Stata: Data Manipulation and Analysis

How to Use This Course Book

This handbook accompanies the taught session for the course. Each section contains a brief overview of a topic for your reference and some sections are followed by exercises.

The Exercises

Exercises are arranged as follows:

- A title and brief overview of the tasks to be carried out;
- A numbered set of tasks, together with a brief description of each;
- Solutions to the exercises will be handed out at the end of the course

Some exercises, particularly those within the same section, assume that you have completed earlier exercises. Your lecturer will direct you to the location of files that are needed for the exercises. If you have any problems with the text or the exercises, please ask the lecturer or one of the demonstrators for help.

This book includes plenty of exercise activities – more than can usually be completed during the hands-on sessions of the course as well as some tasks that can be performed as a homework. These are clearly outlined throughout the coursebook.

Writing Conventions

Certain conventions are used to help you to be clear about what you need to do in each step of a task.

- Stata commands are presented with a small font on a new line similarly to the official Stata syntax conventions.
- A button to be clicked will look like this.

Objectives

From this coursebook you should:

- Be able to set up libraries within Stata
- Be familiar with egen functions
- Understand the role of _N and _n
- Be able to perform a variety of data manipulations using Stata functions
- Know how to use by groups effectively
- Be able to recode string into numeric variables and vice versa
- Be able to merge, append and reshape data

Contents

Software Used

STATA 13

Files Used

bhps_demo.dta
bhps sample file.dta
reshape data.dta
StataDataManipulation.do

Revision Information

Version	Date	Author	Changes made
1.0	July 2008	Adam Whitworth & Kate Wilkinson	Created
1.1	Sept 2009	Neli Demireva	Revised and updated
2.1	March 2013	Neli Demireva	Revised and updated
2.2	July 2013	Ladislav Kozak	Revised and updated
3.3	April 2015	Ines Rombach	Revised and updated

Copyright

Email:
courses@it.ox.ac.uk

1 Setting up a New .do File: Keeping a Log and Specifying Libraries Using Global Macros

The *Stata: Introduction to Data Access and Management* course introduced working in .do files as opposed to working with Stata interactively, and outlined that in order to keep a record of the syntax and to be able to replicate tasks in the future it was advisable to work in .do files. However, the way we used .do files in that course could have been improved by specifying the libraries to be used throughout the syntax at the start of the .do file. This allows us to avoid typing the whole file path each time we wish to open, save or merge a file. To open a new .do file simply open Stata, click the window of the '.do file editor' and then 'New .do file'.

1.1. Clear & set more off

Frequently, there are several things which are (or can be) done at the start of each .do file to set it up. As an example, one of my own syntax files would start as follows:

```
clear

set more off
```

clear just clears anything data that is already open: it creates a blank slate for the current Stata session.

Set more off is a personal preference and lets the results window keep scrolling down automatically when it fills up without being interrupted by the –more- statement.

1.2. Capture log close

`capture log close` shuts down a log file if it is open at this time and this ensures that you will be able to open a log at the start of this .do file without any trouble. It is preferable to put this at the end of each do file if you remember but some people put it at the start of each .do file instead.

```
capture log close
```

1.3. Keeping a log of the .do-file

Next, it is a good idea to keep a log file of the session. First, you have to create a folder in which the log can be kep, i.e. 'Stata logs'. A log file is useful because it keeps a complete record of all of the syntax and output that we get from the analyses:

```
log using ///
```

```
 "H:\StataLevel2\Stata logs\StataDataManipulation.log", replace
```

Note the .log ending (unlike .dta for data files in Stata and .do for .do files). The `, replace` at the end just means that if a log file of this name exists in this folder then write over it – this is useful if you are running the syntax for a project multiple times as you refine the syntax, and you wish to simply the replacement of the log file accordingly each time. The main option for log files is the `append` option and this is used when you wish to tack a log file onto the bottom of an existing log file. Now that the log is open everything which passes through the results window will be saved to the log to give a complete record of the session (both syntax and output) which can be used to check that the syntax has worked as expected.

1.4. Annotations

Next you would probably want to give your .do file a title and date so you know what it is and the date that changes were last made. We can annotate .do files by surrounding text with

/* *text* */ and Stata ignores everything in between – it does not matter how many stars there are but if you include slashes then the direction of the slashes should be forwards as shown here. It is good practice to annotate all .do files throughout the steps so that they make sense to you in the future (when you may well have forgotten what you did and why) and so they make sense to others who may have to read the syntax.

```
/* Stata: Intermediate Data manipulation and analysis  */
/* 20 June 2014*/
```

Alternatively, annotations can be made by simply using * *text*, where * has to be sued at the beginning of each line.

1.5. Global Macros

Finally, during a syntax file (or you can think of this as a single Stata session) we will typically want to pick up and save data from a few specific places. One option is to write out the complete folder path each time we wish to use or save a file throughout a do file, for example:

```
use "H:\StataLevel2\bhps household file.dta",clear
```

but this is likely to become tedious if we are using and saving many files as is typically the case. Instead, **global macro**s can be used at the start of each .do file to set up library folders for the various file paths which will be used in the syntax to either use or save data. It does not matter what you call your global macro but it should only be a single word. Take the example from the above – its global macro will be assigned as:

```
 global raw "H:\StataLevel2\raw data"
```

Once, the global macro is assigned, it is used by typing the dollar sign followed by the name of the global macro: Stata understands this by reading whatever the global macro is set to (in this case the folder paths). Macros (both global and local) are very useful and one common usage of global macros is to set up libraries at the start of a syntax file - this saves us having to write out a file path every time we open, save or merge a file.

It is a good idea to keep your folders tidy so that it is obvious which file is which and what are the most recent versions of everything. One common way of working is to use separate folders for logs & syntax files, raw data files, working files, and final data files.

```
*raw data folder
global raw "H:\StataLevel2\Raw data"

*working files folder
global work "H:\StataLevel2\Stata data\Work"

*final data files
global final "H:\StataLevel2\Stata data\Final"
```

Note that above we could, if we wanted to, specify a global macro for the folder in which we wanted to save the log file, and then we could have called that global macro to save the log file to that folder in the next line of the syntax. However, this is not actually that useful in reality because we will likely only specify it once (i.e. when we open the log on the next line) and so we may prefer just to type the folder path in full for the log as we did above.

Your .do file is now ready to go and should look something like this:

```
clear
set more off
capture log close

log using "H:\StataLevel2\Stata Data manipulation and analysis.log",replace

global raw "H:\StataLevel2\raw data"
global work "H:\StataLevel2\Stata data\Work"
global final "H:\StataLevel2\ Stata data\Final"

use "$raw\bhps household file. dta",clear
```

NB: Note that both work and raw folders should have been previously created.

1.6. The current working file

Similarly to other programmes, Stata has a concept of a “current working directory” in which it stores the files we have been working upon.

It is important to ensure that files are saved in and accessed from the correct location, to ensure the appropriate data is used and files are not overwritten accidentally. The ‘use’ and ‘save’ commands both require extensive directory and sub-directory specification. The “*cd*” enables us to shorten the information required when files are accessed and saved.

```
cd H:/name of folder
```

more specifically, if all the relevant data was stored under “H:\Data\Stata\”, this could be specified as follows:

```
cd "H:\StataLevel2\"
```

Then, rather than accessing data using the full command:

```
use "H:\StataLevel2\Raw data\bhps.dta", clear
```

we can take advantage of having specified a current working directory and access data using a shortened version of the file path, only specified the subfolders and file names within the working directory:

```
use "Raw data\bhps.dta", clear
```

Similarly, rather than using

```
save "H:\StataLevel2\Stata data\Final\bhps1.dta"
```

data can be saved using the following command:

```
save “Stata data\Final\bhps1.dta"
```

You can always display the working directory by typing:

```
pwd
```

2 Intermediate data manipulation commands in Stata: (egen; _n & _N)

2.1. Egen (extensions to generate)

Egen (egenerate) is a very useful command with lots of different options. It is also often helpful to combine egen with bysort in order to do analyses by group. Here, we will work through some of the most common egen choices but there are several others – the Stata manual and Stata Help outline these in detail.

This first example calculates the mean annual household labour income for the households in the dataset (i.e. column sum of the labour income variable):

```
/*Reopen main working file*/
```

```
use "$work\bhps_demo.dta", clear
```

Step 1 **/*egen mean: mean of the annual household labour income column*/**

```
egen mean_labinc=mean(inc_lab)
```

The structure of the egen command is as follows. Literally, it is saying: **make a new variable** called **mean_labinc** and make it **equal** to the **mean of** the **variable inc_lab** (annual household labour income). What this will do therefore is give every case the same value for this new variable.

The second case uses the total as opposed to mean (this particular example is just an illustration):

Step 2 **/*egen total: column sum of annual household labour income*/**

```
egen total_labinc=total(inc_lab)
```

This syntax tells Stata to make a new variable called mean_labinc and for each case make it equal to the column total of the variable inc_lab in the dataset. Other common uses of egen are:

```
/*row total - adding up all income components for each
household (i.e. each row of data*/
egen rowtotal_labinc=rowtotal(inc*)
```

```
/*returns minimum value*/
egen min_labinc=min(inc_lab)

/*returns maximum value*/
egen max_labinc=max(inc_lab)

/*median*/
egen median_labinc=median(inc_lab)

/*rank*/
egen rank_labinc=rank(inc_lab)

/*standard deviation*/
egen sd_labinc=sd(inc_lab)

/*count number of non-missing cases*/
egen nonmiss_inc_lab=count(inc_lab)

/*mean absolute deviation from mean*/
egen mdev_inc_lab=mdev(inc_lab)

/*median absolute deviation from median*/
egen mad_inc_lab=mad (inc_lab)
```

Step 3 /*bysort egen mean*/

We may be interested in whether the mean annual household labour income in London is likely to be different to the mean annual household labour income in the North East or Midlands. Therefore, it may be useful to calculate the mean annual household labour income for each region. This can be done either by using the collapse command (shown in Section 3) or by combining egen with bysort. The answers will be the same in each case. If the egen command is used with bysort then a new variable is added to the existing data file and the variable created takes the same value for each case within the same category of the by group.

This example calculates the mean annual household labour income within each region:

```
bysort region: egen reg_mean_labinc=mean(inc_lab)
```

or, equivalently,

```
bys region: egen reg_mean_labinc=mean(inc_lab)
```

In this example all people within the same region therefore take the same value for mean labour income (given that it is the mean labour income of that region).

In order to have a feel of the new variable, codebook or describe it:

```
/* codebook for the new variable reg_mean_labinc */
codebook reg_mean_labinc
```

/*which are the regions with the highest annual household labour income and which with the lowest */

```
tab reg_mean_labinc house_type, col nofreq
```

This final example below uses the rowtotal option to identify low-income households (i.e. their income falls below 60% of the median income):

Step 4 /* **calculate total household income***/

```
egen tot_hh_inc=
 rowtotal(inc_lab inc_nonlab inc_pens inc_bens inc_inv)
```

This creates a row total of the variables included in the brackets. This is similar to, but not always equal to, simply adding the variables because `egen` **ignores missing values when summing** whereas writing

```
generate tot_hh_inc_sum =  inc_lab + inc_nonlab….
```

would be equivalent when no missing values are contained in the variables. However, if at least one observation is missing, the `generate` command would calculate the result as missing.

Next calculate median income and create a binary variable (in this case, unusually, a binary string variable) to identify households living above and below median household income:

Step 5 /* **calculate median income***/

```
egen median_hh_inc=median(tot_hh_inc)
```

Step 6 /*identify households that live below a certain income threshold using a string variable (lowincome)*/

```
gen deprived=""
replace deprived="lowincome" if ///
 tot_hh_inc < (0.6*median_hh_inc) & tot_hh_inc !=.

replace deprived="Total income >= 60% of median" ///
 if tot_hh_inc >= (0.6*median_hh_inc) & tot_hh_inc !=.
```

Here a string variable is created simply to demonstrate how string variables (categorical variables) work but it is usually easier to work with numeric variables rather than string variables where possible (you will have to destring string variables for any analysis).

2.2. _n and _N

_n and _N are used to make lags and leads: they are useful but can take a bit of getting used to. _n and _N do not exist as variables and it is not possible to see them (unless you write something like gen n=_n) but Stata understands them in the following way. **_n refers to the observation's row number**; for example, if an observation is on the top row in the dataset then _n=1, if it is ten rows down then _n=10, and so on. **_N refers to the total number of observations**. Therefore, in a dataset each observation has a unique value for _n (its row) but all share the same value of _N (total observations in the data). Sorting the data in different ways will therefore alter the value of _n but not the value of _N.

_n can be used in a **relative** and an **absolute** sense. In this first example _n is used in a relative sense to work out the difference between each household's total income and the total household income of the household below it in the income distribution. That command has very useful applications for example if you want to conduct different quarterly comparisons, etc and provided that your data is in wide format or that your long data format does not contain duplicate values.

Step 7 /* sort by total household income to get the correct ordering of households by total household income*/

```
sort tot_hh_inc
gen prior_diff = tot_hh_inc - tot_hh_inc[_n-1]
```

In the **relative** sense above _n is used in relation to other rows around it (i.e. _n-1 in this case, but we could just as equally write _n-2, _n-3, etc., or _n+1, _n+2, etc. if necessary).

NB. Sometimes it might be easier to use Stata's time-series operators: L. (for lags); F. (for leads), D. (for differences) and S. (for seasonal differences). Unlike the use of _n, time-series operators will never

misclassify the observation. Expressions such x[_n] – x[_n-1] could be cumbersome and dangerous: consider the following example. You have data from 1981, 1982, 1984 and 1985 with data for 1983 missing. When constructing the lag with the [_n] expression, Stata will assume that the lag of 1984 is 1982 and the first difference will incorrectly span the two-year gap. The time series operator will not make such a mistake. See Baum (2009) for further information.

_n can also be use in an **absolute** sense and this is done below to generate new variables with values of the lowest and highest values of labour income in the data:

Step 8 **/*calculate the lowest and highest labour income in the data*/**

```
sort inc_lab
gen min_inc_lab=inc_lab[1]
gen max_inc_lab=inc_lab[_N]
```

The first generate command makes a new variable called min_inc_lab and sets this equal to the value of inc_lab of the first row in the data. The second generate command does the same thing with the last observation in the data – note that we don't need to work out what row that actually is because whatever it is it will be equal to _N. Clearly, it is important that the data is sorted in the correct way when using _n and _N, and you should be aware of missing values as these will be the final numeric values of ascending sorted (i.e. sort) numeric variables.

It is also possible to combine using _n and _N with bysort introduced earlier – this would be the case for instance if we wanted to know how many cases were in each region (using _N with bysort) or if we wanted to identify the household in each region with the lowest labour income (using _n==1 with by). If you are using bysort with _n and _N then _n is understood as the observation row number within the bygroup rather than within the whole dataset, and _N is similarly understood as the total number of observations within each bygroup.

To work out how many observations in each region,

```
bys region: gen region_cases=_N
```

To identify the most deprived household in each region in the data according to labour income,

Step 9 /*this sorts by region first and then orders the cases by income within each region*/

```
sort region inc_lab
```

/*this then uses by (**NB NOT bysort**) with the sorted data to identify the first case in **each region – note here that the 'by' applies only to the region**

variable (given that the previous sort has sorted by income within region)*/

```
by region: gen region_lowest_income=1 if _n==1
```

These examples show the usefulness of _n and _N and their ability to be used in both an absolute and a relative sense. It should be noted that it may often be easier (and safer) to use bysort with egen instead of _n and _N in an absolute sense. For example,

```
bys region: egen reg_max_inc = max(tot_hh_inc)
```

is easier and safer than

```
sort region tot_hh_inc
by region : gen reg_max_inc=tot_hh_inc[_N]
```

Exercise 1 Understanding the Egen Command and the role of _n & _N (25 mins)

- *Practice using the egen function*
- *Gain familiarity using _n and _N*
- *Use the car_data.dta dataset, which is saved in H:\StataLevel2\Raw data*

Task 1

- Set up a global macro pointing to the raw data folder as described above.
- Open the car_data.dta set using the global macro you have created.

  Hint: the relevant Stata code is described in section 1.5.

Task 2

- Use *egen* to create a new variable which equals the median value of all car prices (price). Name this new variable price_median_egen.

  Hint: the relevant Stata commands are described in section 2.1.
- Now try performing the same task with *generate* instead of egen.

  Hint: first find out the median of the price variable. Do this by using the detail option of the summarize command. Then generate a new variable and set it equal to the median price as shown in the output window.
- Compare the two new price variables you created using the *summarize* command. Are they different?

Task 3

- Use *egen* to create a total cost variable summing up the three cost variables. Name the new variable cost_total_egen.

  Hint: use rowtotal. You can use the describe command to find out the variable names.
- Now try performing the same task with *generate.*

  Hint: this requires using mathematical functions (+) to add variables together and the generate command.

  Compare the two new total income variables using the *summarize* command. Are they different? If so why?

  Hint: look at the number of non-missing observations available for both variables.

Task 4

- Create a new variable that takes the value of the mean labour cost (cost_lab) for domestic and foreign cars (foreign variable). Name this variable foreign_mean_cost_lab.
- This variable should take the same value for all domestic cars, and another value for all foreign cars.
- *Hint: perform this task by combining the egen* command with *bysort*
- Which category has the lower mean labour cost?

  Hint: you can find this information by tabulating the variables foreign and the newly created variable foreign_mean_cost_lab, or by looking at the Data Editor.

Task 5

- Create a new variable which shows the total number of cars for different headroom categories. Perform this task in two different ways:

  -Firstly, do this by using the *egen* command, and call the new variable headroom_cases_egen. Make use of the count function.

  -Secondly, do this by using the *generate* command, and call the new variable headroom_cases_gen. Make use of the _N function.

- Tabulate each new variable separately to find out the number of observation within each category.
  Hint: both methods should generate the same results. Information the use of _n and _N can be found in section 2.2.

Task 6

- Use _n to work out the difference between a car's total cost and the total cost of the next more expensive car. To do this use the cost_total_egen variables you created in Task 2.

  Hint: ensure that you have sorted the data by cost_total_egen first using the sort command.

- Note: an alternative is to use lags. For more information on lagged variables see the Stata help documentation.

3 Functions

There are many different functions and these relate to string, numeric, date and other variable types. Many are very specific but it is worth having a look through them. Some examples of functions are given here just to give a flavour of the ways that functions work and to give an indication of the type of task they can be used for. Searching on 'functions' in the help menu in Stata shows all the different types of functions you can use.

3.1. Numeric functions

To create a new variable which is equal to the value of the highest of a number of variables or numbers the max function is used:

```
gen max = max(inc_lab, inc_nonlab, inc_inv)
```

Square root of number or variable uses the sqrt function:

```
gen sqrt = sqrt(inc_lab)
```

Rounding is commonly needed. To round to the nearest whole number

```
gen round_inc = round(inc_lab, 1)
```

and rounding to one decimal place would be

```
gen round_inc3 = round(inc_lab, 0.1)
```

3.2. Random number functions

`uniform` is used to make random numbers in Stata. `uniform()` returns uniformly distributed pseudorandom numbers on the interval 0,1.

If you wish to make pseudorandom numbers over a range *a* to *b* rather than 0 to 1 then the syntax is:

```
gen newvar = a + (b – a) * uniform( )
```

and to make random integers over the range *a* to *b*:

```
gen newvar = a + int((b – a) * uniform( ))
```

So, for example, to make a variable called new equal to a variable called old plus or minus some integer value within the range of -5 to +5 we would type:

```
gen new = old + (-5 + int((5 - - 5) * uniform( ) ))
```

It is often the case that whilst we wish to use a random number in syntax we also wish to be able to rerun the syntax again in future and be able to get the same random number – and hence the same results – each time the syntax is run. uniform() can be seeded with the set seed command – this is helpful when the syntax will be rerun as it ensures that the same random numbers will be generated for each case in the future, thus ensuring the same results can be reached again. To do this we simply set the seed prior to the use of uniform:

```
set seed 3

gen new = old + (-5 + int((5 - - 5) * uniform( ) ))
```

seed does not need to be 3 – it can be any number. What is important is that prior to running the `uniform( )` function Stata will set the seed to the same number each time – setting the seed does not affect the randomness of the `uniform( )` function but allows results to be replicated each type the syntax is run.

3.3. String functions

As with functions generally, there are many string functions and here only those which we have had need to use most commonly are presented. Three particularly useful string functions are trim, substr (substring) and subinstr (subinstring).

The trim function deletes leading and trailing blanks from string variables – this is particularly useful when merging with string variables and when the merge is not working as you expect. Common reasons are blanks either at the start, within or at the end of either of the string variable – trim and subinstr (below) combined deal with this. The trim function is easy to use – here we make a new variable which has trimmed any blank string characters from the front and back of the region variable, creating a new variable (int_place_trimmed) and leaving the original untrimmed variable intact (int_place):

```
generate int_place_trimmed=trim(int_place)
```

The substr function allows you to make a new variable based on some part of an existing variable:

```
/*substr*/

gen deprived=substr(lowincome,1,1)
```

The syntax structure is as follows. Make a new variable called deprived using the substr function. In the brackets, we stipulate what the values of the new variable should look like. There are three elements to specify: first, the variable that the new variable is to be based on (lowincome); second, at what point in this variable to start making the new variable from (in this case from the first character); third, how many characters to read from this starting point. Therefore, as 'lowincome' takes the values "lowincome" or "above10000" then 'lowincome2' will equal "l" or "a" (i.e. based on lowincome, starting at the first character, and reading one character). The second option in the brackets can be a negative number and this will read the stated number of characters from the end of the original variable rather than from its start.

This can be used to, for example, to perform a function only on cases where the interview month begins with a certain letter. For example, this can be useful if we have a list of unique area codes which all begin with "E" if they are in England and "S" if they are in Scotland: we could use substr to only keep cases where the first letter is an "E" for instance if we were only interested in English cases. In this dataset we might decide (though in reality it seems unlikely!) that we are only interested in people interviewed in their 'office' or 'other place' (i.e. not home). We can identify these cases with the following:

```
gen int_place_o=1 if substr(int_place,1,1) == "o"
```

Subinstr works similarly except that this function replaces each occurrence of a character within a string variable with a different character. The syntax is as follows:

```
gen newvar=subinstr(oldvar,"old_text","new_text",no_of_instances)
```

One common use of this command is to remove spaces in text in a string variable and to remove leading and trailing blanks. For example, if the value for deprived was written as "Above 10000" and we wanted it to be written as "Above10000" then subinstr could be used to achieve this as follows:

```
gen deprived3 = subinstr(deprived," ","",1)
```

What this syntax will do is to look at the variable specified (deprived) and replace the character " " with the character "" the first time it encounters " " and only on this occurrence (i.e. if the variable has two blanks in it then only the first one will be removed). Naturally, " " and "" can be any characters. If we were not certain that there would always be a maximum of one blank then we could change the 1 to some higher value to be sure that the syntax changed all occurrences of blank spaces.

3.4. Tostring and destring

These two commands are used to convert numeric variables into string variables (tostring) and, inversely, to convert string variables to numeric variables (destring).

For example, assume that the first 3 digits of the hhid variable related to the area in which the household was located and what we wanted to do was to create a new variable equal to this area code. We can do this using the string functions outlined above but first we need to convert the hhid variable from a numeric to a string variable using the tostring command:

```
tostring hhid, gen(hhid_string)
```

In this syntax we first specify the variable we wish to convert (hhid) and then we specify the generate option and state the name variable we wish to create (hhid_string). Using this syntax the original variable (hhid) remains untouched and a new string variable is simply created: hhid_string is a string variable which is identical to hhid except that it is not numeric. We could now use the substr command to make a new variable equal to the first 3 characters, first trimming out any leading string blanks to be safe:

```
gen hhid_string2=trim(hhid_string)

gen hhid_area=substr(hhid_string2,1,3)
```

Now we can use destring to make this a numeric variable, and notice that because we are happy to write over this variable we use the replace rather than the generate option as in the example above:

```
destring hhid_area, replace
```

4 Other Intermediate Data Manipulation Commands

4.1. Rename

In the *Stata: An introduction to data access and management* course, the *rename* command was introduced to change the variable names of existing variabels:

```
rename old_name new_name
```

The command can be extended to rename several variables at the same time, in the same command line:

```
rename (old_name1 old_name2) (new_name1 new_name2)
```

The command can also be used to change parts of variable names, and be used to change several variables simultaneously. For this, * and ? can be used as wildcards, where each ? represents exactly one variable, and * can represent zero or more variables.

For example, if we wanted to change the name of all variables beginning with inc_ (i.e. inc_total, inc_lab etc.) to start with income_ (i.e. income_total, income_lab etc.), the following code could be used:

```
rename inc_* income_*
```

The following command removes inc_ if it has been used anywhere in a variable name (prefix, midfix, suffix):

```
rename *inc_* **
```

In comparison, the following command replaces the 'jan' with 'January' only in variable names that have exactly one character in front of, and one character behind the 'jan' component of the variable name:

```
rename ?jan? ?January?
```

Typing 'help rename group' into the Stata command window will show other available option for the rename command.

4.2. Move and order

It is often desirable to reorder the variables within the dataset for some reason, and there are three commands which can do this.

If we wanted to pick up one variable and move it to a particular place in the dataset then move will do this for us. For example, let's assume that we wanted to move the tenure variable to now be the second variable in the dataset. To do this we would type:

```
move tenure int_day
```

In this syntax move relocates variable one (tenure) to the position of variable two (int_day – the second variable in the dataset) and shifts the remaining variables, including variable two (int_day), to make room.

Alternatively we can accomplish the same thing by simply using order. For example we would type:

```
order tenure, before (int_day)
```

This newer Stata syntax, simply states that we want to place tenure before int_day.

order is used either to bring variables to the front of the dataset or to specify the exact order in which a list of variables are to appear in the dataset. The variables specified are moved, in order, to the front of the dataset. Hence,

```
order rooms hhcost hhvalue toilet* hhsize age
```

places these variables at the start of the dataset in the order listed, with all other variables being moved along to make way for them. Where an asterisk is used (as in the case of toilet*) then all of the variables picked up (in this case toilet_indoors and toilet_shared) remain between themselves in the same order as they originally appear and are moved as a group to the new position.

```
order _all, alphabetic
```

alphabetizes all of the variables specified. Finally, it should be noted that the order command has recently received additional useful options. As is the case for all Stata commands, these options are described in the help file by typing “help order” in the command menu.

4.3. Erase

The erase command erases data files – it can be useful in syntax files if lots of temporary files are made for some reason which can then be deleted after they have served their purpose if space on the hard drive is an issue. It is a simple command to use. This example erases the file we have just made:

```
/**Erase files**/

erase "$work\Stata level 2 working file two.dta"
```

4.4. Calling a .do file from within a .do file

Should you wish to, it is possible to call another .do file to run from within an existing .do file and this is very simple to do. This can be useful if your first .do file runs out of space (it would need to be an extremely long .do file though) or, for example, you wish to run different long .do files of syntax following branching ‘if’ statements.

To call a new .do file simple type the do command followed by the location of the .do file to run:

```
do "H:\StataLevel2\logs & syntax\new do file.do"
```

On encountering this in the original syntax file, Stata will begin running the specified .do file and, once this is completed, will continue from that point onwards in the original .do file.

Exercise 2 Variable Manipulations (15 mins)

- *Practice how to move and order variables*
- *Practice basic data manipulation commands*
- *Gain familiarity with the destring and tostring functions*
- *Use the car_data.dta dataset, which is saved in H:\StataLevel2\Raw data*

Task 1

- Create a new variable called cost_mat_1d. This variable should take the values of the material cost variable (cost_mat), but is rounded to one decimal place.
- Create a new variable called cost_mat_0d. This variable should take the values of the material cost variable (cost_mat), but is rounded to zero decimal places.

  Hint: Details of the relevant Stata command are provided in section 3.1.

Task 2

- Create a new variable (make1) which is equal to the first three characters of the string variable make.

  Hint: You will need to use the substring function. Details of the relevant Stata command are provided in section 3.3.

Task 3

Often, the data you will receive may not be in the appropriate format for you to work with:

- Display summary statistics of the weight variable

  Hint: you will receive an error message: the required output cannot be generated. Check what format the variable is in.
- Transform the weight variable into numeric format (generate a new string variable called weight_num).
- *Hint: You will need to use the* `destring` *function. Details of the relevant Stata command are provided in section 3.4.*
- Look at the error message displayed in the output window.
- Look at the weight variable in the Data Editor. Replace the observation that is not a number with a missing value (for text, missing values are defined as "", i.e. empty quotation marks).
- The `destring` command should now work.
- Finally, display summary statistics of the weight_num variable.

Task 4

- Use *rename* to change cost_lab and cost_mat (and all other variables starting with cost_) to c_lab, c_mat etc.

  Hint: Details of the relevant Stata command are provided in section 4.1.

Task 5

- Bring the cost variables (now starting with c_) towards the front of the dataset so that they are the next variables after the make variable.

  Hint: Details of the relevant Stata command are provided in section 4.2.

5 Commands that change the shape of the data (merge; append; duplicates; collapse; expand; xpose; reshape)

5.1. Merge

The merge command is used to add variables from one dataset into another dataset where they share a common characteristic (e.g. they relate to the same person, household or area). This can be imagined as extending the dataset sideways by adding new variables. The two datasets involved in the merge have names in Stata: 'master' and 'using; dataset. When merge is used one dataset is open and this is the 'master' data. Data from a second dataset (in memory) is merged into the master data and this second dataset is called the 'using' data.

It is possible to just stick two datasets together without a linking variable (i.e. add the first row of data from the using dataset to the first row of data in the master data), but this is unusual and risky. Typically, merge works by merging according to a common link variable (or variables) which identifies the case in both the master and using data – both the master and using datasets must be sorted by the link variable(s) immediately prior to the merge.

There are several types of merge to choose from but the three most popular are (1:1; m:1; 1:m). 1:1 stands for one-to-one merge and means that each particular value of the link variable occurs only once in the master dataset and only once in the using dataset (i.e. it is a unique observation in both); m:1 means many-to-one merge and happens when the value of the link variable occurs multiple times in the master dataset but that it occurs only once in the using dataset; 1:m means one-to-many and happens when each value of the link variable occurs once in the master dataset but multiple times in the using dataset. Most merges tend to be either 1:1 or m:1. It is possible to run the merge without specifying what kind of merge it is – this is usually safe to do as Stata would give you an error if not - but it is good practice to be clear about what kind of merge you are doing and to therefore specify this.

To illustrate this, imagine the two (admittedly small) datasets below, the master dataset being on the left and the using dataset being on the right. This example merges individual's ages onto their names and as the link variable (the individual's id) occurs only once in both master and using datasets it can be seen that this is a 1:1 merge:

id	name
1	John
2	Mary
3	Sharon
4	Kevin

id	age
1	23
2	14
3	46
4	67

In this second example, below, in the master data there is household number, the area the household is in, and the population of that area. Note that area 2 appears three times in the master data as three households within this area are in the data. We want to merge in the area unemployment rate. Note that in the using data lookup table area 2 appears only once. This cannot therefore be a unique merge because area 2 is not unique in the master data. Rather, this is a m:1 merge because the values of the link variable '*area*' are not unique in the master dataset but are unique in the using dataset. What will happen is that the area unemployment variable will be merged onto all instances of area 2 which occur in the master data, therefore giving both household 45 and household 46 the value of 10 for area_unem.

hh	area	area_pop
45	2	150
46	2	150
79	2	150
126	5	350

area	area_unem
2	10
4	7
5	5

As an example, assume we want to summarise the income data we have in our working file by region. Let's drop the region variable from the file and pretend we don't have it and need to merge it in:

```
drop region
```

Since we now do not have a region variable in this file we need to merge in the region variable from the BHPS sample file (also in the raw data folder) before we can carry out any analyses by region. To do this we merge the master data with the using dataset in which the region variable is located (bhps sample file.dta), specifying the link variable which (in this example uniquely) identifies the observations in both datasets (hhid):

```
/*Open main dataset and merge region variable from bhps sample file with hhid as the link variable for the merge*/


use "$raw\bhps_demo.dta", clear

drop region

/*a file that for example does not contain the variable region*/


merge 1:1 hhid using "$raw\bhps sample file.dta”
```

Note the syntax structure: the merge command comes first and this is followed by the type of merge that is going to be performed, 1:1 in our case. Then, the link variable(s) which Stata is to merge by is specified, followed by the path for the using dataset using the global macro raw that we set up at the start of the syntax file to identify the correct folder that the data is in (or just the correct path). With the 1:1, m:1 and 1:m, data does not need to be sorted beforehand.

In addition to merging in the variables in the using dataset, after each merge Stata creates a new variable called _merge. This is a useful variable for checking that the merge and the matching has worked properly and it is advisable to analyse this variable after each merge, either by running tab _merge or by opening up the browse window, sorting by _merge and checking the values or just list the data. _merge can take values of 1, 2 or 3. If for an observation in the data the link variable (in this example hhid) existed in the master data file only then _merge will equal 1; if for an observation the link variable existed in the using data file only then _merge will equal 2; if for an observation the link variable existed in both the master data file and the using data file then _merge will equal 3. Often this _merge variable is used to discard unnecessary observations, for example, if you only wanted to keep observations that are in both the master and using files (e.g. keep if _merge==3). In our example therefore if an observation has _m==1 then we know about their household characteristics but not about which region they are in; if _m==2 we have a region variable for the household but we do not actually have that household's characteristics in the main data file; observations with _m==3 have both the characteristics and the region variable as they appear in both files.

Let's assume we are interested in regional analyses and so only cases for which we have characteristics and a valid region variable will be any use, so only keep these:

```
/*keep only those observations which had non-missing values
i.e. _merge==3*/

keep if _m==3
```

It is also possible to merge on more than one common link variable, for instance when the data has individuals within households and where we therefore need both the household and the individual variables in order to uniquely identify individuals. Therefore, if we wanted to merge using a unique individual identifier we would have to use both the household and individual identifiers as the link variables.

5.2. Append

Whilst merge adds variables to the side of a dataset, append adds cases to the bottom of a dataset. In order to do this correctly the variables need to have the same names in both files. Append is a simple command to use – data does not need to be sorted and it is not necessary to append by any link variable as we are simply tacking cases with shared variable names onto the bottom of the dataset. In order to demonstrate the append command we will take the first 100 cases from our working file, identify them with a new variable, and append them back into the bottom of the file (so that they appear twice in the file).

```
use "$raw\bhps sample file.dta", clear
```

Next, take the first 100 cases, make the variable append_flag and set it equal to one for these cases. Save this as a separate file to append:

```
keep  in  1/100
```

*make a variable to show this observation is from the append file

```
gen append_flag=1
```

*save for append

```
save "$work\data for append.dta",replace
```

Next, open the main working file we saved above and append in the data just saved:

```
use "$raw\bhps sample file.dta", clear
append using "$work\data for append.dta"
```

Sort by the append variable and analyse the results of the append/

```
sort append_flag
```

All of the other variables except append_flag had the same variable name as existing variables and so just slotted on to the bottom of these variable columns. The append_flag variable was the only variable which did not exist in the master data and so Stata has added this as a new variable in the data and given a value of missing to all of the cases in the master data file as these did not previously contain this variable. As missing is the highest numeric value, the 100 cases we appended come to the top of the data when the data is sorted by the append_flag variable.

5.3. Duplicates

'Duplicates' is a data manipulation command rather than one which changes the shape of the data but it can be used after append and so we discuss it here. The 100 cases just appended to our original file are now duplicated within this dataset, enabling us to show the duplicates command. The duplicates command can be used to identify (tag) or to remove (drop) duplicate cases (it can also do other things besides these but these are common uses). If we wanted to know which cases (i.e. observations of hhid) were duplicates then we could write:

```
duplicates tag hhid, gen(dup_flag)
```

This would count up the number of duplicate cases of each observation. Hence, unique cases would take the value zero for the dup_flag variable, cases with one duplicate case would take the value one for dup_flag, and so on.

Alternatively, we could use the duplicates drop option to automatically drop these cases. It is possible to ask Stata to look for duplicate values on more than one variable and this is often safer because doing so only identifies (or drops) cases with identical values for *all* of the specified variables. Therefore, if we had not wanted to identify duplicates and then drop them as above but instead wanted to simply drop them straight away if they were duplicates, and if we wanted to look for duplicates across multiple variables in order to be safe, then the syntax would be:

```
duplicates drop hhid int_day int_month, force
```

The syntax starts with the duplicates command and we then tell Stata what we would like to do with any duplicates – in this case we want to drop them. Next we tell Stata within which variables to search for duplicates. In this case we use three variables: hhid (household id), int_day (day of interview) and int_month (month of interview). What we are saying to Stata is that if there are cases which have identical values for hhid, int_day and int_month then we are assuming that they are duplicates of the same case and that only one of them should be kept. The force option is required to tell Stata that we are aware that we will lose data by doing this. It can be seen in the output window that Stata tells us that 100 cases have been deleted – the 100 cases that we appended in to the data.

Note then that running duplicates tag hhid and then dropping cases with the value one for the resulting 'tag' variable is not identical to running duplicates drop hhid: in the former instance **all** cases which have duplicates are deleted (leaving **none** of these cases in the dataset) whilst in the second instance 'copies' of the case are deleted but one of the cases remains in the dataset.

It is necessary to be careful when using this command – if for instance we just wrote

```
duplicates drop int_month,force
```

then Stata would drop an awful lot of data (it would keep only the first occurring case of each int_month in the dataset) and this would almost certainly be an error in our syntax!

Now that these 100 duplicate cases have been deleted the append_flag variable can be dropped:

```
drop append_flag
```

5.4. Collapse

At this point we have a data file of cases in England with a simplified region variable merged in. The data are at household level but what if we wanted to look at the data at regional level? This could be done using egen with bysort as shown earlier. Another way this could be done is to use the collapse command to collapse the data to region-level rather than the household-level that it is currently at. Assume that we do want to collapse the data to region-level and that we want it to contain:

i) the total number of households in that region

ii) total_mortgage (total_mortgage)

iii) average monthly mortgage payment (monthly_mortgage)

To do this the syntax would be:

```
collapse (count) hhid (mean) ///
total_mortgage monthly_mortgage, by(region)
```

Collapse uses three main options – mean, sum and count. Be careful: mean is the default option and so if you do not specify what kind of collapse you want the Stata will give a mean of the variable(s) even if what you wanted were sums. In this example we specify that collapse should for each region count the number of cases of the hhid variable and give us the means for total_mortgage and monthly_mortgage. The by option tells Stata what variable to collapse by – in this case it is the categories of the region variable. This file is now at region level. Save this file:

```
save "$work\mortgage collapsed to region.dta",replace
```

5.5. Expand

The expand command is used to copy existing rows of data or as part of the process to create new rows in the dataset. It is not very often used and there may well be safer ways to do what you want, but it is possible to use it to make new observations if that is what you wish to do.

In our example above we will use expand to make a new row in our collapsed regional dataset. In this new row we will then sort the data and use _n to calculate the difference between the mean total mortgage in London and the Midlands.

First, to present the example clearly only the two relevant cases of London and the Midlands are kept:

```
keep if region==1 | region==4
```

Expand essentially copies existing rows of data, which is then often rest to missing and some new data values placed into this new row.

The simplest way to use expand is to simply new cases which are equal to *n* duplicates of the existing cases in the dataset, for example:

expand 2 (this makes one duplicate copy of each case in the original dataset)

expand *n* (this is therefore the generic syntax to make *n-1* duplicates of each case)

An alternative way to use expand is to specify the single case which you wish to make copies of rather than making copies of all cases in the existing dataset. We will use this type of expand in our example to make a new row of data equal to case 1:

```
expand 2 in 1
```

This syntax asks Stata to make 2 copies of row 1 and this results in the dataset now having a new case (row 3) which is a duplicate of the case on row 1. It would be perfectly possible to change this and to, for example, make a different number of new duplicates rows of a different case:

```
expand 15 in 2
```

would for example make 15 new cases in the dataset which are all equal to case 2.

Now we will reset all of the values in this new row equal to missing data to be avoid problems:

```
replace region=. if _n==3

replace hhid=. if _n==3

replace total_mortgage=. if _n==3

replace monthly_mortgage=. if _n==3
```

and then use the new row 3 to show the difference between the values in London and the Midlands:

```
replace hhid = hhid[1] - hhid[2] if _n==3

replace total_mortgage = ///
   total_mortgage[1] - total_mortgage[2] if _n==3

replace monthly_mortgage= ///
   monthly_mortgage[1] - monthly_mortgage[2] if _n==3
```

5.6. Xpose

Xpose is the command to transpose data from columns to rows, or vice versa, and it works just the same way as transposing data in Excel. It is a very easy command to use. Let's open the region-level file that we created by collapsing our household level data and then transpose this:

```
use "$work\mortgage collapsed to region.dta",clear

xpose, clear varname
```

All that is required is the command itself and then the clear option. The clear option is required and this is to warn you that the original data file prior to the transpose will be permanently lost (unless you have saved it prior to running xpose). The varname option helpfully carries the variable names through to the transposed dataset. The resulting dataset contains all the same information as the original dataset except that the cases (i.e the regions) are now columns rather than rows – region was in column 1 so regions are now in row 1, hhid was in column 2 so this is now in row 2, and so on. Note that variable names are lost during the transpose and so it is sensible to rename the variables after transposing.

5.7. Reshape

Finally, the command is used to change the shape of the dataset from long to wide format, and vice versa.

By wide format what is meant is that each unit of analysis (e.g. people) have one row of data and on this row are multiple observations which usually relate to data for variables (e.g. income and savings) at different time points. For example, our variables in a wide format could look as follows in the dataset:

/* wide format*/

Id	income03	income04	income05	savings03	savings04	savings05
1	300	350	400	50	20	25
2	500	300	250	80	35	20

By long format what is meant is that the case (e.g. person) has multiple rows of data, that there is a single variable called income in this example, and that each year of income has a different row in the dataset. So, for example, the data above could equally be presented in long format as follows:

/* long format */

id	year	income	savings
1	03	300	50
1	04	350	20
1	05	400	25
2	03	500	80
2	04	300	35
2	05	250	20

In order to demonstrate and explain this command we will use a small artificial dataset of income and savings in different years for ten people:

```
use "$raw\reshape data.dta", clear
```

The dataset is currently in wide format. To reshape it to long format the syntax would be:

```
reshape long inc sav, i(id) j(year)
```

where ‘long’ specifies the type of reshape we wish to perform (i.e. the format we wish to reshape to), ‘inc’ and ‘sav’ are the variables to reshape (notice that suffixes relating to years are not needed), ‘i’ relates to the unique identifier for each case (e.g. person id) within which sub-observations (e.g. multiple income observations over different years) fall. Note that ‘j’ in the above example does not exist in the wide format dataset but year becomes the ‘j’ variable (i.e. that which identifies sub-observations ‘j’ within each case ‘i’) in the long format dataset. In order to use reshape it is necessary to be able to identify what the ‘i’ and ‘j’ variables are.

In terms of the variables included in the reshape command, here our dataset contains only variable relating to income (inc) and savings (sav) and these variables are both included in the syntax. If there had been other variables in the original dataset which we had not specified in the reshape command then these variables would remain unchanged in the reshaped data format and would be repeated.

To get back to a wide format after having reshaped the dataset to long format we can simply type the opposite reshape command without arguments to get back to the original dataset format, and this, in fact, does not need to be done immediately after the initial reshape because Stata will remember what reshape commands have been used previously in the session and will seek to reverse that:

```
reshape wide
```

Alternatively, if we had started with the data in long format and wanted to specify the reshape to take it to wide format the syntax would be:

```
reshape wide inc sav, i(id) j(year)
```

If the ‘j’ variable is a string variable then the string option can be specified to deal with this. For example, if we had data on husbands and wives (string variable ‘sex’ is the ‘j’ variable) within households (numeric ‘hh’ is the ‘i’ variable) then the syntax would be:

```
reshape wide inc sav, i(hh) j(sex) string
```

It is possible to have more than one ‘i’ variable where unique cases are nested (e.g. patient id within hospital id, person id within household id). It is also possible to have more than one ‘j’ variable – consider the case for instance where households are the ‘i’ variable and both sex and year are the ‘j’ variables which relate to the income variables in several years of male and female partners in the household. Where there is more than one ‘j’ variable there are many more possible ways in which the data can be reshaped as both of the ‘j’ variables can be set up in a long or wide format, and the reshape syntax is consequently slightly more difficult to specify correctly in order to obtain the desired format. This is a more advanced topic and is not dealt with here but the Stata manuals and help menu cover this issue.

Exercise 3 Data Manipulations (25 mins)

- *Practice merging and appending datatsets*
- *Gain familiarity with identifying duplicates and collapsing data*

Task 1

- The file trial_data contains data from a clinical trial. The treatment allocations for the patients are saved in a separate file, trial_data_rand. Both files are saved in "H:\StataLevel2\Raw data\.
- Open both files separately using the global macro set up previously. Find out what the unique patient identifier variable is called.

Task 2

- Using the above identified variable, merge the information from both datasets. Use trial_data_rand as the master set.
- Considering the _merge variable, data for how many patients is available in both datasets, and data for how many patients is only available in the using dataset or the master dataset?
  Hint: you can view this information directly from the merge output, or you can tabulate the _merge variable. More information on Stata's merge command is provided in section 5.1.
- Now try and use the append command to add the dataset trial_data to trial_data_rand. The newly created dataset is not useful in this context, but consider when you would need to use the append command.
  Hint: Information on the append command is provided in section 5.2.

Task 3

- Tag duplicates of the patient identifier variable (id). Are there any duplicates?
  Hint: use the duplicates tag command. The number of duplicates can be viewed by tabulating the variable generated in the command, or by using the duplicates report command.
  Information on Stata's duplicates command is provided in section 5.3.
- Check how many duplicates for the variable age there are.
  *Hint: u*se the command *duplicates report*.
- Flag the duplicates for age (generate a new variable called dup_flag_age). Use the *tab age dup_flag_age* command to show the number of duplications per age.

Task 4

- Use the *collapse* command to produce a file with the mean for the variable *studytime* for each drug. Which drug has the lowest mean value for *studytime*?
  Hint: information on Stata's collapse command is provided in section 5.4.
- Note that we could have found the same result by running bys drug: *egen newvar = mean(studytime).* The only difference is that with *collapse* command the dataset is changed.
- Extend the collapse command to also show the number of observations within each drug group.
  Hint: you need to read in the trial_data again, and merge it as above. Use the count function to arrive at the number of subjects in each drug group.

Task 5

- Open the file reshape_test
- The dataset contains data on income and outgoings over three years.
- Drop the variables sex and out2012, out2013, out2014 (these variables provide information on monthly outgoings).
- Reshape the data into long format, so that there is one variable for income, and a new separate variable for the relevant years. Consider carefully which variables you have to specify for the i() and j() options.
- Use the full command (not a shortcut) to reshape the data into wide format.
  Hint: Information on Stata's reshape command is provided in section 5.7. the gender variable can be ignored in the programming, it will be reshaped automatically.
  Consider the information provided in the output window. This information is very useful in assessing if the command has worked as desired.
- The above reshape commands also work if you want to reshape more than one variable.
- Read the raw data in again. Programme the reshape commands as described above, but this time reshape the income and outgoings variable.

5.8. Simple Regression Analysis

Let us make our first steps in running a linear regression of price of house on number of rooms per house (you may want to first create a log of the house price variable generate housevalue= log(hhvalue). We will use the regress command, which lists the outcome followed by the predictors (we will have only one for the sake of the exercise: number of rooms).

regress housevalue rooms

```
      Source |       SS       df       MS              Number of obs =    6284
-------------+------------------------------           F(  1,  6282) = 2697.24
       Model |  777.46104     1  777.46104             Prob > F      =  0.0000
    Residual | 1810.74576  6282  .288243515            R-squared     =  0.3004
-------------+------------------------------           Adj R-squared =  0.3003
       Total |  2588.2068  6283  .411938055            Root MSE      =  .53688

------------------------------------------------------------------------------
  housevalue |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       rooms |   .2143517   .0041273    51.93   0.000     .2062608    .2224427
       _cons |   10.88504   .0215792   504.42   0.000     10.84274    10.92735
------------------------------------------------------------------------------
```

Note that the regression is based on only 6284 observations. Stata omits observations that are missing the outcome or any values for one or combination of the predictors. The number of rooms "explains" around 30% of the variation in house prices in Britain. We also see that increase the number of rooms with one is associated with an increase of 0.24 in log of house price.

Following a regression (or in fact any estimation command) you can retype the command with no arguments to see the results again. Just type reg.

5.9. Post-Estimation Commands

A number of post-estimation commands can be particularly useful. Consider for example predict, which can be used to generate fitted values or residuals following a regression. The command

```
. predict newvar, xb
(option xb assumed; fitted values)
(314 missing values generated)
```

generates a new variable of the predicted values for each observation. No predictions are made individuals with lack of observations on the predictor variables. (If the dependent variable was missing for an individual it would be excluded from the regression, but a prediction would be made for it. This technique can be used to fill-in missing values.)

6 Appendices

6.1. Answers to the exercises

Solutions are provided at the end of the session.

6.2. .do file for the session

```
*Stata log for Stata data manipulation and analysis
clear
capture log close
set more off

log using "H:\StataLevel2\Stata logs\StataDataManipulation.log", replace

/* Stata: Intermediate Data manipulation and analysis */
/* Date 05 Feb 2015 */

/*set libraries*/
/*raw data folder*/
global raw "H:\StataLevel2\Raw data"

/*working files folder*/
global work "H:\StataLevel2\Stata data\Work"

/*final data files*/
global final "H:\StataLevel2\Stata data\Final"

/*files can then be opened as follows: */
use "$raw\bhps household file.dta",clear

/* Accessing the working directory as an alternative: */
pwd

*set working directory:
cd "H:\StataLevel2\

/* Check working directory */
pwd
```

```
/* open a file within this directory: */
use "Raw data/bhps household file.dta",clear

/***************************************************************************
** Section 1: Intermediate data management commands
***************************************************************************/
/*open raw datafile*/

use "$raw\bhps_demo.dta",clear

/*1.egen mean: mean of the annual household income column*/

      egen mean_labinc=mean(inc_lab)

/*2.egen total: column sum of annual household incomes*/
      egen tot_labinc=total(inc_lab)
      egen rowtotal_labinc=rowtotal(inc*) /*row total*/
      egen min_labinc=min(inc_lab) /*returns minimum value*/
      egen max_labinc=max(inc_lab)  /*returns maximum value*/
      egen median_labinc=median(inc_lab)   /*median*/
      egen rank_labinc1=rank(inc_lab), unique   /*rank*/
      egen sd_labinc=sd(inc_lab)      /*standard deviation*/

      egen nonmiss_inc_lab=count(inc_lab)
      /*count number of non-missing cases*/

      egen mdev_inc_lab=mdev(inc_lab)
      /*mean absolute deviation from mean*/

      egen mad_inc_lab=mad(inc_lab)
      /*median absolute deviation from median*/

tab nonmiss_inc_lab

/*3. bysort egen mean*/
bysort region: egen reg_mean_labinc=mean(inc_lab)

/* codebook the new variable reg_mean_labinc */
codebook reg_mean_labinc
```

```
/*which are the regions with the highest annual
 household labour income and which with the lowest */

tab region reg_mean_labinc, missing
save "$work\workfile.dta"

/*4. bysort egen total income */

describe inc*
egen tot_hh_inc = rowtotal(inc_lab inc_nonlab inc_pens inc_bens inc_inv)
codebook tot_hh_inc

/* are the results the same as : */
gen tot_hh_inc_gen = inc_lab + inc_nonlab + inc_pens + inc_bens + inc_inv

su tot_hh_inc_gen tot_hh_inc
/* No, the results are not the same. gen cannot perform calculations when
at least variable is missing. egen treats missing values as 0 and still
generates results.
*/
/* what is the maximum value for each region */
bys region: egen reg_max_labinc1 = max(inc_lab)

/**Final example (rowtotal)- identify low income households using  a
 string variable (low income equals those that are leaving with below 60%
 of the median total income)**/
use "$raw\bhps_demo.dta",clear

/* step 1: calculate total household income*/
egen tot_hh_inc=rowtotal(inc_lab inc_nonlab inc_pens inc_bens inc_inv)

/*step 2: calculate median income*/
egen median_hh_inc=median(tot_hh_inc)

/*step 3: identify high income households using a string variable */
gen deprived=""
replace deprived="lowincome" if tot_hh_inc < (0.6*median_hh_inc) ///
 & tot_hh_inc !=.
replace deprived="Total income >= 60% of median" ///
 if tot_hh_inc >= (0.6*median_hh_inc) & tot_hh_inc !=.
```

```
codebook deprived

/************* _n and _N *************/
gen n=_n
gen N=_N
drop n N

/** _n in a relative sense - calculating income difference to next well
 off household **/

/*sort the households in terms of total income*/
sort tot_hh_inc

/*drop those with tot_hh_inc == 0 as likely due to missing data */
drop if tot_hh_inc == 0

/*now the data is sorted by income we can calculate the difference between
 each household and the one below it in the income distribution*/
gen inc_diff_prior= tot_hh_inc - tot_hh_inc[_n-1]


/** _n in an absolute sense **/

/*calculate the lowest and highest labour income in the data*/
/* NB would be safer to use bysort egen min/max to do some of these */

sort inc_lab

/* lowest in dataset */
gen min_inc_lab=inc_lab[1]

/* largest in dataset */
gen max_inc_lab=inc_lab[_N]

/* lowest per region - note we sort by two variables but only generate by
 one variable */
sort region inc_lab
by region: gen reg_min_tot_inc=inc_lab[1]
```

```
/* number of cases per region - note than when used with bygroups _N
relates to the cases per bysgroup rather than the whole dataset */
bys region: gen reg_cases=_N
tab region reg_cases

/* Identifying the most deprived household in the region according
to the labour income */
sort region inc_lab
drop if inc_lab == 0 | inc_lab == .

/* identifying the observation with the lowest labour income per region */
by region: gen region_lowest_income=1 if _n==1

/* extract the information: */
di "List the lowest labour incomes per region"
list region inc_lab if region_lowest_income == 1

/* be aware that it is often easier and safer to use bysort with egen
 instead of _n and _N in an absolute sense */
bys region: egen reg_max_inc = max(tot_hh_inc)

/* is simpler and safer than */
sort region tot_hh_inc
by region : gen reg_max_inc1=tot_hh_inc[_N]

/***************************************************************************
**Section 2:  Functions
***************************************************************************/
*The Stata help function is able to produce a full list of all functions:
help functions

use "$raw\bhps_demo.dta",clear

/**** numeric functions ****/

/*Maximum value of a set of variables or numbers*/
gen max = max(inc_lab, inc_nonlab, inc_inv)

/*Square root of number or variable*/
gen sqrt = sqrt(inc_lab)
```

```
/*Round to the nearest whole number*/
gen round_inc = round(inc_lab)


/* and rounding to 1 decimal place */
gen round_inc1=round(inc_lab, 0.1)

/* generating ages - need to be rounded down: */
gen round_down = floor(inc_lab)

/**** random number functions ****/

/* random number between 0-1 */
gen temp = uniform( )

/*pseudorandom over a range a to b rather than 0-1:
 gen newvar = a + (b - a)*uniform( )*/
gen new = 5 + (15 - 5) * uniform( )
su new

/* add a random whole number in range 5 to 15 to the temp variable */
/* NB 1. adding to an existing variable 2. int means random integers rather
than random psudonumbers */

gen new2 = 5 + int((15 - 5) * uniform( ))
su new2

*are these new variables reproducible?
* Compare
gen test1 = uniform( )
gen test2 = uniform( )
gen test3 = uniform( )
gen test4 = uniform( )
su test1 test2 test3 test4

/* set seed */
set seed 1408
gen test5 = uniform( )
set seed 1408
```

```
gen test6 = uniform( )
su test5 test6

/**** string functions ****/

/* trim */
generate int_place_trimmed=trim(int_place)

/* making deprived to show substr function- ie below 60% median labour
income */
egen med_inc=median(inc_lab)
capture drop deprived
gen deprived="Above 10000"
replace deprived="Lowincome  " if inc_lab < (0.6 * med_inc)

/* trim - trim leading and trailing blanks*/
gen deprived1=trim(deprived)

/*substr*/
gen deprived2=substr(deprived,1,1)
gen int_place_o=1 if substr(int_place,1,1) == "o"

/*subinstr*/
/*gen newvar = subinstr(oldvar,"old_text","new_text",no_of_instances)*/
gen deprived3 = subinstr(deprived," ","",5)
*up to 5 blanks per variable are removed
gen int_place2 = subinstr(int_place,"f","F",1)
gen int_place3 = subinstr(int_place,"f","F",2)

/*e.g. replace the first two o characters with X*/
gen deprived4=subinstr(deprived,"o","X",7)

/**other functions not covered in the course booklet include**/
/*reverse*/
gen deprived_rev=reverse(deprived)
/*proper (ie capitalise)*/
replace int_place2=proper(int_place2)

/********* tostring and destring *******/
tostring hhid, gen(hhid_string)
```

```
/* take off first three string non-blank characters for household area
variable,
 then convert back to numeric variable */

gen hhid_string2=trim(hhid_string)
gen hhid_area=substr(hhid_string2,1,3)
destring hhid_area, replace

/*********************************************************************
** Section 3: Other intermediate commands
*********************************************************************/
use "$raw\bhps_demo.dta",clear
/* using rename to change variable names, or part of variable names */
rename emp employment

*change all variables beginning with inc_ to beginning with income_
rename inc_* income_*

*rename several variables at the same time:
rename (int_month int_year) (interview_month interview_year)

/* move, order and aorder - changing the position of variables in the
dataset */
use "$raw\bhps_demo.dta",clear
/* moving one variable to the location/ in front of another variable */
move tenure int_day

*using order for same results:
use "$raw\bhps_demo.dta",clear
order tenure, before(int_day)

/*ordering a list of variables to appear in order from the start of the
 dataset*/

order rooms hhcost hhvalue toilet* hhsize
/* put all variables in alphabetical order */
order _all, alphabetic

/* save and show erase */
save "$work\stata level working file to erase.dta",replace
```

```
/******* Erase files ************/
erase "$work\stata level working file to erase.dta"

/***************************************************************************
**  Section 5: Commands which change the shape of the data
***************************************************************************/
/********* Merge **********/
/*can do straight merge (ie just slot them in sideways with obs 1 merging
 with obs 1 of another dataset) but almost never do. Normally merge by
 common variables (ie merge using)*/
use "$raw\bhps_demo.dta",clear


drop region
/*we want to summarise variables at regional level, but in order to do this
we  need to merge in the region variable we have just dropped from file
bhps sample file.dta*/


/*Merge region variable into the main working file*/
merge 1:1 hhid using "$raw\bhps sample file.dta"
*bhps sample file contains only hhid and region


/*Analyse the _merge variable which is created*/
tab _merge


/*keep only those observations which had non-missing values i.e.
_merge==3*/
keep if _m==3
drop _merge


/*save the file at this point for the append*/
save "$work\bhps working file three.dta",replace


/********** Append *************/
use "$raw\bhps sample file.dta", clear
/*keep the first 100 cases*/
keep in 1/100


/*make a flag to show this observation is from the append file*/
gen append_flag=1
```

```
/*save for append*/
save "$work\cases for append.dta",replace

/*Reopen main working file and append. NB append under existing variable
 columns where there are the same variable names, otherwise it will tack
 new variables on the side. Append does not need to be sorted first*/

use "$raw\bhps sample file.dta",clear
append using "$work\cases for append.dta"
/*Sort by the append variable and analyse the results of the append*/
sort append_flag

/*Now in this case we have taken the top 100 cases from this file and
appended them back into the same file, which clearly worked because
they have identical variable names. This means that 100 of the cases
are repeated and this allows us to show the duplicates command*/

/********** Duplicates **************/
duplicates report hhid
duplicates tag hhid, gen (dup_flag)
*show that the correct observations were tagged:
tab dup_flag append_flag, miss

duplicates drop hhid ,force

/* We can see that 100 cases are dropped, note too that the 'originals'
tagged with dup_flag remain and only the 'duplicates' are dropped ie one of
the  two is kept */
/*drop the append_flag variable*/
drop append_flag

*be careful when dropping duplicates!
use "$raw\bhps_demo.dta",clear
duplicates drop int_month, force

use "$raw\bhps_demo.dta",clear
duplicates drop int_year region , force

/************ Collapse ***********/
use "$raw\bhps_demo.dta",clear
```

```
/*We now have a datafile of cases in england with a region variable merged
in. Say now that we want to create a new file at region level which
contains:
 i)   the total number of households in that region
 ii) total mortgage (total_mortgage)
 iii) average monthly mortgage payment (monthly_mortgage)

NB Be careful - mean is the default and below I state explicitly what
I want each time, even if it is mean*/

collapse (count) hhid (mean) total_mort monthly_mort , by(region)
*collapse can use a variety of statistics, see:
help collapse

save "$work\mortgage collapsed to region.dta",replace

/**** xpose ***/
use "$work\mortgage collapsed to region.dta",clear
xpose, clear varname

/******** reshape *********/
use "$raw\reshape data.dta", clear
/* reshape to long format */
reshape long inc sav, i(id) j(year)
/* revert back to original shape - does not need to be done immediately
after the reshape as Stata remembers within the session */
reshape wide

/* full syntax to reshape to wide format */
reshape wide inc sav, i(id) j(year)

/* if 'j' was a string variable */
tostring year, replace
reshape wide inc sav, i(id) j(year)
*code no longer works - need to tell Stata that j is in string format:
reshape wide inc sav, i(id) j(year) string

*other variables within the dataset:
use "$raw\reshape data.dta", clear
gen num = _n
```

```
reshape long inc sav, i(id) j(year)
/*Other variables are just being repeated when transforming from wide
 into long format */

/*******************************************************************
**  Section 6: Simple regression commands
*********************************************************************/
use "$raw\bhps household file.dta",clear
regress monthly_mortgage rooms

/* Just a demonstration of the regress command - the model may not be
 statistically appropriate */

predict newvar, xb
/*genereates newvar, which takes the value of the linear prediction - i.e.
the value the model would calculate given the explanatory variables in the
model
```

6.3. Variable List

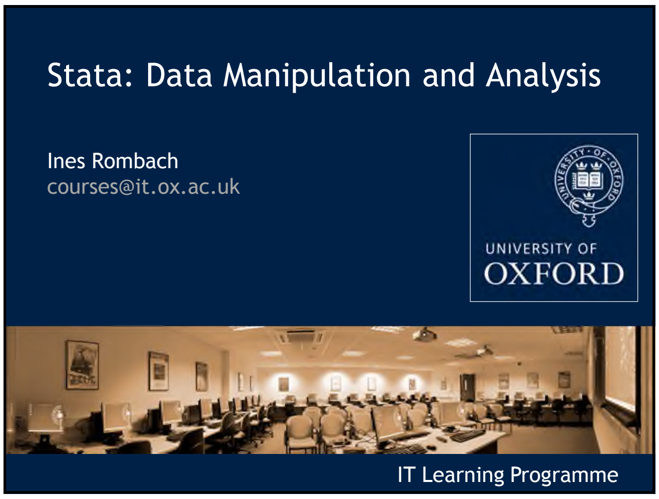
```
              storage   display     value
variable name   type    format      label      variable label
-------------------------------------------------------------------------------
hhid            long    %12.0g                 household identification number
int_day         byte    %8.0g                  day of interview
int_month       byte    %9.0g       month      month of interview
int_year        int     %8.0g                  year of interview
house_type      byte    %20.0g      hh_type    type of accommodation
rooms           byte    %8.0g                  number of bedrooms
tenure          byte    %20.0g      tenure     house owned or rented
hhvalue         long    %12.0g                 value of property: home owners
hhcost          long    %12.0g                 original purchase price of
                                                 property
monthly_mortg~e int     %8.0g                  last total monthly mortgage
                                                 payment
toilet_indoor   byte    %8.0g       loo_in     accom: has indoor toilet
toilet_shared   byte    %8.0g       loo_sh     accom: is indoor toilet shared
garden          byte    %8.0g       gdn        accom: has terrace/garden
total_mortgage  long    %12.0g                 total mortgage on all property
exp_food        int     %8.0g                  total monthly food and grocery
                                                 bill
vehicle_access  byte    %8.0g       veh_acc    car or van available for private
                                                 use
car_own         byte    %22.0g      car_own    household member owns vehicle
car_value       long    %12.0g                 value vehicle(s) less amount
                                                 outstanding
hhsize          byte    %8.0g                  number of persons in household
kids            byte    %8.0g                  number of children in household
pens            byte    %8.0g                  number over pensionable age in
                                                 household
emp             byte    %8.0g                  number in employment in household
wage            byte    %8.0g                  number in household of working
                                                 age
inc_lab         double %10.0g                  annual household labour income
inc_nonlab      double %10.0g                  annual household non-labour
                                                 income
inc_pens        double %10.0g                  annual household pension income
inc_bens        double %10.0g                  annual household benefit income
inc_inv         double %10.0g                  annual household investment
                                                 income
int_place       str6    %9s                    interview location
int_dur         float   %9.0g                  interview duration
age             float   %9.0g
region          byte    %13.0g      region     RECODE of regioncode (region /
                                                 metropolitan area )
hh_weight       double %10.0g       oxhwtuk2   household weight within uk
                                                 estimates
```

6.4. Additional Literature

Baum, C. (2009). “An Introduction to Stata Programming”, Stata Press Publication

Long, S. and Freese, J. (2006). “Regression models for categorical dependent variables using Stata”, Stata Press Publication

Stata Online Channel:
http://www.youtube.com/user/StataCorp/?utm_source=MailingList&utm_medium=email&utm_content=20121010+Training+YouTube

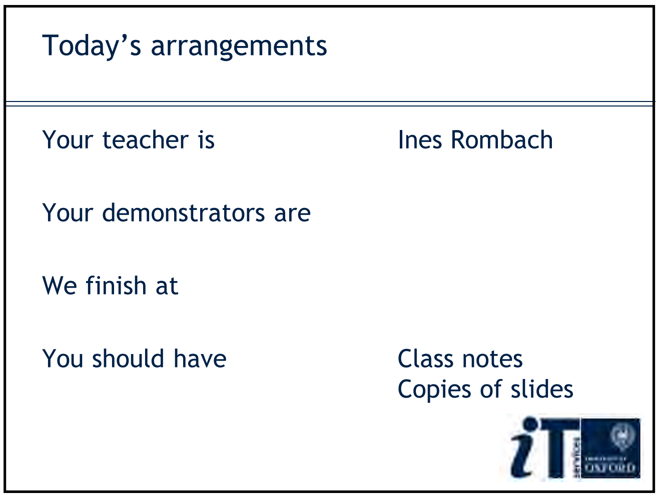


Stata: Data Manipulation and Analysis

Ines Rombach
courses@it.ox.ac.uk

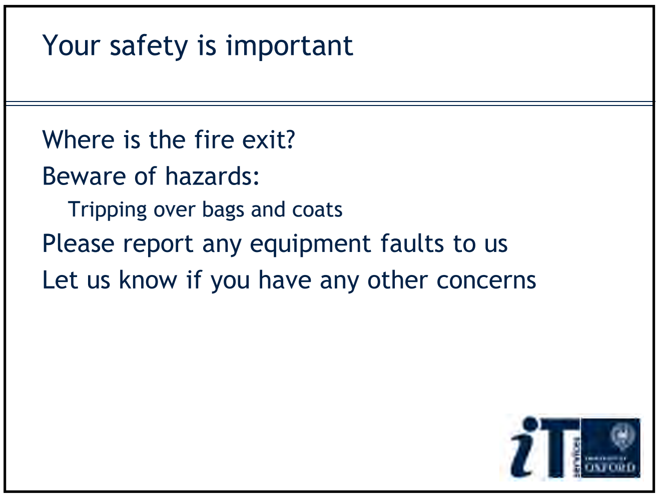
UNIVERSITY OF OXFORD

IT Learning Programme



Today's arrangements

Your teacher is	Ines Rombach
Your demonstrators are	
We finish at	
You should have	Class notes Copies of slides



Your safety is important

Where is the fire exit?

Beware of hazards:

- Tripping over bags and coats

Please report any equipment faults to us

Let us know if you have any other concerns

Your comfort is important

- The toilets are along the corridor outside the lecture rooms
- The rest area is where you registered; it has vending machines and a water cooler
- The seats at the computers are adjustable
- You can adjust the monitors for height, tilt and brightness

Objectives for today's course

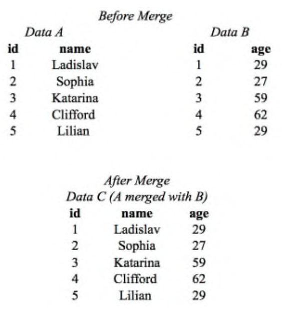
Be able to:

- set up libraries within Stata
- use egen functions
- understand the role of _N and _n
- perform a variety of data manipulations using Stata functions
- use by groups effectively
- recode string into numeric variables and vice versa
- merge, append and reshape data

Review: Conditional Statements

==	!= *or* ~=	>	<	>=	<=	&	|
Equal to (2 = signs)	**Not equal to**	**Greater than**	**Less than**	**Greater than or equal to**	**Less than or equal to**	**And**	**Or**

1 : 1 Merge



Before Merge

Data A

id	name
1	Ladislav
2	Sophia
3	Katarina
4	Clifford
5	Lilian

Data B

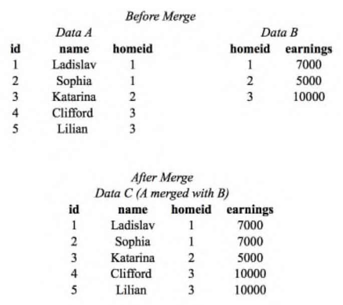
id	age
1	29
2	27
3	59
4	62
5	29

After Merge

Data C (A merged with B)

id	name	age
1	Ladislav	29
2	Sophia	27
3	Katarina	59
4	Clifford	62
5	Lilian	29

M : 1 Merge



Before Merge

Data A

id	name	homeid
1	Ladislav	1
2	Sophia	1
3	Katarina	2
4	Clifford	3
5	Lilian	3

Data B

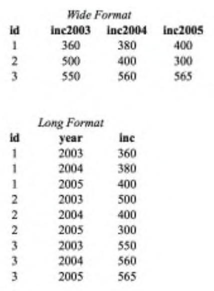
homeid	earnings
1	7000
2	5000
3	10000

After Merge

Data C (A merged with B)

id	name	homeid	earnings
1	Ladislav	1	7000
2	Sophia	1	7000
3	Katarina	2	5000
4	Clifford	3	10000
5	Lilian	3	10000

Reshape



Wide Format

id	inc2003	inc2004	inc2005
1	360	380	400
2	500	400	300
3	550	560	565

Long Format

id	year	inc
1	2003	360
1	2004	380
1	2005	400
2	2003	500
2	2004	400
2	2005	300
3	2003	550
3	2004	560
3	2005	565

courses@it.ox.ac.uk