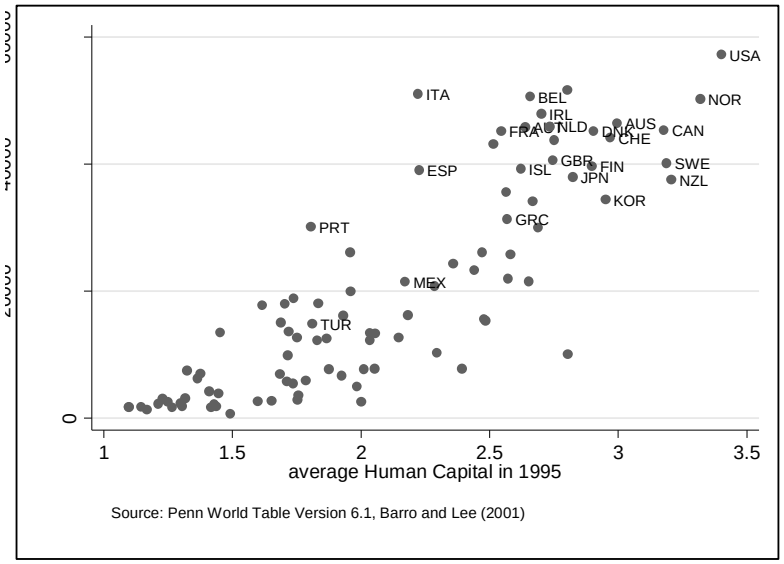
```
#d;
graph twoway scatter y h,
      scheme(s2mono) ytitle("real GDP per worker")
      xtitle("average Human Capital in 1995")
      note("Source: Penn World Table Version 6.1, Barro and Lee (2001)"
            , margin(medium))
      graphregion(fcolor(white) lpattern(solid) lcolor(black) lwidth(medium));
#d cr
```



To give the reader a better idea of what is represented by the dots we could add labels to them. Given the number of observations

this would probably be more confusing then helpful. So instead of putting a country name to each single point we could select only a few. But how can we tell Stata to distinguis between two sets of points? We generate a dummy variable that indicates to which set an observation belongs and draw to scatter plots within one graph. One for each set. We are not limited to drawing only scatter plots in the same graph, we could for example overlay the scatter plot with the fitted line from a regression. Note that as with the "note" textbox in the previous example, each individual element of the command can have own options, in addition to the options that apply to the whole graph. To seperate to plots within one graph we can either surround each plot command by parentheses – `()` – or use two bars – `||` –, both are fine. Note that the plots are drawn in the order that we use in the command, i.e. the first plot will be the backmost and each additional plot will be layered on top of it.

```
#d;
graph twoway
       (scatter y h if oecd != 1, mstyle(p1))
       (scatter y h if oecd == 1, mstyle(p1) mlabel(iso))
       , scheme(s2mono) ytitle("real GDP per worker")
       xtitle("average Human Capital in 1995")
       note("Source: Penn World Table Version 6.1, Barro and Lee (2001)"
              , margin(medium))
       graphregion(fcolor(white) lpattern(solid) lcolor(black) lwidth(medium))
       legend(off);
#d cr
```



We need to explicitly specify the style of the scatter markers since Stata chooses different markers for each plot by default, we also need a variable that specifies the label names for the markers. Also by default Stata adds a legend that normally tells the reader what the different plots/symbols/lines in a graph mean. But since we only use two plots as a trick to add labels we do not need a legend an surpress it with the `legend(off)` option.

How can we now save the graphs so we can use them in Word or Latex? The easiest (and probably worst way) is to copy the graphs directly into Word or Scientific Workplace. Note that you cannot do this if you use an editor other than Scientific Workplace (e.g. WinEdt, TeXnicCenter) for Latex documents. To do this simply select "Copy Graph" from the "Edit" menu of the graph window in Stata and paste into your program of choice.

The better choice is to export your graphs. Stata allows for different file types when you use the `graph export` function. Sadly not all of them are available under all types of operating systems. Under Windows I would recommend saving graphs as PNG files. PNG is a file format similar to GIF which was used a lot (and is somethimes still used) on internet websites. If you are using Word, PNG is the way to go. If you use Latex EPS is a good choice, but might give you trouble compiling your document (it is not possible to compile a document with EPS graphics directly to PDF but you have to compile a DVI file and then in turn make it into a PDF). EPS is a format that should be readable under all operating systems. So we finish our graph with:

```
graph save "H:\ECStata\MyFirstGraph", replace
graph export "H:\ECStata\MyFirstGraph.png", replace
```

Estimation syntax

Before we delve into some particular regression commands let us consider the syntax in Stata help for the details on estimation. The basic information is:

- There are many different models of estimation. The main commands include `regress`, `logit`, `logistic`, `sureg`.
- Most have a similar syntax:

```
command varlist [weight] [if exp] [in range] [, options]
```

- 1st variable in the varlist is the dependent variable, and the remaining are the independent variables.
- You can use Stata's syntax to specify the estimation sample; you do not have to make a special dataset.
- You can, at any time, review the last estimates by typing the estimation command without arguments.
- The level() option to indicate the width of the confidence interval. The default is level(95).

Once you have carried out your estimation, there are a number of post-estimation commands that are useful:

- You can recall the estimates, VCM, standard errors, etc…;
- You can carry out hypothesis testing => `test` (Wald tests), `testnl` (non-linear Wald tests), `lrtest` (likelihood-ratio tests), `hausman` (Hausman's specification test);
- You can use Stata's `predict` command, which does predictions and residual calculations.

Weights and subsets

Most Stata commands allow the use of `weights` and the `if` and `in` qualifiers. `if` and `in` were discussed earlier and have the same use when applied to regression commands as with other commands. Instead of running the regression on all observations, the `if` and `in` qualifier limit the data to a certain subset.

```
. regress cgdp pop if year < 2000
```

This command runs an ordinary least squared (OLS) regression of per capita GDP on the population level for all years prior to 2000. What is important to note is that when we want to apply post-estimation commands, for example if we want to generate the residual for each observation, that we apply the same conditions as for the regression command. If we fail to control for the selection of a subset of the data we might inadvertently conduct an out-of-sample prediction.

What we haven't discussed yet is the use of weights in Stata. There are four types of weights in Stata.

Sampling weights [`pweight`] Sampling weights are the inverse sampling probability, that is how many subjects of the population are represented by the observation. This is a very common type of weight in micro data samples. Often certain firms of different sizes have different probabilities of being in a sample, ethnic groups are often oversampled in surveys, etc. The reason is that with a fixed sampling probability the number of observations for certain types of subjects in the population is too small to yield any statistically valid results, so oversampling is applied. Sampling weights allow the user to account for those different sampling probabilities.

Analytic weights [`aweight`] Analytic weights are appropriate when you run commands on aggregated data. We might want to run a regression of county level average wages on county characteristics. Analytical weights are then the number of individuals that are used to calculate the average wage. The reason is that the more observations are used to calculate the average wage the more precisely the population average wage is estimated and therefore the more we want to rely on that information. Analytic weights should not be used in lieu of sampling weights when Stata does not allow the use of sampling weights!

Frequency and importance weights These two types of weights are less frequently used. Frequency weights are used when memory is conserved by dropping observations that have exactly the same values in all variables. Importance weights are a programmers option and beyond the need of this course (See the [U] User's Guide for more detail).

For most day to day-to-day work pweights and aweights should suffice. If you use a complex survey you might need to go further and use the survey (`svy`) commands in Stata. These allow you account for stratified and clustered sampling as well as differing sampling probabilities.

Linear regression

Stata can do a lot of general and very specialized regression commands. With Stata 10 there are now three sets of specialized commands for time series data, panel data, survey data and survival data. Stata can do a lot of fancy regressions (and most of which we will not talk about in these classes). The syntax for most of them is very similar and so we will focus on few commands in detail rather than discuss the whole list. Just so that you know the main ones, here is an abbreviated list of other regression commands that may be of interest:

anova	analysis of variance and covariance
cnreg	censored-normal regression
heckman	Heckman selection model
intreg	interval regression
ivreg	instrumental variables (2SLS) regression
newey	regression with Newey-West standard errors
prais	Prais-Winsten, Cochrane-Orcutt, or Hildreth-Lu regression
qreg	quantile (including median) regression
reg	ordinary least squares regression
reg3	three-stage least squares regression
rreg	robust regression (NOT robust standard errors)
sureg	seemingly unrelated regression
svyheckman	Heckman selection model with survey data
svyintreg	interval regression with survey data
svyivreg	instrumental variables regression with survey data
svyregress	linear regression with survey data
tobit	tobit regression
treatreg	treatment effects model
truncreg	truncated regression
xtabond	Arellano-Bond linear, dynamic panel-data estimator
xtintreg	panel data interval regression models
xtreg	fixed- and random-effects linear models
xtregar	fixed- and random-effects linear models with an AR(1) disturbance
xttobit	panel data tobit models

We will focus on this is the most basic form of linear regression. `regress` fits a model of depvar on varlist using linear regression. The `help regress` command will bring up the following instructions for using `regress`.

regress depvar [varlist] [if exp] [in range] [weight] [, level(#) beta vce(robust/cluster(varname)/bootstrap/jackknife/hc2/hc3) hascons noconstant tsscons noheader eform(string) depname(varname) mse1 plus]

Looking in the bottom of this help file will explain the options as follows:

Options

level(#) specifies the confidence level, in %, for confidence intervals of the coefficients; see help level.

beta requests that normalized beta coefficients be reported instead of confidence intervals. beta may not be specified with cluster().

vce(robust) specifies that the Huber/White/sandwich estimator of variance is to be used in place of the traditional calculation. See [U] 20.15 Obtaining robust variance estimates.

vce(hc2) and **vce(hc3)** specify an alternative bias correction for the robust variance calculation. hc2 and hc3 may not be specified with cluster(). hc2 uses u_j^2/(1-h_j) as the observation's variance estimate. hc3 uses u_j^2/(1-h_j)^2 as the observation's variance estimate. Specifying either hc2 or hc3 implies robust.

vce(cluster(varname)) specifies that the observations are independent across groups (clusters) but not necessarily independent within groups. varname specifies to which group each observation belongs; e.g., cluster(personid) in data with repeated observations on individuals. cluster() can be used with pweights to produce estimates for unstratified cluster-sampled data, but see help svyregress for a command especially designed for survey data. Specifying cluster() implies robust.

vce(bootstrap) and **vce(jackknife)** specify that the variance is estimated by resampling the data. The bootstrap resamples from the data with replacement while the jackknife consecutively deletes one observation. With both options the estimator is calculated several times which might take very long.

hascons indicates that a user-defined constant or its equivalent is specified among the independent variables. Some

caution is recommended when using this option as resulting estimates may not be as accurate as they otherwise would be. Use of this option requires "sweeping" the constant last, so the moment matrix must be accumulated in absolute rather than deviation form. This option may be safely specified when the means of the dependent and independent variables are all "reasonable" and there are not large amounts of colinearity between the independent variables. The best procedure is to view hascons as a reporting option -- estimate with and without hascons and verify that the coefficients and standard errors of the variables not affected by the identity of the constant are unchanged. If you do not understand this warning, it is best to avoid this option.

noconstant suppresses the constant term (intercept) in the regression.

tsscons forces the total sum of squares to be computed as though the model has a constant; i.e., as deviations from the mean of the dependent variable. This is a rarely used option that has an effect only when specified with nocons. It affects only the total sum of squares and all results derived from the total sum of squares.

noheader, **eform()**, **depname()**, **mse1**, and **plus** are for ado-file (i.e. self-written commands) writers; see [R] regress.

As described above, most estimation commands will follow this type of syntax but the available options will differ and so you should check the relevant help files if you wish to use these approaches. Of course, Stata has a number of defaults and so you don't need to include any options if you don't wish to change the default (though it is always good to figure out what the default is!)

Lets start with a very simple regression of GDP per worker (y) on capital-output ratio (k).

```
. regress y k

      Source |       SS       df       MS              Number of obs =     104
-------------+------------------------------           F(  1,   102) = 1110.99
       Model |  2.5465e+10     1  2.5465e+10           Prob > F      =  0.0000
    Residual |  2.3380e+09   102  22921482.3           R-squared     =  0.9159
-------------+------------------------------           Adj R-squared =  0.9151
       Total |  2.7803e+10   103   269936187           Root MSE      =  4787.6

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
           k |   .3319374   .0099587    33.33   0.000     .3121844    .3516904
       _cons |   4720.016   617.1018     7.65   0.000     3495.998    5944.035
```

There are a few points to note here:

- The first variable listed after the `regress` (or `reg` for short) command is the dependent variable, and all subsequently listed variables are the independent variables.
- Stata automatically adds the constant term or intercept to the list of independent variables (use the `noconstant` option if you want to exclude it).
- The top-left corner gives the ANOVA decomposition of the sum of squares in the dependent variable (Total) into the explained (Model) and unexplained (Residual).
- The top-right corner reports the statistical significance results for the model as a whole.
- The bottom section gives the results for the individual explanatory variables.

The `regress` command can be used with the `robust` option for estimating the standard errors using the Huber-White sandwich estimator (to correct the standard errors for heteroscedasticity):

```
. regress y k, robust


Regression with robust standard errors                Number of obs =     104
                                                      F(  1,   102) =  702.15
                                                      Prob > F      =  0.0000
                                                      R-squared     =  0.9159
                                                      Root MSE      =  4787.6

------------------------------------------------------------------------------
             |               Robust
           y |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
```

```
-------------+----------------------------------------------------------------
           k |   .3319374   .0125268    26.50   0.000      .3070905    .3567842
       _cons |   4720.016   506.2807     9.32   0.000      3715.811    5724.222
```

The coefficient estimates are exactly the same as in straightforward OLS, but the standard errors take into account heteroscedasticity. Note, the ANOVA table is deliberately suppressed as it is no longer appropriate in a statistical sense.

Sometimes you also want to allow for more general deviations from the iid-assumption on the error term. The option `cluster(group)` allows for arbitrary correlation within specified groups (see Wooldridge, "Econometrics of Cross-Section and Panel Data", chapter 4, for more details and limitations of this approach). For example, you might think that in a panel of countries, errors are correlated across time but independent across countries. Then, you should cluster standard errors on countries. In our example, we do not have a time dimension so clustering on country yields the same results as the robust option (which is a special case of the cluster option):

```
. regress y k, cluster(country)
```

Stata comes with a large amount of regression diagnostic tools, such as tests for outliers, heteroskedasticity in the errors etc. A good survey is available at http://www.ats.ucla.edu/stat/stata/webbooks/reg/chapter2/statareg2.htm. We will focus on two useful tools for detecting influential observations and looking at partial correlations. The first tool is the command `lvr2plot` (read leverage-versus-residual squared plot). This is not available after the *robust* option is used so let us revert back to the original regression:

```
. regress y k

. lvr2plot, mlabel(country)
```

This plots the leverages of all observations against their squared residuals (the option `mlabel` labels points according to the variable listed in brackets behind it). Leverage tells you how large the influence of a single observation on the estimated coefficients is. Observations with high values could potentially be driving the results obtained (especially if they also have a large squared residual) so we should check whether excluding them changes anything.

The second command is `avplot` (added-variable plot) which graphs the partial correlation between a specified regressor and the dependent variable. For this not to be simply the fitted values, we should add another variable such as human capital (h). Formally

```
. regress y k

. avplot k, mlabel(country)
```

For some very basic econometrics which also comes with the necessary Stata commands, see http://psc.maths.lancs.ac.uk/shortCourses/notes/stata/session5.pdf for model diagnostics.

Now you should play around with the regressions by adding constants, dropping variables from the regression.

Post-estimation

Once you have done your regression, you usually want to carry out some extra analysis such as forecasting or hypothesis testing. Here is a list of the most useful post-estimation commands:

Command	Description
adjust	Tables of adjusted means and proportions
estimates	Store, replay, display, ... estimation results
hausman	Hausman's specification test after model fitting
lincom	Obtain linear combinations of coefficients
linktest	Specification link test for single-equation models
lrtest	Likelihood-ratio test after model fitting
mfx	Marginal effects or elasticities after estimation
nlcom	Nonlinear combinations of estimators
predict	Obtain predictions, residuals, etc. after estimation
predictnl	Nonlinear predictions after estimation
suest	Perform seemingly unrelated estimation
test	Test linear hypotheses after estimation
testnl	Test nonlinear hypotheses after estimation
vce	Display covariance matrix of the estimators

Prediction

A number of predicted values can be obtained after all estimation commands, such as `reg`, `cnsreg`, `logit` or `probit`. The most important are the predicted values for the dependent variable and the predicted residuals. For example, suppose we run the basic regression again:

```
. regress y k h
. predict y_hat                    /* predicted values for dependent var */
. predict r, residual              /* predicted residuals */
```

Stata creates new variables containing the predicted values, and these variables can then be used in any other Stata command, e.g. you can graph a histogram of the residuals to check for normality.

If we run a selected regression (e.g. just using OECD countries) and then wish to know how well this regression fits, we could run the following commands:

```
regress y k h       if oecd==1

predict y_hat_oecd  if oecd==1
predict r_oecd      if oecd==1, residual
```

The `if` statements are only necessary if you are running the analysis on a subset of dataset currently loaded into Stata. If you want to make out-of-sample predictions, just drop the `if` statements in the `predict` commands.

```
predict y_hat_oecd_full
predict r_oecd_full, residual
```

Hypothesis testing

The results of each estimation automatically include for each independent variable a two-sided t-test (for linear regressions) and a z-test (for regressions such as logit or probit) on the null hypothesis that the "true" coefficient is equal to zero. You can also perform an F-test or χ_2 test on this hypothesis using the `test` command:

```
. regress y k h y1985 ya
. test y1985  /*since Stata defaults to comparing the listed terms to zero, you can
simply use the variable*/

 ( 1)  y1985 = 0

       F(  1,    63) =   15.80
            Prob > F =   0.0002
```

The F-statistic with 1 numerator and 63 denominator degrees of freedom is 15.80. The *p*-value or significance level of the test is basically zero (up to 4 digits at least), so we can reject the null hypothesis even at the 1% level – y1985 is significantly different

from zero. Notice that, since the critical values of the *F*-distribution and the *F*-statistic with 1 numerator degree of freedom is identical to the square of the same values from the t-distribution, so the F-test result is the same as the result of the t-test. Also the *p*-values associated with each test agree.

You can perform any test on linear hypotheses about the coefficients, such as:

```
. test y1985=0.5           /* test coefficient on y1985 equals 0.5 */
. test y1985 h             /* test coefficients on y1985 & h jointly zero */
. test y1985+h=-0.5        /* test coefficients on y1985 & h sum to -0.5 */
. test y1985=h             /* test coefficients on y1985 & h are the same */
```

With many Stata commands, you can refer to a list of variables using a hyphen, e.g. `desc k- ya` gives descriptive statistics on exp, ya and every other variable on the list between them. However, the test command interprets the hyphen as a minus, and gets confused because it thinks you are typing a formula for it to test. If you want to test a long list of variables, you can use the `testparm` command (but remember to use the `order` command to bring the variables in the right order first)

```
. order k h y1985 ya

. testparm k-ya

 ( 1)  k = 0
 ( 2)  h = 0
 ( 3)  y1985 = 0
 ( 4)  ya = 0

       F(  4,     63) =  370.75
            Prob > F =    0.0000
```

Extracting results

We have already seen how the `predict` command can be used to extract predicted values from Stata's internal memory for use in subsequent analyses. Using the `generate` command, we can also extract other results following a regression, such as estimated coefficients and standard errors:

```
regress y k h y1985 ya, robust
gen b_cons=_b[_cons]       /* beta coefficient on constant term */
gen b_k=_b[k]              /* beta coefficient on GDP60 variable */
gen se_k=_se[k]            /* standard error */
```

You can `tabulate` the new variables to confirm that they do indeed contain the results of the regression. You can then use these new variables in subsequent Stata commands, e.g. to create a variable containing t-statistics:

```
. gen t_k=b_k/se_k
```

or, more directly:

```
. gen t_k=_b[k]/_se[k]
```

Stata stores extra results from estimation commands in `e()`, and you can see a list of what exactly is stored using the `ereturn list` command:

```
. regress y k h y1985 ya, robust
. ereturn list

scalars:
                  e(N) =  68
              e(df_m) =  4
              e(df_r) =  63
                  e(F) =  273.7198124833108
                 e(r2) =  .9592493796249692
               e(rmse) =  3451.985251440704
                e(mss) =  17671593578.3502
                e(rss) =  750720737.0983406
               e(r2_a) =  .9566620386487768
                 e(ll) =  -647.8670640006279
               e(ll_0) =  -756.6767273270843

macros:
            e(depvar) : "y"
               e(cmd) : "regress"
           e(predict) : "regres_p"
             e(model) : "ols"
           e(vcetype) : "Robust"

matrices:
                 e(b) :  1 x 5
                 e(V) :  5 x 5

functions:
            e(sample)
```

`e(sample)` is a temporary variable that is 1 if an observation was used by the last regression command run and 0 otherwise. It is a useful tool to have. Earlier we ran the following commands:

```
regress  y k h if oecd==1
predict y_hat_oecd if oecd==1
```

but in the event that the "if" statement is complex, we may wish to simple tell Stata to predict using the same sample it used in the regression. We can do this using the e(sample):

```
predict y_hat_oecd if e(sample)
```

`e(N)` stores the number of observations, `e(df_m)` the model degrees of freedom, `e(df_r)` the residual degrees of freedom, `e(F)` the F-statistic, and so on. You can extract any of these into a new variable:

```
. gen residualdf=e(df_r)
```

And you can then use this variable as usual, e.g. to generate p-values:

```
. gen p_k=tprob(residualdf,t_k)
```

The `tprob` function uses the two-tailed cumulative Student's t-distribution. The first argument in parenthesis is the relevant degrees of freedom, the second is the t-statistic.

In fact, most Stata commands – not just estimation commands – store results in internal memory, ready for possible extraction. Generally, the results from other commands – that is commands that are not estimation commands – are stored in `r()`. You can see a list of what exactly is stored using the `return list` command, and you can extract any you wish into new variables:

```
. sum y

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
           y |       105    18103.09    16354.09   630.1393   57259.25


. return list


scalars:
                  r(N) =  105
              r(sum_w) =  105
               r(mean) =  18103.08932466053
                r(Var) =  267456251.2136306
                 r(sd) =  16354.08973968379
                r(min) =  630.1392822265625
                r(max) =  57259.25
                r(sum) =  1900824.379089356

. gen mean_y=r(mean)
```

Note that the last command will give exactly the same results as `egen mean_y=mean(y)`.

OUTREG2 – the ultimate tool in Stata/Latex or Word friendliness?

There is a tool which will automatically create excel, word or latex tables or regression results and it will save you loads of time and effort. It formats the tables to a journal standard and was originally just for word (`outreg`) but now the updated version will also do tables for latex also.

There are other user-written commands that might be helpful, outtable, outtex, estout, mat2txt, etc. just find the one that suit your purpose best.

However, it does not come as a standard tool and so before we can use it, we must learn how to install extra ado files (not to be confused with running our own do files).

Extra commands on the net

Looking for specific commands

If you are trying to perform an exotic econometric technique and cannot find any useful command in the Stata manuals, you may have to programme in the details yourself. However, before making such a rash move, you should be aware that, in addition to the huge list of commands available in the Stata package and listed in the Stata manuals, a number of researchers have created their own extra commands. These extra commands range from the aforementioned exotic econometric techniques to mini time-saving routines. For example, the command `outreg2`.

You need to first locate the relevant command and then install it into your copy of Stata. The command can be located by trying different searches, e.g. to search for a command that formats the layout of regression results, I might search for words like "format" or "table":

```
. search format regression table

Keyword search

Keywords:  format regression table
  Search:  (1) Official help files, FAQs, Examples, SJs, and STBs

Search of official help files, FAQs, Examples, SJs, and STBs


FAQ     Can I make regression tables that look like those in journal articles?
. . . . . . . . . . . . . . . . . .  UCLA Academic Technology Services
5/01    http://www.ats.ucla.edu/stat/stata/faq/outreg.htm

STB-59  sg97.3  . . . . . . . . . . . . . Update to formatting regression output
(help outreg if installed)  . . . . . . . . . . . . . . . J. L. Gallup
1/01    p.23; STB Reprints Vol 10, p.143
small bug fixes

STB-58  sg97.2  . . . . . . . . . . . . . Update to formatting regression output
(help outreg if installed)  . . . . . . . . . . . . . . . J. L. Gallup
11/00   pp.9--13; STB Reprints Vol 10, pp.137--143
update allowing user-specified statistics and notes, 10%
asterisks, table and column titles, scientific notation for
coefficient estimates, and reporting of confidence interval
and marginal effects

STB-49  sg97.1  . . . . . . . . . . . . . . . . . . . . . . Revision of outreg
(help outreg if installed)  . . . . . . . . . . . . . . . J. L. Gallup
5/99    p.23; STB Reprints Vol 9, pp.170--171
updated for Stata 6 and improved

STB-46  sg97  . . . . . . .  Formatting regression output for published tables
(help outreg if installed)  . . . . . . . . . . . . . . . J. L. Gallup
11/98   pp.28--30; STB Reprints Vol 8, pp.200--202
takes output from any estimation command and formats it as
in journal articles

(end of search)
```

You can read the FAQ by clicking on the blue hyperlink. This gives some information on the command. You can install the command by first clicking on the blue command name (here `sg97.3`, the most up-to-date version) and, when the pop-up window appears, clicking on the install hyperlink. Once installed, you can create your table and then use the command `outreg2` as any other command in Stata. The help file will tell you the syntax.

However, I mentioned `outreg2` and this has not appeared here, so I may need to update more.

Checking for updates in general

New Stata routines and commands appear all the time and existing ones get updates. A simple way to keep up-to-date with any changes is to use the `update` commands. The first step is to check when your version was last updated:

```
. update

Stata executable
    folder:               \\st-server5\stata10$\
    name of file:         wsestata.exe
    currently installed:  11 Aug 2008

Ado-file updates
    folder:               \\st-server5\stata10$\ado\updates\
    names of files:       (various)
    currently installed:  22 Sep 2008

Utilities updates
    folder:               \\st-server5\stata10$\utilities
    names of files:       (various)
    currently installed:  27 May 2008

Recommendation
    Type -update query- to compare these dates with what is available from
    http://www.stata.com.
```

Stata consists of basically two sets of files, the executable file and the ado-files (the utilities are new and so far include only one program that is called internally by Stata). The former is the main programme while the latter present the different Stata commands and routines. In order to check whether there are any more up-to-date versions use the `update query` command:

```
. update query
(contacting http://www.stata.com)

Stata executable
    folder:               \\st-server5\stata10$\
    name of file:         wsestata.exe
    currently installed:  11 Aug 2008
    latest available:     11 Aug 2008

Ado-file updates
    folder:               \\st-server5\stata10$\ado\updates\
    names of files:       (various)
    currently installed:  22 Sep 2008
    latest available:     22 Sep 2008

Utilities updates
    folder:               \\st-server5\stata10$\utilities
    names of files:       (various)
    currently installed:  27 May 2008
    latest available:     27 May 2008

Recommendation
    Do nothing; all files up to date.
```

It looks like my executable and ado files are okay. If I needed to update my ado files, Stata would have told me to type update ado which would lead to the following type of update:

```
. update ado
(contacting http://www.stata.com)

Ado-file update log
    1.  verifying \\st-server5\stata10$\ado\updates\ is writeable
    2.  obtaining list of files to be updated
    3.  downloading relevant files to temporary area
downloading checksum.hlp
            ...
downloading varirf_ograph.ado
downloading whatsnew.hlp
    4.  examining files
    5.  installing files
```

```
    6.  setting last date updated

Updates successfully installed.

Recommendation
    See help whatsnew to learn about the new features
```

Finally, to learn about the new features installed, simply type `help whatsnew`.

But we know that `outreg2` exists so how do we find it to install? Well, type `outreg2` into google to convince yourself that it exists. Then type:

```
search outreg2, net

Web resources from Stata and other users

(contacting http://www.stata.com)

1 package found (Stata Journal and STB listed first)
-------------------------------------------------

outreg2 from http://fmwww.bc.edu/RePEc/bocode/o
    'OUTREG2': module to arrange regression outputs into an illustrative table
    / outreg2 provides a fast and easy way to produce an illustrative / table
    of regression outputs. The regression outputs are produced / piecemeal and
    are difficult to compare without some type of / rearrangement. outreg2

(click here to return to the previous screen)

(end of search)
```

Click on the blue link and follow instructions to install the ado file and help.

Most additional commands that you will find are available from the Statistical Software Archive (SSC) and can be installed by typing `ssc install` followed by the name of the command. If the command you want to install is not available from SSC but elsewhere on the internet you can use the `net install` command. But you should be wary of the source of the commands you install and always test them before starting to use them.

```
. ssc install outreg2
checking outreg2 consistency and verifying not already installed...
installing into c:\ado\plus\...
installation complete.
```

Now using the help, try to figure out the syntax and then run the regressions from earlier in your do file but create a table which places the results of, for example, 6 regressions next to each other in either word or latex.

Problems when installing additional commands on shared PCs

When you are not using your private PC or Laptop but a shared PC, for example in the library, you might run into problems updating Stata or installing additional commands.

```
. ssc install outreg2
checking outreg2 consistency and verifying not already installed...
installing into c:\ado\plus\...
could not rename c:\ado\plus\next.trk to c:\ado\plus\stata.trk
could not rename c:\ado\plus\backup.trk to c:\ado\plus\stata.trk
r(699);
```

This error occurs when someone else installed additional commands for Stata on this PC before. The reason is simply that Windows allows only the “owner” and administrators to change a file. To keep track of installed commands, Stata has to change some files. If you were not the first person to install additional commands (or update) on the PC you are using, these tracking files will belong to someone else and you cannot change them.

But there is no reason for despair as there are several workarounds for this problem. The first solution is not to install the

command but to run the ado file so the command becomes available temporarily. When you install a command it will remain available even if you restart Stata. If on the other hand you only run the associated ado file, the command will only work until you exit the current Stata session.

```
. cap run http://fmwww.bc.edu/repec/bocode/o/outreg2.ado
```

When typing in the commands interactively the `capture` command is not necessary, but if you include the line as part of your do-file you should use it, since you will get an error message when you try to load the command into memory when you have already done so. The advantage of this method is that you will have to explicitly load all the commands you need into memory and so even if you change the PC you use often your do-files will still work. The disadvantage is that you will need internet access and that the help for the commands is not installed and cannot be accessed from Stata directly by typing `help outreg2`. You can however access the help for the command via the web repository.

```
. view http://fmwww.bc.edu/repec/bocode/o/outreg2.hlp
```

The other method to solve the renaming problem is to change the path where you install updates and additional commands. For additional commands this is the "Plus" path. You can check the current path with the `sysdir` command and change it by typing `sysdir set PLUS "H:\mypath"`.

```
. sysdir
   STATA:  \\st-server5\stata10$\
 UPDATES:  \\st-server5\stata10$\ado\updates\
    BASE:  \\st-server5\stata10$\ado\base\
    SITE:  \\st-server5\stata10$\ado\site\
    PLUS:  c:\ado\plus\
PERSONAL:  c:\ado\personal\
OLDPLACE:  c:\ado\

. sysdir set PLUS "H:\ECStata"

. sysdir
   STATA:  \\st-server5\stata10$\
 UPDATES:  \\st-server5\stata10$\ado\updates\
    BASE:  \\st-server5\stata10$\ado\base\
    SITE:  \\st-server5\stata10$\ado\site\
    PLUS:  H:\ECStata\
PERSONAL:  c:\ado\personal\
OLDPLACE:  c:\ado\
```

Exporting results "by hand"

While export commands that were written by other users might give you nice looking tables, they might not be versatile enough for your needs, or give you too much output, i.e. you might not be interested in all results, but just one number from several regressions.

What you can do in these cases is export information "by hand", i.e. use Stata's in-built functions to write information on the hard drive. The easiest way is to simply save a dataset containing the information you need. But that will not do you any good if you want to construct a table in Excel or Latex. What you can do is write an ASCII file that contains the numbers you want, and maybe some additional information.

```
. file open tmpHolder using "H:\ECStata\MyOutputFile.txt", write replace text
(note: file H:\ECStata\MyOutputFile.txt not found)
```

The `file` command handles the export to a text file. For this it uses a link to a file on the hard drive. The link has to be named (since we could in theory open several links at the same time) and will be referred to by its handle after being opened. The handle here is "tmpHolder". In the options we specify that we want to open a file for writing, if the file already exists it should be replaced and we want the output to be plain text. The file we use to save the results is specified after the `using` qualifier.

To write some text into the output file we write:

```
. file write tmpHolder "This is the header for my Output file" _n
```

Again we use the `file` command but now with `write` as subcommand. After the subcommand we specify which link should be used by providing a handle (here `tmpHolder`) and follow by the text we want to write into the text file. The `_n` is the newline command and so the next text will start in the second line of the output file. Now we don't want to export text only but more importantly numbers. Let us say the mean and the variance of a variable:

```
. sum y

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
           y |       105    18103.09    16354.09   630.1393   57259.25

. file write tmpHolder "y," (r(mean)) "," (r(Var)) _n
```

If we refer to some return values of descriptive or estimation commands we need to put them between parentheses, otherwise Stata will regard `r()` as text and just write the text and not the value of `r()` into the output file.

When we have finished writing lines into an output file we close it (i.e. severe the link) and save it by using the `close` subcommand.

```
. file close tmpHolder
```

More Estimation

There are a host of other estimation techniques which Stata can handle efficiently. Below is a brief look at a couple of these. Further information on these or any other technique you may be interested in can be obtained from the Stata manuals.

Constrained linear regression

Suppose the theory predicts that the coefficients for `REV` and `ASSASS` should be identical. We can estimate a regression model where we constrain the coefficients to be equal to each other. To do this, first define a constraint and then run the `cnsreg` command:

```
. constraint define 1 rev=assass        /* constraint is given the number 1 */
. cnsreg gr6085 lgdp60 sec60 prim60 gcy rev assass pi60 if year==1990, constraint(1)

Constrained linear regression                          Number of obs =     100
                                                       F(  6,    93) =   12.60
                                                       Prob > F      =  0.0000
                                                       Root MSE      =  1.5025
 ( 1) - rev + assass = 0
------------------------------------------------------------------------------
      gr6085 |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      lgdp60 |  -1.617205   .2840461    -5.69   0.000    -2.181264   -1.053146
       sec60 |   .0429134    .012297     3.49   0.001     .0184939    .0673329
      prim60 |   .0352023    .007042     5.00   0.000     .0212183    .0491864
         gcy |  -.0231969    .017786    -1.30   0.195    -.0585165    .0121226
         rev |  -.2335536   .2877334    -0.81   0.419     -.804935    .3378279
      assass |  -.2335536   .2877334    -0.81   0.419     -.804935    .3378279
        pi60 |  -.0054616   .0024692    -2.21   0.029    -.0103649   -.0005584
       _cons |    12.0264   2.073177     5.80   0.000     7.909484    16.14332
------------------------------------------------------------------------------
```

Notice that the coefficients for `REV` and `ASSASS` are now identical, along with their standard errors, t-stats, etc. We can define and apply several constraints at once, e.g. constrain the `lGDP60` coefficient to equal –1.5:

```
. constraint define 2 lgdp60=-1.5
. cnsreg gr6085 lgdp60 sec60 prim60 gcy rev assass pi60 if year==1990, constraint(1 2)
```

Dichotomous dependent variable

When the dependent variable is dichotomous (zero/one), you can run a Linear Probability Model using the `regress` command. You may also want to run a `logit` or a `probit` regression. The difference between these three models is the assumption that you make about the probability distribution of the latent dependent variable (LPM assumes an identity function, Logit a logistic distribution function, and Probit a normal distribution function).

For the sake of trying out these commands, let us "explain" why a country is an OECD member using a logit regressions:

```
. logit OECD lrgdpl if year==1990

Iteration 0:   log likelihood = -63.180951
...
Iteration 7:   log likelihood =  -21.99139

Logit estimates                                   Number of obs   =        135
                                                  LR chi2(1)      =      82.38
                                                  Prob > chi2     =     0.0000
Log likelihood =  -21.99139                       Pseudo R2       =     0.6519

------------------------------------------------------------------------------
        OECD |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      lrgdpl |    4.94118   1.119976     4.41   0.000     2.746067    7.136292
       _cons |  -47.38448    10.7335    -4.41   0.000    -68.42176    -26.3472
------------------------------------------------------------------------------
```

Panel Data

If you are lucky enough to have a panel dataset, you will have data on n countries/people/firms/etc, over t time periods, for a total of n × t observations. If t is the same for each country/person/firm then the panel is said to be balanced; but for most things Stata is capable of working out the optimal/maximum dataset available. There are a few things to note before using panel data commands:

1. Panel data should be kept in long form (with separate person and time variables). However, sometimes your data may be in wide form and needs to be converted to long form using the `reshape` command.

2. You have to declare your data a panel. One way to do this is using the `xtset` command (the previously used commands `iis` and `tss` are outdated as of Stata 10). To do this, you need two indicator variables, indicating the unit (panelvar) and time (timevar) dimensions of your panel. In our case, these are simply `year` and `country`. Note that panel dimensions cannot be string variables so you should first `encode` country. Once you have done this, use the `xtset` command:

```
. encode country, gen(country_no)
. xtset country_no year
```

You are now free to use Stata's panel data commands, although I will only make use of a few main ones **(bolded)**:

xtdes	**Describe pattern of xt data**
xtsum	**Summarize xt data**
xttab	**Tabulate xt data**
xtdata	Faster specification searches with xt data
xtline	Line plots with xt data
xtreg	**Fixed-, between- and random-effects, and population-averaged linear models**
xtregar	Fixed- and random-effects linear models with an AR(1) disturbance
xtgls	Panel-data models using GLS
xtpcse	OLS or Prais-Winsten models with panel-corrected standard errors
xtrchh	Hildreth-Houck random coefficients models
xtivreg	Instrumental variables and two-stage least squares for panel-data models
xtabond	Arellano-Bond linear, dynamic panel data estimator
xttobit	Random-effects tobit models
xtintreg	Random-effects interval data regression models
xtlogit	Fixed-effects, random-effects, & population-averaged logit models
xtprobit	Random-effects and population-averaged probit models
xtcloglog	Random-effects and population-averaged cloglog models
xtpoisson	Fixed-effects, random-effects, & population-averaged Poisson models
xtnbreg	Fixed-effects, random-effects, & population-averaged negative binomial models
xtgee	Population-averaged panel-data models using GEE

Describe pattern of xt data

xtdes is very useful to see if your panel is actually balanced or whether there is large variation in the number of years for which each cross-sectional unit is reporting.

```
. xtdes

country_no:  1, 2, ..., 168                                 n =        168
    year:  1950, 1951, ..., 2000                            T =         51
           Delta(year) = 1; (2000-1950)+1 = 51
           (country_no*year uniquely identifies each observation)
```

```
Distribution of T_i:   min      5%     25%       50%       75%     95%     max
                        51      51      51        51        51      51      51

     Freq.  Percent    Cum. |  Pattern
 ---------------------------+-----------------------------------------------------
      168    100.00  100.00 |  111111111111111111111111111111111111111111111111111
 ---------------------------+-----------------------------------------------------
      168    100.00         |  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Of course, in our sample there are year entries for every country and every year, but much of the data is missing. Looking at the patterns given that GDP per capita data exists tells a much more mixed story. Some 50 countries have data for all years, but many other variant patterns are evident (especially samples than begin after 1950).

```
. xtdes if cgdp!=.

country_no:  1, 2, ..., 168                                 n =        168
    year:  1950, 1951, ..., 2000                            T =         51
           Delta(year) = 1; (2000-1950)+1 = 51
           (country_no*year uniquely identifies each observation)

Distribution of T_i:   min      5%     25%       50%       75%     95%     max
                         1       1      15        41        51      51      51

     Freq.  Percent    Cum. |  Pattern
 ---------------------------+-----------------------------------------------------
       50     29.76   29.76 |  111111111111111111111111111111111111111111111111111
       29     17.26   47.02 |  ..........11111111111111111111111111111111111111111
       14      8.33   55.36 |  ..............................................1....
        6      3.57   58.93 |  ........................................11111111111
        6      3.57   62.50 |  .1111111111111111111111111111111111111111111111111
        5      2.98   65.48 |  ..........11111111111111111111111111111111111111.
        4      2.38   67.86 |  ............................1111111111111111111111
        4      2.38   70.24 |  .........11111111111111111111111111111111111111111
        4      2.38   72.62 |  .....11111111111111111111111111111111111111111111
       46     27.38  100.00 | (other patterns)
 ---------------------------+-----------------------------------------------------
      168    100.00         |  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Summarize xt data

`xtsum` is similarly very useful and can be used in the same way that sum is used for non-panel data.

```
. xtsum pop cgdp

Variable         |      Mean   Std. Dev.       Min        Max |    Observations
-----------------+--------------------------------------------+----------------
pop      overall |  31252.47   108217.8      40.82    1258821 |     N =    5847
         between |             89099.92      42.48   913862.3 |     n =     168
         within  |             28391.67  -313609.8   405753.2 | T-bar = 34.8036
                 |                                           |
cgdp     overall |  7.467798   1.272928   4.417209   10.79891 |     N =    5847
         between |             1.052756   5.527193   10.05989 |     n =     168
         within  |             .8357679   5.050297   9.835527 | T-bar = 34.8036
```

This tables tells us the minimum and maximum, standard deviation and mean (in the overall case) of our selected variables (pop and cgdp) in three ways that are of interest:

1. the overall sample
2. the between sample – i.e. $x(bar)_i$
3. the within sample – i.e. x_{it} - $x(bar)_i$ - x(global bar)

Tabulate xt data

xttab is also a generalisation of the tabulate command for panel data and will show overall, within and between variation.

```
. xttab  G7

                  Overall             Between             Within
       G7 |    Freq.  Percent      Freq.  Percent        Percent
----------+-----------------------------------------------------
        0 |    8211     95.83       161     95.83         100.00
        1 |     357      4.17         7      4.17         100.00
----------+-----------------------------------------------------
    Total |    8568    100.00       168    100.00         100.00
                            (n = 168)
```

Panel regressions

xtreg is a generalisation of the regress commands. As with the summary data above, we can make use of the information in the cross-section (between) and also in the time-series (within). Also, as per your econometrics training, Stata allows you to run fixed-effects (fe), random effects (re) and between estimators using `xtreg`. More complicated estimation (such as the Arellano-Bond dynamic GMM estimator) have specific xt estimation commands.

Fixed Effects Regression

Fixed effects regression controls for unobserved, but constant, variation across the cross-sectional units. It is equivalent to including a dummy for each country/firm in our regression. Let us use the `xtreg` command with the `fe` option:

```
. xtreg grgdpch gdp60 openk kc kg ki, fe

Fixed-effects (within) regression               Number of obs      =      5067
Group variable (i): country_no                  Number of groups   =       112

R-sq:  within  = 0.0164                         Obs per group: min =         2
       between = 0.2946                                        avg =      45.2
       overall = 0.0306                                        max =        51

                                                F(4,4951)          =     20.58
corr(u_i, Xb)  = -0.4277                        Prob > F           =    0.0000

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  (dropped)
       openk |  -.0107672   .0042079    -2.56   0.011    -.0190166   -.0025178
          kc |  -.0309774   .0089545    -3.46   0.001    -.0485322   -.0134225
          kg |  -.0733306   .0147568    -4.97   0.000    -.1022604   -.0444007
          ki |   .1274592   .0178551     7.14   0.000     .0924552    .1624631
       _cons |   4.425707   .7451246     5.94   0.000     2.964933    5.886482
-------------+----------------------------------------------------------------
     sigma_u |  1.6055981
     sigma_e |  6.4365409
         rho |  .05858034   (fraction of variance due to u_i)
------------------------------------------------------------------------------
F test that all u_i=0:     F(111, 4951) =     1.82           Prob > F = 0.0000
```

Notice that gdp60, the log of GDP in 1960 for each country, is now dropped as it is constant across time for each country and so is subsumed by the country fixed-effect.

Between Effects

We can now use the xtreg command with the `be` option. This is equivalent to running a regression on the dataset of means by cross-sectional identifier. As this results in loss of information, between effects are not used much in practice.

```
. xtreg grgdpch gdp60 openk kc kg ki, be

Between regression (regression on group means)  Number of obs      =      5067
Group variable (i): country_no                  Number of groups   =       112

R-sq:  within  = 0.0100                         Obs per group: min =         2
```

```
       between = 0.4575                                        avg =      45.2
       overall = 0.0370                                        max =        51

                                                F(5,106)            =     17.88
sd(u_i + avg(e_i.))=  1.277099                  Prob > F            =    0.0000

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5185608   .1776192    -2.92   0.004    -.8707083   -.1664134
       openk |   .0008808   .0029935     0.29   0.769    -.0050541    .0068156
          kc |  -.0151328    .009457    -1.60   0.113    -.0338822    .0036166
          kg |  -.0268036   .0149667    -1.79   0.076    -.0564765    .0028693
          ki |   .1419786   .0213923     6.64   0.000     .0995662     .184391
       _cons |   4.657591   1.587533     2.93   0.004     1.510153     7.80503
------------------------------------------------------------------------------
```

Random Effects

The command for a linear regression on panel data with random effects in Stata is `xtreg` with the re option. Stata's random-effects estimator is a weighted average of fixed and between effects.

```
. xtreg grgdpch gdp60 openk kc kg ki, re

Random-effects GLS regression                   Number of obs      =      5067
Group variable (i): country_no                  Number of groups   =       112

R-sq:  within  = 0.0143                         Obs per group: min =         2
       between = 0.4235                                        avg =      45.2
       overall = 0.0389                                        max =        51

Random effects u_i ~ Gaussian                   Wald chi2(5)       =    159.55
corr(u_i, X)       = 0 (assumed)                Prob > chi2        =    0.0000

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5661554   .1555741    -3.64   0.000    -.8710751   -.2612356
       openk |  -.0012826   .0024141    -0.53   0.595    -.0060142    .003449
          kc |  -.0270849   .0061971    -4.37   0.000     -.039231   -.0149388
          kg |  -.0506839   .0101051    -5.02   0.000    -.0704895   -.0308783
          ki |   .1160396   .0127721     9.09   0.000     .0910067    .1410725
       _cons |   6.866742   1.239024     5.54   0.000       4.4383    9.295185
-------------+----------------------------------------------------------------
     sigma_u |  .73122048
     sigma_e |  6.4365409
         rho |  .01274156   (fraction of variance due to u_i)
------------------------------------------------------------------------------
```

Choosing Between Fixed and Random Effects

Choosing between FE and RE models is usually done using a Hausman test, and this is easily completed in Stata using the Hausman command. To run a Hausman test we need to run the RE and FE models and save the results using the `store` command. We then instruct Stata to retrieve the 2 sets of results and carry-out the test.

For example, using the same estimates as above, we can write the following in our do file:

```
xtreg grgdpch gdp60 openk kc kg ki, fe
estimates store fe
xtreg grgdpch gdp60 openk kc kg ki, re
estimates store re
hausman fe re

                 ---- Coefficients ----
             |      (b)          (B)            (b-B)     sqrt(diag(V_b-V_B))
```

```
             |       fe           re         Difference          S.E.
-------------+----------------------------------------------------------------
       openk |   -.0107672    -.0012826       -.0094846        .0034465
          kc |   -.0309774    -.0270849       -.0038924        .0064637
          kg |   -.0733306    -.0506839       -.0226467        .0107541
          ki |    .1274592     .1160396        .0114196        .0124771
------------------------------------------------------------------------------
                           b = consistent under Ho and Ha; obtained from xtreg
            B = inconsistent under Ha, efficient under Ho; obtained from xtreg

    Test:  Ho:  difference in coefficients not systematic

                  chi2(4) = (b-B)'[(V_b-V_B)^(-1)](b-B)
                          =       20.15
                Prob>chi2 =      0.0005
```

As described in the results, the null hypothesis is that there is no difference in the coefficients estimated by the efficient RE estimator and the consistent FE estimator. If there is no difference, then use the RE estimator – i.e. if the statistic is insignificant that is the Prob>chi2 is larger than .05 or .01. Otherwise, you should use FE, or one of the other solutions for unobserved heterogeneity as outlined in your Econometrics lectures.

Time series data

Stata has a very particular set of functions that control time series commands. But in order to use these commands, you must ensure that you tell Stata. Similar to the panel data commands above, we can do this using the `tsset` For example:

```
tsset datevar
```

Once you have done this, you are free to use the time series commands – I present a selection of these below (type help time for the full list):

tsset	Declare a dataset to be time-series data
tsfill	Fill in missing times with missing observations in time-series data
tsappend	Add observations to a time-series dataset
tsreport	Report time-series aspects of a dataset or estimation sample
arima	Autoregressive integrated moving-average models
arch	Autoregressive conditional heteroskedasticity (ARCH) family of estimators
tssmooth_ma	Moving-average filter
tssmooth_nl	Nonlinear filter
corrgram	Tabulate and graph autocorrelations
xcorr	Cross-correlogram for bivariate time series
dfuller	Augmented Dickey-Fuller unit-root test
pperron	Phillips-Perron unit-roots test
archlm	Engle's LM test for the presence of autoregressive conditional heteroskedasticity
var	Vector autoregression models
svar	Structural vector autoregression models
varbasic	Fit a simple VAR and graph impulse-response functions
vec	Vector error-correction models
varsoc	Obtain lag-order selection statistics for VARs and VECMs
varstable	Check the stability condition of VAR or SVAR estimates
vecrank	Estimate the cointegrating rank using Johansen's framework
irf create	Obtain impulse-response functions and FEVDs
vargranger	Perform pairwise Granger causality tests after var or svar
irf graph	Graph impulse-response functions and FEVDs
irf cgraph	Combine graphs of impulse-response functions and FEVDs
irf ograph	Graph overlaid impulse-response functions and FEVDs

All of these can be implemented where appropriate by using the help function, manuals and internet resources (or colleagues know-how).

Stata Date and Time-series Variables

However, one of the issues with time series in Stata, and something that particularly challenges new users of Stata, is the data format used in the program. Therefore, I below provide some more advanced notes on this specialist topic.

The key thing is that there are 2 possible types of entry – date entries (which work in general for storing dates in Stata) and time-series entries (which are useful when we are not using daily data).Stata stores dates as the number of elapsed periods since January 1, 1960. When using a data-set that is not daily data, we want to use Stata's time-series function rather than the date function – the reason is that the dates for quarterly data will be about 3 months apart but the number of days between them will vary so telling Stata to go from Q1 to Q2 will involve changing the date from (for example) January 1st to April 1st – which is either 90 days or 91 days depending on whether it is a leap-year. Obviously our life would be easier if we could just tell Stata that one entry is Q1, and the other entry is Q2. For example, if we want to take first-differences between quarters, or even more trickily, if we wanted to take seasonal differences – Q1 minus Q1 from previous year.

Therefore when we have a variable that identifies the time-series elements of a dataset, we must tell Stata what type of data we are using – is it daily, weekly, monthly, quarterly, half-yearly or yearly. Therefore, if you use daily data it will be the number of elapsed days since January 1st 1960 (which is therefore zero), but if you use quarterly data, it is the number of elapsed quarters

since 1960 Q1. The following table explains the different formats:

Format	Description	Beginning	+1 Unit	+2 Units	+3 Units
%td	daily	01jan1960	02jan1960	03Jan1960	04Jan1960
%tw	weekly	week 1, 1960	week 2, 1960	week 3, 1960	week 4, 1960
%tm	monthly	Jan, 1960	Feb, 1960	Mar, 1960	Apr, 1960
%tq	quarterly	1st qtr, 1960	2nd qtr, 1960	3rd qtr, 1960	4th qtr, 1960
%th	half-yearly	1st half, 1960	2nd half, 1960	1st half, 1961	2nd half, 1961
%ty	yearly	1960	1961	1962	1963

Obviously, what you tell Stata here is highly important; we will see how to convert our data into Stata dates in a moment, but for now assume that we have a Stata date for January 1, 1999 – this is an elapsed date of 14245 (the number of days since January 1st 1960). If we were to use this number as different types of time-series data, then there would be very different outcomes as shown in the following table:

Daily	**Weekly**	**Quarterly**	**Half-yearly**	**Yearly**
%td	%tw	%tq	%th	%ty
01 Jan 1999	2233 W50	5521 Q2	9082 H2	-

These dates are so different because the elapsed date is actually the number of weeks, quarters, etc., from the first week, quarter, etc of 1960. The value for %ty is missing because it would be equal to the year 14,245 which is beyond what Stata can accept.

Therefore if we have a date format of 14245, but we want this to point to quarterly data, then we would need to convert it using special Stata functions. These functions translate from %td dates:

`wofd(varname)`	daily to weekly
`mofd(varname)`	daily to monthly
`qofd(varname)`	daily to quarterly
`yofd(varname)`	daily to yearly

Looking up in help can also show how to convert numbers between other formats (`help dates_and_times`).

Getting dates into Stata format

This section covers how we get an existing date or time variable into the Stata format for dates – from here we can rewrite it as quarterly, monthly, etc… using the above commands. There are 3 different considerations depending on how your existing "date variable" is set up:

1. Date functions for single string variables
   For example, your existing date variable is called raw_date and is of the form "20mar1999" – then it is said to be a single string variable (the string must be easily separated into its components so strings like "20mar1999" and "March 20, 1999" are acceptable). If you have a string like "200399", we would need to convert it to a numeric variable first and then use technique 3 below.
   To convert the raw_date variable to a daily time-series date, we use the command:

   ```
   gen daily=date(raw_date,"dmy")
   ```

   The "dmy" portion indicates the order of the day, month and year in the variable; so if the variable was of the form values been coded as "March 20, 1999" we would have used "mdy" instead.
   The year must have 4 digits or else it returns missing values – therefore if the original date only has two digits, we place the century before the "y.":

   ```
   gen daily=date(raw_date,"dm19y")
   ```

   Or, if we have non-daily dates, we can use the following functions:

   ```
   weekly(stringvar,"wy")
   monthly(stringvar,"my")
   ```

```
quarterly(stringvar,"qy")
halfyearly(stringvar,"hy")
yearly(stringvar,"y")
```

For example, if our data is 2002Q1, then

```
gen quarterly= quarterly(raw_data,"yq")
```

will get our elapsed quarters since 1960 Q1.

2. Date functions for partial date variables
   If there are separate variables for each element of the date; for example:

month	**day**	**year**
7	11	1948
1	21	1952
11	2	1994
8	12	1993

We can use the mdy() function to create an elapsed Stata date variable. The month, day and year variables must be numeric. Therefore we can write:

```
gen mydate = mdy(month,day,year)
```

Or, with quarterly data, we would use the `yq()` function:

```
gen qtr=yq(year,quarter)
```

All of the functions are:

`mdy(month,day,year)`	for daily data
`yw(year, week)`	for weekly data
`ym(year,month)`	for monthly data
`yq(year,quarter)`	for quarterly data
`yh(year,half-year)`	for half-yearly data

3. Converting a date variable stored as a single number

   As discussed above, if you have a single numeric variable, we need to first convert it into its component parts in order to use the mdy function. For example, imagine the variable is of the form yyyymmdd (for example, 19990320 for March 20 1999); now we need to split it into year, month and day as follows:

```
gen year = int(date/10000)
gen month = int((date-year*10000)/100)
gen day = int((date-year*10000-month*100))
gen mydate = mdy(month,day,year)
```

In each case the `int(x)` command returns the integer obtained by truncating x towards zero.

Using the time series date variables

Once we have the date variable in Stata elapsed time form, it is not the most intuitive to work with. For example, here is how a new variable called stata_date will look by using the command

```
gen stata_date = mdy(month,day,year)
```

month	**day**	**year**	**stata_date**
7	11	1948	-4191
1	21	1952	-2902
8	12	1993	12277
11	2	1994	12724

Therefore to display the stata_date in a more user-friendly manner, we can use the format command as follows:

```
format  stata_date %d
```

This means that stata_date will now be displayed as:

month	day	year	stata_date
7	11	1948	11jul1948
1	21	1952	21jan1952
8	12	1993	12aug1993
11	2	1994	02nov1994

It is possible to use alternatives to %d, or to use %td to display elapsed dates in numerous other ways – in fact, we can control everything about the display. For example if I had instead written:

```
format stata_date %dM_d,_CY
```

Then we would get:

month	day	year	stata_date
7	11	1948	July 11, 1948
1	21	1952	January 21, 1952
8	12	1993	August 12, 1993
11	2	1994	November 2, 1994

Making use of Dates

If we want to use our dates in an *if* command, we have a number of options:

1. Exact dates

   We have a selection of functions d(), w(), m(), q(), h(), and y() to specify exact daily, weekly, monthly, quarterly, half-yearly, and yearly dates respectively. For example:

   ```
   reg x y if w(1995w9)
   sum income if q(1988-3)
   tab gender if y(1999)
   ```

2. A date range
   If you want to specify a range of dates, you can use the tin() and twithin() functions:

   ```
   reg y x if tin(01feb1990,01jun1990)
   sum income if twithin(1988-3,1998-3)
   ```

   The difference between tin() and twithin() is that tin() includes the beginning and end dates, whereas twithin() excludes them. Always enter the beginning date first, and write them out as you would for any of the d(), w(), etc. functions.

Time-series tricks using Dates

Often in time-series analyses we need to "lag" or "lead" the values of a variable from one observation to the next. Or we need to take first, second or seasonal differences. One way to do this is to generate a whole bunch of variables which represent the lag or the lead, the difference, etc… But if we have many variables, this can be take up a lot of memory.

You should use the `tsset` command before any of the "tricks" in this section will work. This has the added advantage that if you have defined your data as a panel, Stata will automatically re-start any calculations when it comes to the beginning of a new cross-sectional unit so you need not worry about values from one panel being carried over to the next.

- Lags and Leads

  These use the L.varname (to lag) and F.varname (to lead) commands. Both work the same way:

```
reg income L.income
```

This regresses income(t) on income(t-1)

If you wanted to lag income by more than one time period, you would simply change the L. to something like "L2." or "L3." to lag it by 2 and 3 time periods respectively.

- Differencing

Used in a similar way, the *D.varname* command will take the first difference, *D2.varname* will take the double difference (difference in difference), etc… For example:

Date	**income**	**D.income**	**D2.income**
02feb1999	120	.	.
02mar1999	230	110	.
02apr1999	245	15	5

- Seasonal Differencing

The *S.varname* command is similar to the *D.varname*, except that the difference is always taken from the current observation to the n-th observation: In other words: S.income=income(t)-income(t-1) and S2.income=income(t)-income(t-2)

Date	**income**	**S.income**	**S2.income**
02feb1999	120	.	.
02mar1999	230	110	.
02apr1999	245	15	125

Survey data

Survey data is very common in many applied microeconomic settings. Labour Economics, Economics of Education, Industrial Organization, all these fields employ surveys regularly. To minimize costs surveys are often designed such that the whole population is split into several strata and within these strata sampling (with differing probabilities) takes place. As an example we can consider so called linked employer-employee (LEE) data. This type of data is very useful for several fields of economics. You can e.g. in labour economics control for firm effects, productivity indicators or capital, or in industrial economics control for workforce characteristics. A good overview of the issues involved can be found at http://www2.napier.ac.uk/depts/fhls/peas/theory.asp.

The sampling for LEE data is usually conducted in two stages. First a sample of firms is selected and in the second stage employees within these firms are randomly sampled. If this method would be conducted without stratification, it would be very unlikely that very large firms (of which there are only a few) would be part of our sample. So in order to be able to make statements about all firms in a country we would need a very large sample so that the probability that we observe a few very large firms is not zero. An easier method would be to split the population of all firms into small, medium and large firms and draw independent samples from each of these groups (or strata as they are called in Stata).

Conversely we would want to sample a share of employees that is inversely related to the size of the firm. If we would for example set the share of surveyed employees in a firm to ten percent, we would only get responses from 1 or 2 employees in very small firms, a number that will not let us infer anything on the firm's total workforce. For large firms ten percent might even be too large a number (think General Electric). So we will have to use different sampling probabilities (the inverse of which are the sampling weights – pweight). Note that we could also apply different sampling probabilities within each stratum, if we have a lot more small than medium and large firms we might want to sample less from that category.

Besides weights and strata Stata can also account for clustering. Clusters arise by sampling groups of individuals (e.g. employees from a firm, children from a school or from a class, households from a district). In our example we sample individuals from firms and set this as our clustering variable.

The dataset would be provided with a firm indicator, the inverse sampling probability of the firm (the sampling weight), information on the strata (which we can use to construct a stratum indicator dummy) and sampling weights for the workers. If there is no overall sampling weight we can generate it by multiplying firm and worker sampling weights (careful when working with weights, in Stata the sampling weight is the inverse sampling probability).

In order to get point estimates right you need to use sampling weights, clustering and stratification (as well as sampling weights) matter for the correct estimation of standard errors.

```
. svyset firm_id [pweight=total_w], strata(firm_size_cat) || _n
```

In small samples we might want to add a finite population correction. This will affect the standard errors, but only in small populations (see the PEAS website). For the first stage this is no problem in our artificial example. We generate a variable that tells Stata the share of the population that the cluster is representing (i.e. the sampling probability) and we are done.

```
. gen double firm_sp = 1/firm_w
. svyset firm_id [pweight=total_w], strata(firm_size_cat) fpc(firm_sp) || _n
```

For the second stage things get a little trickier, the variable that indicates the finite population may not vary within a stratum. If this is the case and the probability of an employee to be part of the sample is the same for each employee in the same stratum it suffices to specify the `fpc()` option using the workers sampling probability. If not, it might be that the second stage is stratified as well, for example we might have first stage strata defined by industry and region of a firm and after selecting the firms we stratify by firm size for the second stage.

```
. svyset firm_id [pweight=total_w], strata(firm_ind_reg_cat) fpc(firm_sp) || _n,
strata(firm_size_cat) fpc(worker_sp)
```

The number of commands that support the survey option is constantly increasing (the number more than doubled from Stata 9 to Stata 10). All the basic estimation commands (ordinary least squares, instrumental variable estimation, logit and probit models and many more) are supported by `svy`. To account for the survey properties we simply use the `svy` prefix:

```
. svy: regress y x
```

Programming

Program Basics

Creating or "defining" a program

A program contains a set of commands and is activated by a single command. A do-file is essentially one big program – it contains a list of commands and is activated by typing:

```
. do "class 4.do"
```

You can also create special programs within a do-file, especially useful when you have a set of commands that are going to be used repetitively. The use of these programs will initially be demonstrated interactively, but they are best used within a do-file.

We will keep on using the PWT dataset from last week's classes. Suppose you want to create new variables that contain the average values (across countries and years) of some of the underlying variables in the dataset and at the same time display on screen these averages. No single Stata command will do this for you, but there are a couple of ways you can combine separate Stata commands to reach your goal. The most efficient method is:

```
. egen mean_kc=mean(kc)
. tab mean_kc

    mean_kc |      Freq.     Percent        Cum.
------------+-----------------------------------
   72.53644 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00

. egen mean_kg=mean(kg)
. tab mean_kg

    mean_kg |      Freq.     Percent        Cum.
------------+-----------------------------------
   20.60631 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00.
```

The tasks are the same for each variable you are interested in. To avoid repetitive typing or repetitive cutting and pasting, you can create your own program that combines both tasks into a single command (note, in what follows the first inverted comma or single-quote of `` `1' `` is on the top-left key of your keyboard, the second inverted comma is on the right-hand side on the key with the @ symbol, and inside the inverted commas is the number one, not the letter L):

```
program define mean
egen mean_`1'=mean(`1')
tab mean_`1'
end
```

You have now created your own Stata command called `mean`, and the variable you type after this new command will be used in the program everywhere there is a `` `1' ``. For example, `mean kg` will use kg everywhere there is a `` `1' ``. This command can now be applied to any variable you wish:

```
. mean ki

    mean_ki |      Freq.     Percent        Cum.
------------+-----------------------------------
   15.74088 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00
```

Naming a program

You can give your program any name you want as long as it isn't the name of a command already in Stata, e.g. you cannot name it `summarize`. Actually, you can create a program called `summarize`, but Stata will simply ignore it and use its own

summarize program every time you try using it. To check whether Stata has already reserved a particular name:

```
. which sum
built-in command: summarize

. which mean
command mean not found as either built-in or ado-file
r(111);
```

Redefining a program

You may want to change your program in some way, such as altering a line, or adding or dropping a line. For example, the tabulate command displays more than just the single number we are interested in. We can provide a more user-friendly result using the display command, where everything in double-quotes (“”) is interpreted as straightforward text and anything not in double-quotes is interpreted as something in Stata memory, such as a variable name or results of a previous command (e.g. e(_b) or e(_se) from a regress command):

```
. display "Mean of kg = " mean_kg
Mean of kg = 21.15341
```

Note, the value of mean_kg that is displayed is that of the first observation (_n=1). In our example, the value just so happens to be the same for all observations so we don’t care whether it is displaying the first, tenth or one-hundredth observation. However, this will not be so in many other examples, so care needs to be taken when using this command in this way.

We can redefine our mean program by replacing the tabulate command with this display command. To do so, we must first drop the old mean program (placing capture before the command avoids Stata tripping up if there is no program called mean defined in the first place – useful for preventing Stata crashing in the middle of a long do-file):

```
capture program drop mean
program define mean
egen mean_`1’=mean(`1’)
display “Mean of `1’ = ” mean_`1’
end
```

Now we need to drop the existing mean_kc and mean_kg variables and re-run the commands to get:

```
. mean kc
Mean of kc = 72.536438

. mean kg
Mean of kg = 20.606308
```

Debugging a program

Your program may crash out half-way through for some reason:

```
. mean kc
mean_kc already defined
r(110);
```

Here, Stata tells us the reason why the program has crashed – you are trying to create a new variable called mean_kc but there is an old variable already called that. Our mean program is a very simple one, so we can figure out very quickly that the problem arises with the first line, egen mean_`1’=mean(`1’). However, with more intricate programs, it is not always so obvious where the problem lies. This is where the set trace command comes in handy. This command traces the execution of the program line-by-line so you can see exactly where it trips up. Because the trace details are often very long, it is usually a good idea to log them for review afterwards.

```
. log using “debug.log”,replace
. set more off
. set trace on
. mean kg
. set trace off
. log close
```

You can further limit the amount of output by using `set tracedepth` to follow nested commands only to a certain depth. So if your program calls another program (which in turn calls another program) you can avoid the output from the second level onwards by using

```
. set tracedepth 1
```

Program arguments

Our `mean` command was defined to handle only one argument – `` `1' ``. It is possible to define more complicated programs to handle several arguments, `` `1' ``, `` `2' ``, `` `3' ``, and so on. These arguments can refer to anything you want – variable names, specific values, strings of text, command names, if statements, and so on. For example, we can define a program that displays the value of a particular variable (argument `` `1' ``) for a particular country (argument `` `2' ``) and year (argument `` `3' ``):

```
capture program drop show
program define show
       tempvar obs
       quietly gen `obs'=`1' if (countryisocode=="`2'" & year==`3')
       so `obs'
       display "`1' of country `2' in `3' is: " `obs'
       end
```

To see this in action:

```
. show pop USA 1980
pop of country USA in 1980 is: 227726

. show pop FRA 1990
pop of country FRA in 1990 is: 58026.102
```

Some things to note about this program:

- Line 1 creates a temporary variable that will exist while the program is running but will be automatically dropped once the program has finished running. Once this `tempvar` has been defined, it must be referred to within the special quotes (`` `' ``), just as with the arguments.
- Line 2 starts with `quietly`, which tells Stata to suppress any onscreen messages resulting from the operation of the command on this line.
- Make sure to properly enclose any string arguments within double-quotes. `` `2' `` will contain a string of text, such as ARG or FRA, so when `` `2' `` is being used in a command it should be placed within double-quotes (`""`).
- Line 3 ensures that observation number one will contain the value we are interested in. Missings are interpreted by Stata as arbitrarily large, so when the data is sorted in ascending order our value will be at the top of the list, ahead of these missings.

As we have seen, use of strings can cause a bit of a headache. A further complication may arise if the argument itself is a string containing blank spaces, such as United States instead of USA. Stata uses blank spaces to number the different arguments, so if we tried `show kg United States 1980`, Stata would assign kg to `` `1' ``, United to `` `2' ``, States to `` `3' `` and 1980 to `` `4' ``. The way to get around this is to enclose any text containing important blank spaces within double-quotes – the proper command then would be:

```
. show kg "United States" 1980
```

Renaming arguments

Using `` `1' ``, `` `2' ``, `` `3' ``, and so on can be confusing and prone to error. It is possible to assign more meaningful names to the arguments at the very beginning of your program so that the rest of the program is easier to create. Make sure to continue to include your new arguments within the special quotes (`` `' ``):

```
. capture program drop show
. program define show
1. args var cty yr
2. tempvar obs
3. quietly gen `obs'=`var' if countryisocode=="`cty'" & year==`yr'
4. so `obs'
5. display "`var' of country `cty' in `yr' is: " `obs'
6. end
```

```
show kg USA 1980

kg of country USA in 1980 is: 13.660507
```

Macros

A Stata macro is different to an Excel macro. In Excel, a macro is like a recording of repeated actions which is then stored as a mini-program that can be easily run – this is what a do file is in Stata. Macros in Stata are the equivalent of variables in other programming languages. A macro is used as shorthand – you type a short macro name but are actually referring to some longer name or string of characters. For example, you may use the same list of independent variables in several regressions and want to avoid retyping the list several times. Just assign this list to a macro. Using the PWT dataset:

```
. local varlist gdp60 openk kc kg ki
. regress grgdpch `varlist' if year==1990

      Source |       SS       df       MS              Number of obs =     111
-------------+------------------------------           F(  5,   105) =    6.70
       Model |  694.520607     5  138.904121           Prob > F      =  0.0000
    Residual |  2175.67123   105  20.7206784           R-squared     =  0.2420
-------------+------------------------------           Adj R-squared =  0.2059
       Total |  2870.19184   110  26.0926531           Root MSE      =   4.552

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -1.853244   .6078333    -3.05   0.003    -3.058465   -.6480229
       openk |  -.0033326   .0104782    -0.32   0.751    -.0241088    .0174437
          kc |  -.0823043   .0356628    -2.31   0.023     -.153017   -.0115916
          kg |  -.0712923   .0462435    -1.54   0.126    -.1629847    .0204001
          ki |   .2327257   .0651346     3.57   0.001     .1035758    .3618757
       _cons |   16.31192   5.851553     2.79   0.006     4.709367    27.91447
------------------------------------------------------------------------------

. regress  grgdpch `varlist' if year==1980

      Source |       SS       df       MS              Number of obs =     111
-------------+------------------------------           F(  5,   105) =    2.27
       Model |  880.685302     5   176.13706           Prob > F      =  0.0524
    Residual |  8130.51957   105  77.4335197           R-squared     =  0.0977
-------------+------------------------------           Adj R-squared =  0.0548
       Total |  9011.20487   110  81.9200443           Root MSE      =  8.7996

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.2969159   1.143709    -0.26   0.796    -2.564679    1.970847
       openk |   .0023893   .0218479     0.11   0.913    -.0409311    .0457097
          kc |  -.1349823   .0518524    -2.60   0.011     -.237796   -.0321686
          kg |  -.1363929   .0845697    -1.61   0.110     -.304079    .0312932
          ki |  -.1307708   .1124651    -1.16   0.248    -.3537683    .0922267
       _cons |   16.98343   9.885368     1.72   0.089    -2.617433    36.58429
------------------------------------------------------------------------------
```

Macros are of two types – local and global. Local macros are "private" – they will only work within the program or do-file in which they are created. Thus, for example, if you are using several programs within a single do-file, using local macros for each means that you need not worry about whether some other program has been using local macros with the same names – one program can use `varlist` to refer to one set of variables, while another program uses its `varlist` to refer to a completely different set of variables. Global macros are "public" – they will work in all programs and do files – `varlist` refers to exactly the same list of variables irrespective of the program that uses it. Each type of macro has its uses, although local macros are the most commonly used type.

Just to illustrate this, let's work with an example. The program `reg1` will create a local macro called `varlist` and will also use that macro. The program `reg2` will not create any macro, but will try to use a macro called `varlist`. Although `reg1` has a macro by that name, it is local or private to it, so `reg2` cannot use it:

```
. program define reg1
  1. local varlist gdp60 openk kc kg ki
  2. reg  grgdpch `varlist' if year==1990
  3. end

. reg1

      Source |       SS       df       MS              Number of obs =     111
-------------+------------------------------           F(  5,   105) =    6.70
       Model |  694.520607     5  138.904121           Prob > F      =  0.0000
    Residual |  2175.67123   105  20.7206784           R-squared     =  0.2420
-------------+------------------------------           Adj R-squared =  0.2059
       Total |  2870.19184   110  26.0926531           Root MSE      =   4.552

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -1.853244   .6078333    -3.05   0.003    -3.058465   -.6480229
       openk |  -.0033326   .0104782    -0.32   0.751    -.0241088    .0174437
          kc |  -.0823043   .0356628    -2.31   0.023     -.153017   -.0115916
          kg |  -.0712923   .0462435    -1.54   0.126    -.1629847    .0204001
          ki |   .2327257   .0651346     3.57   0.001     .1035758    .3618757
       _cons |   16.31192   5.851553     2.79   0.006     4.709367    27.91447
------------------------------------------------------------------------------

. capture program drop reg2
. program define reg2
  1. reg  grgdpch `varlist' if year==1990
  2. end

. reg2
      Source |       SS       df       MS              Number of obs =     129
-------------+------------------------------           F(  0,   128) =    0.00
       Model |           0     0           .           Prob > F      =       .
    Residual |  4008.61956   128  31.3173404           R-squared     =  0.0000
-------------+------------------------------           Adj R-squared =  0.0000
       Total |  4008.61956   128  31.3173404           Root MSE      =  5.5962

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _cons |   .9033816    .492717     1.83   0.069    -.0715433    1.878306
------------------------------------------------------------------------------
```

Now, suppose we create a global macro called `varlist` – it will be accessible to all programs. Note, local macros are enclosed in the special quotes (`` `' ``), global macros are prefixed by the dollar sign (`$`).

```
. global varlist gdp60 openk kc kg ki

. capture program drop reg1
. program define reg1
  1. local varlist gdp60 openk kc kg ki
  2. reg grgdpch `varlist'
  3. reg grgdpch $varlist
  4. end

. capture program drop reg2
. program define reg2
  1. reg grgdpch $varlist
  2. reg grgdpch `varlist'
  3. end

. reg1
      Source |       SS       df       MS              Number of obs =    5067
-------------+------------------------------           F(  5,  5061) =   41.21
       Model |  8692.09731     5  1738.41946           Prob > F      =  0.0000
```

```
    Residual |  213498.605  5061  42.1850632           R-squared     =  0.0391
-------------+------------------------------           Adj R-squared =  0.0382
       Total |  222190.702  5066   43.859199           Root MSE      =   6.495

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5393328   .1268537    -4.25   0.000    -.7880209   -.2906447
       openk |  -.0003768   .0020639    -0.18   0.855    -.0044229    .0036693
          kc |  -.0249966   .0055462    -4.51   0.000    -.0358694   -.0141237
          kg |  -.0454862   .0089808    -5.06   0.000    -.0630924     -.02788
          ki |   .1182029    .011505    10.27   0.000     .0956481    .1407578
       _cons |   6.344897   1.045222     6.07   0.000     4.295809    8.393985
------------------------------------------------------------------------------

      Source |       SS       df       MS              Number of obs =    5067
-------------+------------------------------           F(  5,  5061) =   41.21
       Model |  8692.09731     5  1738.41946           Prob > F      =  0.0000
    Residual |  213498.605  5061  42.1850632           R-squared     =  0.0391
-------------+------------------------------           Adj R-squared =  0.0382
       Total |  222190.702  5066   43.859199           Root MSE      =   6.495

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5393328   .1268537    -4.25   0.000    -.7880209   -.2906447
       openk |  -.0003768   .0020639    -0.18   0.855    -.0044229    .0036693
          kc |  -.0249966   .0055462    -4.51   0.000    -.0358694   -.0141237
          kg |  -.0454862   .0089808    -5.06   0.000    -.0630924     -.02788
          ki |   .1182029    .011505    10.27   0.000     .0956481    .1407578
       _cons |   6.344897   1.045222     6.07   0.000     4.295809    8.393985
------------------------------------------------------------------------------

. reg2

      Source |       SS       df       MS              Number of obs =    5067
-------------+------------------------------           F(  5,  5061) =   41.21
       Model |  8692.09731     5  1738.41946           Prob > F      =  0.0000
    Residual |  213498.605  5061  42.1850632           R-squared     =  0.0391
-------------+------------------------------           Adj R-squared =  0.0382
       Total |  222190.702  5066   43.859199           Root MSE      =   6.495

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5393328   .1268537    -4.25   0.000    -.7880209   -.2906447
       openk |  -.0003768   .0020639    -0.18   0.855    -.0044229    .0036693
          kc |  -.0249966   .0055462    -4.51   0.000    -.0358694   -.0141237
          kg |  -.0454862   .0089808    -5.06   0.000    -.0630924     -.02788
          ki |   .1182029    .011505    10.27   0.000     .0956481    .1407578
       _cons |   6.344897   1.045222     6.07   0.000     4.295809    8.393985
------------------------------------------------------------------------------

      Source |       SS       df       MS              Number of obs =    5621
-------------+------------------------------           F(  0,  5620) =    0.00
       Model |           0     0           .           Prob > F      =       .
    Residual |  250604.237  5620  44.5915012           R-squared     =  0.0000
-------------+------------------------------           Adj R-squared =  0.0000
       Total |  250604.237  5620  44.5915012           Root MSE      =  6.6777

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _cons |   2.069907   .0890675    23.24   0.000       1.8953    2.244513
------------------------------------------------------------------------------
```

As you will see, Stata runs two fully specified regressions in the first case but only one in the last case since again, the program reg2 does not recognize `` `varlist' ``.

Macro contents

We introduced macros by showing how they can be used as shorthand for a list of variables. In fact, macros can contain practically anything you want – variable names, specific values, strings of text, command names, if statements, and so on. Note, we were actually using macros implicitly earlier in the class. When we created the programs `mean` and `show`, the arguments (e.g. pop ARG 1980) were passed to the programs via local macros (`` `1' ``, `` `2' ``, `` `3' ``). These local macros contained variables (kg) and specific values (ARG and 1980). Some other examples of what macros can contain:

Text

Text is usually contained in double quotes (“”) though this is not necessary for macro definitions:

```
. local ctyname “United States”
```

gives the same result as

```
. local ctyname United States
```

A problem arises whenever your macro name follows a backslash (\). Whenever this happens, Stata ignores the first single quote (`) of the macro name and so fails to properly load the macro:

```
. local filename PWT.dta
. use “H:\ECStata\`filename’”

file H:\ECStata`filename'.dta not found
r(601);
```

To get around this problem, use double backslashes (\\) instead of a single one or slashes (/) instead of backslashes:

```
. use “H:\ECStata\\`filename’”

. use “H:/ECStata/`filename’”
```

Statements

Using macros to contain statements is essentially an extension of using macros to contain text. For example, if we define the local macro:

```
. local year90 “if year==1990”
```

then,

```
. reg grgdpch $varlist `year90’
```

is the same as:

```
. reg grgdpch gdp60 openk kc kg ki if year==1990
```

Note that when using `if` statements, double quotes become important again. For simplicity, consider running a regression for all countries whose codes start with “B”. First, I define a local macro and then use it in the `reg` command:

```
. local ctyname B
. reg grgdpch gdp60 openk kc kg ki if substr(country,1,1)=="`ctyname'"
```

Although it does not matter whether I define `ctyname` using double quotes or not, it is important to include them in the if-statement since the variable `country` is string. The best way to think about this is to do what Stata does: replace `` `ctyname’ `` by its content. Thus, `substr(country,1,1)=="`ctyname'"` becomes `substr(country,1,1)=="B".` Omitting the double quotes would yield `substr(country,1,1)==B` which as usual results in an error message (since the results of the `substr`-operation is a string).

Numbers and expressions

```
. local i=1
. local result=2+2
```

Note, when the macro contains explicitly defined numbers or equations, an equality sign must be used. Furthermore, there must be no double-quotes, otherwise Stata will interpret the macro contents as text:

```
. local problem=“2+2”
```

Thus, the `problem` macro contains the text `2+2` and the `result` macro contains the number `4`. Note that as before we could also have assigned “2+2” to `problem` while omitting the equality sign. The difference between the two assignments is that assignments using “=” are evaluations, those without “=” are copy operations. That is, in the latter case, Stata simply copies “2+2”

into the macro `problem` while in the former case it evaluates the expression behind the "=" and then assigns it to the corresponding macro. In the case of strings these two ways turn out to be equivalent. There is one subtle difference though: evaluations are limited to string lengths of 244 characters (80 in Intercooled Stata) while copy operations are de facto only limited by available memory. Thus, it is usually safer to omit the equality sign to avoid parts of the macro being secretly cut off (which can lead to very high levels of confusion…)

While a macro can contain numbers, it is essentially holding a string of text that can be converted back and forth into numbers whenever calculations are necessary. For this reason, macros containing numbers are only accurate up to 13 digits. When precise accuracy is crucial, scalars should be used instead:

```
. scalar root2=sqrt(2)
. display root2
1.4142136
```

Note, when you call upon a macro, it must be contained in special quotes (e.g. `display `result'`), but this is not so when you call upon a scalar (e.g. `display root2` and not `display `root2'`).

Manipulation of macros

Contents of macros can be changed by simply redefining a macro. For example, if the global macro `result` contains the value "2+2" typing:

```
. global result "2+3"
```

overwrites its contents. If you want to drop a specific macro, use the `macro drop` command:

```
. macro drop year90
```

To drop all macros in memory, use `_all` instead of specific macro names. If you want to list all macros Stata has saved in memory instead (including a number of pre-defined macros), type:

```
. macro list
```

or

```
. macro dir
```

Macro names starting with an underscore ("_") are local macros, the others are global macros. Similarly, to drop or list scalars, use the commands `scalar drop` and `scalar list` (or `scalar dir`) respectively.

Temporary objects

Besides in macros and variables, Stata can also store information in so-called temporary variables which are often used in longer programmes:

- `tempvar` assigns names to the specified local macro names that may be used as temporary variable names in a dataset (we have already seen this type earlier on). When the program or do-file concludes, any variables with these assigned names are dropped:

```
. program define temporary
  1. tempvar logpop
  2. gen `logpop'=log(pop)
  3. sum pop if `logpop'>=8
  4. end

.
. temporary
(2721 missing values generated)

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         pop |      4162    43433.71    126246.4       2989    1258821
```

Since the tempvar logcgdp is dropped at the end of the program, trying to access it later on yields an error message:

```
. sum pop if `logpop'>=8
>8 invalid name
r(198);
```

- `tempname` assigns names to the specified local macro names that may be used as temporary scalar or matrix names. When the program or do-file concludes, any scalars or matrices with these assigned names are dropped. This command is used more rarely then tempvar but can be useful if you want to do matrix-algebra in Stata subroutines.
- `tempfile` assigns names to the specified local macro names that may be used as names for temporary files. When the program or do-file concludes, any datasets created with these assigned names are erased. For example, try the following programme:

```
. program define temporary2
  1.   tempfile cgdp
  2.   keep country year cgdp
  3.   save "`cgdp'"
  4.   clear
  5.   use "`cgdp'"
  6.   sum year
  7. end

. temporary2
file C:\DOCUME~1\Michael\LOCALS~1\Temp\ST_0c000012.tmp saved

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        year |      8568        1975    14.72046       1950       2000
.
```

  This saves the variables `country year cgdp` in a temporary file that is automatically erased as soon as the programme terminates (check this by trying to reload "`cgdp'" after termination of the programme "temporary").

Looping

There are a number of techniques for looping or repeating commands within your do-file, thus saving you laborious retyping or cutting and pasting. These techniques are not always mutually exclusive – you can often use one or more different techniques to achieve the same goal. However, it is usually the case that one technique is more suitable or more efficient in a given instance than the others. Therefore, it is best to learn about each one and then choose whichever is most suitable when you come across a looping situation.

for

While the `for` command is outdated in Stata 10 it is still useful in some settings. for-processing allows you to easily repeat Stata commands. As an example, we can use the PWT dataset and create the mean of several variables all at once:

```
. for varlist kc ki kg: egen mean_X=mean(X)
->  egen mean_kc=mean(kc)
->  egen mean_ki=mean(ki)
->  egen mean_kg=mean(kg)
```

The `egen` command is repeated for every variable in the specified `varlist`, with the X standing in for the relevant variable each time (note, instead of typing out a long `varlist`, you could e.g. `use varlist kc-ki` to signify every variable listed between `kc` and `kg`, inclusive). You can see in the variables window that our three desired variables have been created.

```
for varlist kc ki kg: display "Mean of X = " mean_X

->  display `"Mean of kc = "' mean_kc
Mean of kc = 72.536438

->  display `"Mean of ki = "' mean_ki
Mean of ki = 15.740885

->  display `"Mean of kg = "' mean_kg
Mean of kg = 20.606308
```

The onscreen display includes both the individual commands and their results. To suppress the display of the individual commands, use the `noheader` option:

```
for varlist kc ki kg, noheader: display "Mean of X = " mean_X
```

```
Mean of kc = 72.536438
Mean of ki = 15.740885
Mean of kg = 20.606308
```

To suppress both the individual commands and their results, you need to specify `quietly` before `for`. The example we have used above repeats commands for a list of existing variables (`varlist`). You can also repeat for a list of new variables you want to create (`newlist`):

```
. for newlist ARG FRA USA : gen Xpop=pop if countryisocode=="X" & year==1995
```

It is also possible to repeat for a list of numbers (`numlist`) or any text you like (`anylist`). For example, suppose we wanted to append several similarly named data files to our existing dataset:

```
. for numlist 1995/1998: append using "H:\ECStata\dataX.dta"
```

Note, the full file name `H:\ECStata\dataX.dta` must be enclosed in double quotes, otherwise Stata will get confused and think the backslash `\` is a separator belonging to the `for` command:

```
. for numlist 1995/1998: append using H:\ECStata\dataX.dta
-> append using F:
file F: not found
r(601);
```

It is possible to nest several loops within each other. In this case, you need to specify the name of the macro Stata uses for the list specified after `for` (in above examples, Stata automatically used "X"):

```
. for X in varlist kg cgdp: for Y in numlist 1990/1995: sum X if year==Y
```

It is also possible to combine two or more commands into a single for-process by separating each command with a backslash `\`:

```
. for varlist kg cgdp, noheader: egen mean_X=mean(X) \ display "Mean of X = " mean_X
```

If the list of commands you want to repeat is very long and/or complicated, it may be worthwhile using `for` in conjunction with a custom-made program containing your list of commands:

```
capture program drop mean
program define mean
quietly egen mean_`1'=mean(`1')
display mean_`1'
end

for varlist kg cgdp: mean X
```

foreach and forvalues

The `for` loop is officially replaced from version 0 onwards by the `foreach` and `forvalues` loops. The former is used in conjunction with strings, the latter with numeric values. We know that it is possible to combine several commands into a single for-process. This can get quite complicated if the list of commands is quite long, but we saw how you can overcome this by combining `for` with a custom-made program containing your list of commands. The `foreach` command does the same thing without the need for creating a separate program:

```
foreach var in kg cgdp {
       egen mean_`var'=mean(`var')
       display "Mean of `var' = " mean_`var'
}

Mean of kg = 20.606308
Mean of cgdp = 7.4677978
```

With the `foreach...in` command, `foreach` is followed by a macro name that you assign (e.g. `var`) and `in` is followed by the list of arguments that you want to loop (e.g. `kg cgdp`). This command can be easily used with variable names, numbers, or any string of text – just as `for`.

While this command is quite versatile, it still needs to be redefined each time you want to execute the same list of commands for a different set of arguments. For example, the program above will display the mean of `kg` and `cgdp`, but suppose that later on in your do-file you want to display the means of some other variables – you will have to create a new `foreach` loop. One way to get around this is to write the `foreach` loop into a custom-made program that you can then call on at different points in your do-file:

```
capture program drop mean
program define mean
foreach var of local 1 {
       egen mean_`var'=mean(`var')
       display "Mean of `var' = " mean_`var'
}
end

. mean "kg cgdp"
Mean of kg = 20.606308
Mean of cgdp = 7.4677978

. mean "ki pop"
Mean of ki = 15.740885
Mean of pop = 31252.467
```

This method works, but can be quite confusing. Firstly, `of local` is used in place of `in`. Secondly, reference to the `local macro `1'` in the first line does not actually use the single quotes we are used to. And thirdly, the list of arguments after the executing command must be in double quotes (so that everything is passed to the macro `` `1' `` in a single go). For these reasons, it can be a good idea to use `foreach` only when looping a once-off list. A technique called macro shift can be used when you want to loop a number of different lists (see later).

The `forvalues` loop works similarly but is defined over a range of values. Instead of `in` an equality sign is used and the values are given as a range, increasing by 1 unit or by a certain increment.

```
forvalues t = 1980/1983 {
       show pop USA `t'
}
pop of country USA in 1980 is: 227726
pop of country USA in 1981 is: 230008
pop of country USA in 1982 is: 232218

forvalues t = 1980(5)1990 {
       show pop USA `t'
}
pop of country USA in 1980 is: 227726
pop of country USA in 1985 is: 238506
pop of country USA in 1990 is: 249981
```

Incremental shift (number of loops is fixed)

You can loop or repeat a list of commands within your do-file using the `while` command – as long as the `while` condition is true, the loop will keep on looping. There are two broad instances of its use – the list of commands are to be repeated a fixed number of times (e.g. 5 loops, one for each year 1980-84) or the number of repetitions may vary (e.g. maybe 5 loops for a list of 5 years, or maybe 10 loops for a list of 10 years). We will look first at the incremental shift technique for a fixed number of loops. We can see how it works using the following very simple example:

```
local i=1
while `i'<=5 {
       display "loop number " `i'
       local i=`i'+1
}
loop number 1
loop number 2
loop number 3
loop number 4
loop number 5
```

The first command defines a local macro that is going to be the loop increment – it can be seen as a counter and is set to start at 1. It doesn't have to start at 1, e.g. if you are looping over years, it may start at 1980.

The second command is the `while` condition that must be satisfied if the loop is to be executed. This effectively sets the upper limit of the loop counter. At the end of the `while` command is an open bracket `{` that signifies the start of the looped or repeated set of commands. Everything between the two brackets { } will be executed each time you go through the `while` loop.
The final command before the close bracket `}` increases or increments the counter, readying it to go through the loop again (as long as the `while` condition is still satisfied). In actuality, it is redefining the local macro `` `i' `` – which is why there are no single quotes on the left of the equality but there are on the right. The increase in the counter does not have to be unitary, e.g. if you are using bi-annual data you may want to fix your increment to 2. All the looped commands within the brackets are defined in terms of the local macro `` `i' ``, so in the first loop everywhere there is an `` `i' `` there will now be a 1, in the second loop a 2, and so on. If the increase is unitary we can use the shorthand notation:

```
local ++i     /* same result as local i=`i'+1 */
```

To see a more concrete example, we will create a program to display the largest per capita GDP each year for every year 1980-84:

```
capture program drop maxcgdp
program define maxcgdp
       local i=1980
       while `i'<=1984 {
              tempvar mcgdp
              quietly egen `mcgdp'=max(cgdp) if year==`i'
              so `mcgdp'
              display `i' " " `mcgdp'
              local ++i
       }
end

maxcgdp
1980 9.4067564
1981 9.5102491
1982 9.5399132
1983 9.6121063
1984 9.7101154
```

Macro shift (number of loops is variable)

The incremental shift technique used a fully defined counter with a fixed start (1980), end (1984) and increment (1 year). You type a single command (`maxrgdpl`) to execute the program that loops over this fully defined counter. However, this technique cannot be used if the required replications are not so neatly definable, e.g. you want to repeat a set of commands for 1980, 1984, 1986 and 1995, or you want to repeat the commands for 1980-84 and 1990-94. Instead, you write a program that is executed by the command and a list of arguments that represent the required replications (e.g. `maxrgdpl 1980 1984 1986 1995`). Stata will allocate the first argument to local macro `` `1' ``, the second to local macro `` `2' ``, and so on. Thus, you need to shift through each of these arguments or local macros in order to shift through the required replications. A simple example of how this works:

```
capture program drop displayno
program define displayno
       while "`1'"~="" {
              display `1'
              macro shift
       }
end
displayno 1 2 4 8 10
1
2
4
8
10
. displayno 77 90876 8
77
90876
8
```

The command `macro shift` is used here instead of the counter increment device – it shifts the contents of local macros one place to the left; `` `1′ `` disappears and `` `2′ `` becomes `` `1′ ``, `` `3′ `` becomes `` `2′ ``, and so on. So, in the example above, `` `1′ `` initially contained the number 77, `` `2′ `` contained 90876 and `` `3′ `` contained 8. The looped commands are in terms of `` `1′ `` only so the first replication uses the number 77. The `mac shift` command then shifts `` `2′ `` into the `` `1′ `` slot, so the second replication uses the number 90876. Similarly, the third replication uses the number 8.

The `while` command at the start of the loop ensures that it will keep on looping until the local macro `` `1′ `` is empty, i.e. it will work as long as `"`1′"` is not an empty string `""`. This is similar to the `while` command in the incremental shift technique, but here the loop is defined in terms of `` `1′ `` instead of `` `i′ `` and it is contained in double quotes. The use of double quotes is a convenient way to ensure the loop continues until the argument or macro `` `1′ `` contains nothing – it has nothing to do with whether the arguments are strings of text or numbers.

For a more realistic application of this technique, we can revisit our `maxcgdp` program:

```
capture program drop maxcgdp
program define maxcgdp
       while "`1'"~="" {
              tempvar mcgdp
              quietly egen `mcgdp'=max(cgdp) if year==`1'
              so `mcgdp'
              display `1' " " `mcgdp'
              macro shift
       }
end

. maxcgdp 1983 1991 1995 1999
1983 9.6121063
1991 10.137326
1995 10.426952
1999 10.699246
```

Note that, essentially, the only things that have changed are the format of the `while` command, the format of the shifting mechanism and the way in which the local macro in the loop is defined (`` `1′ `` instead of `` `i′ ``).

The `macro shift` technique is commonly used to shift through variables rather than actual values. For example:

```
capture program drop mean
program define mean
       while "`1'"~="" {
              tempvar mean
              quietly egen `mean'=mean(`1')
              display "Mean of `1' = " `mean'
              macro shift
       }
end
```

Now, we can display the mean of a single variable:

```
. mean kg
Mean of kg = 20.606308
```

or of a list of variables:

```
. mean kg pop cgdp
Mean of kg = 20.606308
Mean of pop = 31252.467
Mean of cgdp = 7.4677978
```

Branching

Branching allows you to do one thing if a certain condition is true, and something else when that condition is false. For example, suppose you are doing some sort of analysis year-by-year but you want to perform different types of analyses for the earlier and later years. For simplicity, suppose you want to display the minimum per capita GDP for the years to 1982 and the maximum value thereafter:

```
capture program drop minmaxcgdp
program define minmaxcgdp
       local i=1980
       while `i'<=1984 {
              if `i'<=1982 {
                     local function min
                     }
              else {
                     local function max
              }
              tempvar mcgdp
              quietly egen `mcgdp'=`function'(cgdp) if year==`i'
              so `mcgdp'
              display `i' " " `mcgdp'
              local ++i
       }
end
. minmaxcgdp
1980 5.518826
1981 5.9500399
1982 6.0682001
1983 9.6121063
1984 9.7101154
```

The structure of this program is almost identical to that of the `maxcgdp` program created earlier. The only difference is that `egen` in line 6 is now a `min` or `max` function depending on the if/else conditions in lines 3 and 4.

It is very important to get the brackets `{}` correct in your programs. Firstly, every `if` statement, every `else` statement, and every `while` statement must have their conditions fully enclosed in their own set of brackets – thus, if there are three condition with three open brackets `{`, there must also be three close brackets `}`. Secondly, nothing except comments in `/* */` should be typed after a close bracket, as Stata automatically moves on to the next line when it encounters a close bracket. Thus, Stata would ignore the `else` condition if you typed:

```
. if `i'<=1982 {local function min} else {local function max}
```

Thirdly, it is necessary to place the brackets and their contents on different lines, irrespective of whether the brackets contain one or more lines of commands. Finally, it is possible to embed if/else statements within other if/else statements for extra levels of complexity, so it is crucial to get each set of brackets right. Suppose you want to display the minimum per capita GDP for 1980 and 1981, the maximum for 1982 and 1983, and the minimum for 1984:

```
capture program drop minmaxrgdpl
program define minmaxrgdpl
       local i=1980
       while `i'<=1984 {
              if `i'<=1981 {
                     local function min
              }
              else {
                     if `i'<=1983 {
                            local function max
                     }
                     else {
                            local function min
                     }
              }
              tempvar mrgdpl
              quietly egen `mrgdpl'=`function'(rgdpl) if year==`i'
              so `mrgdpl'
              display `i' " " `mrgdpl'
              local i=`i'+1
       }
end

. minmaxcgdp
```

```
1980 5.518826
1981 5.9500399
1982 9.5399132
1983 9.6121063
1984 6.1164122
```

One final thing to note is that it is important to distinguish between the conditional `if`:

```
. sum cgdp if cgdp>8
```

and the programming `if`:

```
if cgdp >8 {
       sum cgdp
}
```

The conditional if summarizes all the observations on `cgdp` that are greater than 8. The programming if looks at the first observation on `cgdp` to see if it is greater than 8, and if so, it executes the sum `cgdp` command, i.e. it summarizes *all* observations on `cgdp` (try out the two commands and watch the number of observations).

ADO programming

ADO programming involves setting up user-defined programmes which will be stored in the memory of Stata and then can be retrieved as a command whenever you use Stata. For example, *regress* is used as an ado file. While I think it is pretty advanced to start programming your own complex ado files, some very simple files might be of use. The following simple examples come from http://www.ats.ucla.edu/stat/stata/seminars/stata_programming/default.htm. They are 3 programs of increasing complexity (and therefore flexibility) to take the median of a series or set of series.

Median Program

Version #1

Basic program to deal with one variable.

```
program define median1
  version 6
  sort `1'
  quietly count if `1' ~= .
  local n = r(N)
  local mid = int(`n'/2)
  local odd = mod(`n',2)

  if `odd' {
    local median = `1'[`mid'+1]
  }
  else {
    local median = (`1'[`mid'] + `1'[`mid'+1])/2
  }

  display "Median of `1' = `median'"
end
```

Version #21

Multiple variables and saves results in return list.

```
program define median2, rclass
  display
  display in green " Variable        N    Median"
  display in green "----------------------------"

  while "`1'" ~= "" {
    quietly count if `1' ~= .
    local n = r(N)
    local i = int(`n'/2)
    local odd = mod(`n',2)
    sort `1'
    if `odd' {
      local median = `1'[`i'+1]
    }
    else {
      local median = (`1'[`i'] + `1'[`i'+1])/2
    }
    display in yellow %9s "`1'" %9.0f `n' %10.2f `median'
    macro shift
  }
  return local Mdn = `median'
  return local N = `n'
end
```

Version #3

Allows for multiple series and for “if” and “in” statements.

```
program define median3, rclass
  syntax varlist [if] [in]
  tokenize `varlist'
  preserve
  marksample touse
  display
  display in green " Variable        N    Median"
  display in green "----------------------------"

  while "`1'" ~= "" {
    quietly keep if `touse'
    quietly count
    local n = r(N)
    local i = int(`n'/2)
    local odd = mod(`n',2)
    sort `1'
    if `odd' {
      local median = `1'[`i'+1]
    }
    else {
      local median = (`1'[`i'] + `1'[`i'+1])/2
    }
    display in yellow %9s "`1'" %9.0f `n' %10.2f `median'
    macro shift
  }
  return local Mdn = `median'
  return local N = `n'
end
```

These programmes can be written (separately) in the do file editor and then saved as .ado files. You should save them in the Stata directory which contains ado file updates, under the “m” folder. Then whenever you type:

```
median1 variablename
```

it will calculate the median for you.