Contents

1 Essentials of the Stata Language

1.1 Introduction

There are currently over 2000 commands in the stata language. Fortunately, nearly all of them relate to particular kinds of data analysis, and any given individual only needs to know a tiny fraction of them. However, there are a small number of commands and basic concepts that relate to data manipulation and management, which it is essential for all users to be familiar with. This document gives a basic introduction to these essential concepts and commands.

The most important aspect of stata to come to terms with is the fact that it is command-based. There are graphical shortcuts to almost all of the available commands, but you can only use stata really efficiently if you think of a stata session as a series of commands. There are three main reasons why this is important:

1. You must be able to reproduce your analysis. If your analysis consists of a series of commands, they can be stored as a script (called a *do-file* in stata) and rerun any time with a single command. Reproducing an analysis by pointing and clicking is at least as much work as performing the analysis in the first place: if you forget which options you selected and which variables you adjusted for you cannot reproduce your original results.

2. It is easier to produce an analysis that is nearly, but not quite the same. You may want to repeat your analysis after excluding certain subjects, or adjusting for an additional confounder that you forgot about in your original analysis. Again, with a do-file this is very straightforward: using point-and-click software it is not.

3. Finally, using a command line is much quicker *once you are familiar with it.* You can type `reg ht wt` far quicker than you can search through menus to select linear regression of ht on wt, as long as you know that `reg ht wt` is the command you need. However, since most of your analysis will be done using do-files, rather than interactively, the total time saved is not great.

Having said that, there are times when the point-and-click interface can be useful. Most obviously, it can be quicker to use a dialog box to select a file to read in than to type out the full name of the file, particularly if it is nested in a large number of folders.

1.2 Getting Help

1.2.1 Help

Stata has a very comprehensive help system. There is online help for every stata command, which can be accessed by typing `help` *`cmdname`*. This will outline the syntax of the command, the meaning all of the available options, and often there will be a number of examples of its use. Unfortunately, this option is only useful if you know the exact name of the command you are interested in, but are unsure of the precise use or syntax.

1.2.2 Manuals

There is a very comprehensive set of manuals available online. Each entry in the manual begins with a section more-or-less identical to the help-file, but then goes into more detail. References and explanations of the mathematics are normally given, as well as more detailed worked examples. Clicking on the blue command name at the top of the help page will take you directly to the appropriate page in the PDF version of the manual.

1.2.3 Search

The main problem with the help system is that you need to know the exact name of a command in order to use it. Obviously, this is a drawback: how do you find out what command to use if you can't find out about it until you know its name ? The solution is to use `search` *`string`*, which looks through all of the helpfiles on the computer and materials available via stata's `net` command to find *string*, and produce a list of resources that mention it, along with a brief description of the resource.

There are a number of options to search to control where the search takes place. By default, it searches keyword databases on your computer and the internet: you can restrict the search to your local files only with the option [local]. For other possible options, see `help search`. Results are ordered according to likely relevance: first local material, then material from the stata web-site, then user-written material.

1.2.4 Website

There is a stata website at http://www.stata.com. There are literally hundreds of FAQ's online dealing with both common, simple problems and more complicated ways of using stata to perform analyses that it may appear at first sight that stata cannot perform.

1.2.5 Statalist

There is also a mailing list devoted to stata. Instruction for joining (& leaving) are given on the stata website. This is a fairly busy list (around 50 messages a day), consisting mainly of people saying "How do I do ..." with stata", and getting replies that are usually very good. Stata support staff are also involved in the list and will jump in if they think that more detailed technical explanations are required. Old messages are archived, so you may find that searching the archives turns up the answer to your problem.

1.2.6 Stata Journal

The Stata Journal is available online at `"N:\Unit\The Stata Journal"`, and also in the CFMR library. Many of the articles are highly specialised, but there is a series entitled "Speaking Stata" which is aimed at beginners and gives a good introduction to how to use stata efficiently. Older issues of the Journal are also available online.

1.2.7 Me

If all else fails, try me. I'll probably get the answer from one of the above sources (if I get it at all), but I will probably have a good idea of where to look, at least.

1.3 Basic Concepts

1.3.1 Main Stata Windows

On opening stata, you should see 4 windows: the command, results, variables and review windows. The results window has a white background, with black text[a]. The small window at the bottom is the command window, in which you enter commands. The review window contains a list of all of the commands that you have entered in the current stata session, and the variables window contains a list of all the variables in the current dataset.

Command Window

Command Syntax All stata commands follow a common syntax. The full syntax is explained in the User's Guide. A very simplified version, sufficient to start with, is:

`command [varlist] [,options]`

What does that all mean ? Well, the square brackets `[]` surround optional parts, so that the only compulsory part is the actual command name. For example

`describe`

produces a list of the variables in the dataset, and

`exit`

ends the session.

However, most of the time, the command needs the name of one or more variables. For example,

`summarize age`

will produce the mean, standard deviation and some other statistics of the age variable, and

`regress height age gender`

will perform regression to predict height as a function of age and gender.

Most commands also have options, which always given at the end of the command, following a comma. For example,

`summarize age, detail`

provides a more detailed summary than

`summarize age`

If an option needs to be given a value, it is given in backets after the option name. For example, 95% confidence intervals are calculated by default, but for many commands this can be changed using the option `level`. If you want to get 99% confidence intervals in your regression output, you would use the command

`regress varnames, level(99)`

[a] by default: you can change these colours if you wish, but if you have found out how to do that, you certainly know what the results window is.

There are other clauses that can be slotted into this syntax, some of which we will see next week.

You do need to be careful with the syntax of a command, since simple errors like a misplaced space or a missing comma will give an error message. However, it does not take long to get used to it, and once you understand the logic of the syntax it is quite difficult to get it wrong.

You must remember that the language is case-sensitive: `height` and `HEIGHT` are two different variables. You could have both of these variables in the same dataset without confusing stata, but you will confuse yourself if you do. So I recommend that all your variable names contain only lower case letters.

Many commands and options have abbreviations, but I do not recommend using them until you are very familiar with the commands. It only takes a few seconds to type in the additional letters, and when you look at your work again in six months time, it will make it far easier to remember what you did. Also, you need to know the full name of a command in order to get help on that command. For this reason, I do not use any abbreviations in this document[b].

Repeating previous commands The command that has just been entered can be brought back to the command window using the `PageUp` key. Pressing this key repeatedly will bring back all of the previous commands. If you go too far back with the `PageUp` key, you can use the `PageDown` key to get more recent commands.

Variable name completion If you enter the beginning of a variable name and then press the `tab` key, Stata will complete the variable name as much as possible. If there are several possible completions, it will complete as far as possible and wait for further input. For example, if your data contains variables `height0` and `height1`, and you enter `hei` then press the `tab` key, the variable name will be completed to `height` and wait for you to type either 0 or 1. If you are used to filename completion in the bash shell, this works in the same way.

Variables Window

This contains a list of variables in the data set, together with their labels, although the labels may not be visible if the window is not sufficiently wide. Clicking on a variable name copies that name to the command window.

Review Window

This contains a list of commands that have been entered previously. Clicking the left mouse button on a command will copy it to the command window, where it can be edited and re-run if required. Double-clicking on a command in this window runs it directly. The entire list of commands in the review window can be saved as a do file by clicking on the upper left corner of the window: this gives a menu, one option being `"Save Review Contents"`.

Results Window

The results of any commands you type will appear in the results window. There is a scroll bar to return to previous output. However, the results window only retains a limited amount of your analysis, so you need to start a logfile (explained below) to ensure that you do not lose any output.

[b] well, I try not to, but `gen` may slip instead of `generate` and `tab` instead of `tabulate`

The results window is interactive, to the extent that any blue text represents a hyperlink: clicking on the blue text will generally open a viewer window with more information.

Normally, data is presented in the results window one screenful at a time. When paging through data in this way, pressing the `Return` advances by one line, the `Space` key advances by one page, and pressing `q` stops the output. The button containing a red circle with a white cross can also be used to stop the output.

It is possible to turn this paging off with the command

```
set more off
```

This is very useful in a do-file: it will then produce all of the output you need without you needing to press space all of the time. To turn paging back on again, type

```
set more on
```

1.3.2 Do-Files

A do-file is simply a list of stata commands. Giving the command

```
do do-file.do
```

causes stata to run all of the commands in the do-file. This is one of the greatest strengths of stata, the ability to perform exactly the same actions repeatedly, and it is vital to get to grips with this concept as soon as possible.

For example, if you wish to create a new variable, you should always use a do-file to do so, rather than just entering a command and saving the resulting dataset. This has two advantages:

1. You can see exactly how the variable was calculated by looking at the do file. If two people try to calculate the same variable and get different answers, it is possible to check exactly what each did and why they differ.

2. If you need to add subjects to the dataset later, or calculate the same variable in a different dataset, it can be done very simply and quickly.

It is also a good idea to keep a do-file containing your analyses. You can be certain that any analysis that you do will need to be performed many times, with very slight changes, before your write-up of the data-analysis is published. It is far easier to edit a do-file to make these minor changes than try to remember exactly how you performed an analysis originally and then do it again slightly differently.

You can also collect together a group of files needed for a particular analysis in a "Project". These files can be do files or datasets, and can make it easier to find all of the files that you need if they are spread across multiple directories. For example, most analyses involve some do-file used by everyone working on a given project, and some do-files specific to that analysis, which would be stored in different places. Type "help Project Manager" for more information.

Profile.do

Every time Stata is run, it searches for a file called `profile.do` in certain places, including `C:\ado\personal\profile.do`, which is where I would recommending creating your own (for details, type `help profilew` into a Stata command window). This file contains commands which you wish to have run every time that you start Stata. Possible uses of `profile.do` is to

define your own entries in the Stata menu system, or set up logging of commands as you enter them.

1.3.3 Log Files

Stata does not log your results by default. There is a "Stata Results" window which contains the results of each command, but this is of a limited size. It may not even contain the complete output of a single command if it is sufficiently complicated. To preserve your results for posterity, you need to open a log file. This can be done using

```
log using filename
```

All of your output will be stored in *filename* until you close the log with

```
log close
```

Stata can keep logs in two formats: SMCL or text. SMCL is a text markup language that is only understood by stata, so such logs must be viewed in stata. Text logs can be viewed in any text editor. The default format is SMCL: if you want the log to be in text format, the command to use is

```
log using filename, text
```

You should always open a log file as one of the first things that happens in a do file. The syntax to use is

```
capture log close
log using myfile.log, options
```

The reason for the `"log close"` command is that it is impossible to open a new log-file if one is already open. However, if a log file were not open, then the command `"log close"` would generate an error, and the do-file would halt. The command `"capture"` means "do not stop, even if there is an error", so that the `"log using"` command will be run regardless.

1.3.4 Interaction with Operating System

Stata can only find files if either:

1. it is in the current working directory
2. you give the full path name to the file

Full names can be long and tedious to type in (not to mention error-prone), so it is useful to be able to change the current working directory. There are a number of commands to help with this.

`pwd` displays the name of the current working directory

`cd "dirname"` changes the current working directory to *dirname*. The inverted commas are optional, unless there is a space in *dirname*, but are a good habit to get into.

`mkdir "dirname"` creates a new directory called *dirname*.

`dir` lists all of the files in the current working directory

`shell command_name` runs the command *`command_name`* in a command prompt window

Windows uses the symbol "\" to separate directories on a path, whilst Unix uses "/". Stata on windows can understand either, so either can be used in giving file and directory names. However, "\" will cause problems if it is followed by a macro (see below), and we now have a Linux workstation that requires Unix filenames, so it is a good idea to get into that habit.

1.3.5 Macros

A macro is a way of using a short name to represent a longer piece of text. For example, if you store your data in `N:\projects\A_Major_project\My_subproject\Data`, it can be a nuisance having to type that directory name in whenever you wish to read in a dataset. To make life easier, you can define a macro by typing

```
global mydir N:\projects\A_Major_project\My_subproject\Data
```

This creates a global macro called `mydir` containint the text `N:\projects\A_Major_project\My_subprojec`
Then, whenever you type `$mydir` in stata, it will be replaced by the text
`N:\projects\A_Major_project\My_subproject\Data`. So, if you type

```
use $mydir/mydata
```

stata will read in the dataset `N:\projects\A_Major_project\My_subproject\Data\mydata.dta`

A major advantage of macros is the ease with which do-files can be made portable. All directories that are used in a file should be referenced using macros. Then, when the do-file and data-files are moved to a different directory (as they will be when they are archived, if not before), it is only necessary to change the macro definitions and all of the references in the do-file will be changed.

Global vs Local Macros

The macros we have seen so far (beginning with "$") are *global* macros, meaning that once they are set, they keep their value until stata finishes. There are also *local* macros, that only retain their value until the end of the do-file in which they are defined. Local macros are also used by the commands `foreach` and `forvalues`, which we will see next week. Local macros are defined using the command `local`, and used by putting them in single inverted commas: ‘ and ’ (the opening inverted comma is found at the top left of a standard UK keyboard, the closing one at the right of the middle row). So the "mydir" example above, rewritten to use local macros, would be

```
local mydir N:\projects\A_Major_project\My_subproject\Data
use "‘mydir’\mydata"
```

1.3.6 Variable lists

Often you need to present a list of variables to stata. Rather than listing the name of every variable individually, there are a number of shortcuts that can be used. For example, if you wish to summarize every variable which begins with the letters "age", you would use the command

```
summarize age*
```

You can also give a list of variables which are consecutive in the dataset as `firstvar-lastvar`. For detailed information about specifying variable lists (or *varlists*), type

```
help varlist
```

1.3.7 Number Lists

There is also a shorthand for entering lists of numbers. For fuller details, type

```
help numlist
```

.

Symbol	Meaning	Example	Expansion
	list of numbers	1 2 3	1 2 3
x/y	whole numbers from x to y inclusive	1/5	1 2 3 4 5
x y to n	numbers from x to n, increasing by $y-x$	5 10 to 20	5 10 15 20
x y : n	same as x y to n	5 10:20	5 10 15 20
$x(y)n$	numbers from x to n, increasing by y	10(10)50	10 20 30 40 50
$x[y]n$	same as $x(y)n$	10[10]50	10 20 30 40 50

Table 1.1: Number Lists

1.4 Manipulating Variables

1.4.1 Creating and modifying variables

Generate and Replace

The simplest command to create a new variable is `generate`. For example, if the date of birth is stored in the variable `date_of_birth` and the date the questionnaire was filled in is stored in `date_of_quest`, then the subject's age at the time the questionnaire was completed can be calculated as

```
generate age = (date_of_quest - date_of_birth) / 365.25
```

The above command would not have worked if a variable called `age` already existed in the dataset. In this case, it would have been necessary to either drop the variable `age` before generating the new variable:

```
drop age
generate age = (date_of_quest - date_of_birth) / 365.25
```

or to use the `replace` command:

```
replace age = (date_of_quest - date_of_birth) / 365.25
```

Either of the above will end up with exactly the same dataset.

By default, `generate` creates variables of type float (i.e. they contain numerical values, and can contain a decimal point). If you wish to generate a variable of a different type, you need to explicitly state that in your command. The available types are listed below:

For example, the command

type	size (bytes)	min	max	precision	missing values
byte	1	-127	126	whole numbers	.
int	2	-32,767	32,766	whole numbers	.
long	4	-2,147,483,647	2,147,483,646	whole numbers	.
float	4	-10^{36}	10^{36}	7 digits	.
double	8	-10^{308}	10^{308}	15 digits	.
str$n$	n				""
strL	varies				""

Table 1.2: Available data types

```
gen str6 name = "MyName"
```

creates a variable called `name`, containing "MyName" in every observation. Variables of the type `str`***`n`*** can contain up to ***`n`*** characters, where n has a maximum value of 2,045. Variables of type `strL` can contain up to 2,000,000,000 characters.

Missing Values

Missing values in string variables are always represented by `""`, but numerical variables can take a number of different missing values. The default missing value is simply a dot, but you can also use ".a" ".b" ... ".z" if you wish to distinguish between different types of missing data: that subject was not asked the question, the question was not answered, the answer was illegible etc. However, all missing values are larger than the largest number that stata can represent with that data type, so you can always exclude *all* missing values with

```
if variable < .
```

Egen

A more powerful way of generating new variables is with `egen` (short for *Extended GENerate*. The syntax is very similar to that of `generate`:

```
egen newvar = fcn(varlist)
```

However, there are a large number of functions which can be used for *fcn(varlist)* which can perform calculations that `generate` cannot. For example, there are functions to calculate sums, means, medians, variances, z-scores, etc. Type `help egen` for a full list of available functions.

1.4.2 Labelling variables

Variable names in Stata are limited to 32 characters. This enables reasonably descriptive names to be used, but it may be that a fuller description of a variable is required. For example, it is often useful to know the exact wording used for a question. In this case, the command

```
label variable varname "label"
```

can be used. For example, if you wish to assign the label "How many days in the week do you drink alcohol ?" to the variable `alcohol`, the command used would be

```
label variable alcohol "How many days in the week do you drink alcohol ?"
```

Equally important is the ability to label the values of a variable. Usually, categorical variables are stored as numbers, since this requires less storage and is more efficient. However, it is important to know which values these numbers refer to. This can be done using value labels.

Assigning labels to values is done in two stages. First the labels are assigned to numbers. This is done with a `label define` command, such as:

```
label define yesno 1 "Yes" 0 "No"
```

which assigns the value "Yes" to the number 1 and the value "No" to the value 0.

The second step is to assign the label to a variable, with the `label values` command. This command takes the form

```
label values varname labelname
```

For example, to use the label `yesno` defined above with the variable `back_pain`, the command to use would be

```
label values back_pain yesno
```

Only one variable can be given in the `label values` command, yet there may be several variables which have the same label applied to them. We will see an efficient way of doing this next week.

1.4.3 Selecting variables

Often there are more variables in your dataset than you are interested in, and you may wish to use only a subset of the available variables. There are two commands to facilitate this: `drop` and `keep`. They work as you might expect: `drop` *`varlist`* removes all of the variables in *`varlist`* from the dataset, and `keep` *`varlist`* removes all of the variables that are not in *`varlist`* from the dataset.

1.4.4 Formatting variables

The command `format` can be used to change how stata presents data to you. It is most commonly used for dates. Dates are stored as the number of days since January 1, 1960. So January 2, 1960 would be stored as 1, and February 10, 2005 as 16477. This has the advantage that you can do arithmetic with dates, to calculate the time between two dates etc. However, it means that if you list a series of dates, you will see a list of numbers, and converting from the number to the date is not trivial.

But you can use the `format` command to do this for you. The command

```
format %dD/N/CY date
```

means that the variable `date` will be presented in the form Day/Month/Year, rather than as numbers. Alternatively,

```
format %dCY-N-D date
```

will present the dates as Year-Month-Day instead. For a full list of all of the characters that can be used in a date format, along with their meanings, type

```
help dfmt
```

The format command can also be used to determine the formatting of other numbers: how many decimal places to use etc. Full details are given by

```
help format
```

1.5 Manipulating Datasets

1.5.1 Filenames

Filenames that contain spaces can cause problems in stata, unless the filename is put in inverted commas. For this reason, I strongly suggest that whenever you use a filename, you put the name of the file in inverted commas, even if it does not contain a space: you may move it to a different directory later and wonder why your do-file no longer works. This is especially important when using a macro as part of the filename, since the macro may expand to something containing a space.

1.5.2 Reading and Saving data

The most important commands for manipulating datasets are `use`, which reads a stata dataset into stata, and `save`, which saves a stata dataset. These commands have a very simple syntax, although there are a few safeguards built in to them to stop you destroying your own data by mistake.

Use

The simplest way to read a file into stata is to issue the command

```
use "filename"
```

This will read the file *filename* into stata. However, if you have data already in stata, reading new data in will replace it and it may be lost forever. For this reason, stata will not read in the new dataset if there is data in memory that has been changed since it was read from the disk. If you want to keep the data you are currently working on, you need to enter

```
save "old_filename"
use "filename"
```

If you don't mind losing the existing data, you can type either

```
clear
use "filename"
```

or

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

either of which have the same effect.

Typing in the full name of a file can be tedious, especially when there are may levels of directories to go through. When working interactively, this is one of the commands which are much easier to use from the menus or button-bar. Once you have opened the dataset once, however, the command can be stored in a do-file for subsequent use. If you need to reopen the dataset (if you have made an error in manipulating the dataset, for example), the command can

be rerun from the review window. Be warned, however, that using the menu always implies the option `clear`, so the safeguards built into `use` do not work.

Save

A stata dataset is saved using the command

`save "filename"`

However, this might not be what you really want: if a file of that name already exists, it will be replaced with the new file. If you want to have copies of both files available, you must use different names for them. If you want to replace the original file, you can with the command

`save "filename", replace`

Saveold

It may be that you need to save a dataset in an old format: you may be collaborating with someone who only has access to an old version of stata, or you may want to use StatTransfer, which does not recognise the last stata file format. This can be done using the command

`saveold filename`

You may specify `,replace` if you wish, but it would be a bad idea to replace a file in a more recent format with one in an older format: you may be losing information.

1.5.3 Combining Datasets

Sometimes you may want to combine data from 2 or more data sets. There are two possible situations:

1. The datasets contain the same information about two different groups of people
2. The datasets contain different information about the same group of people.

In the first situation (e.g. you have received datasets from two different centres, each recording the same information about different populations, and you want to combine the datasets), the command to use is `append`. In the second case (e.g. data from x-rays are stored in one file, DNA data in another, and you want to look at genetic risk factors for x-ray outcomes), the command is `merge`.

append

The command `append` adds new observations to the end of an existing dataset. It is essential that corresponding variables in the two datasets have exactly the same names, otherwise they will be treated as different variables. If variables have the same meaning, but different names, the command `rename` can be used to change the name in one of the files.

A simple example of appending data is shown below:

The variables `ID, common_1 & common_2` exist in both files, whilst `file1_1 & file1_2` exist only in file1 and `file2_1 & file2_2` exist only in file2. Thus there will be missing data for these variables in the combined file.

ID	common_1	common_2	file1_1	file1_2
1	a_1	b_1	c_1	d_1
2	a_2	b_2	c_2	d_2
3	a_3	b_3	c_3	d_3

Table 1.3: Appending Data: File 1

ID	common_1	common_2	file2_1	file2_2
4	a_4	b_4	e_4	f_4
5	a_5	b_5	e_5	f_5
6	a_6	b_6	e_6	f_6

Table 1.4: Appending Data: File 2

It will almost always be useful to create a variable which will tell you from which file a given subject was taken. A simple way of doing this is shown below:

```
use "filename1"
gen fromfile = 1
append using "filename2"
replace fromfile = 2 if fromfile == .
```

Here, the file *filename1* is read into memory. Then a new variable, `fromfile` is created, which takes the value 1 for all observations in *filename1*. Next, addition observations are added to the dataset from *filename2*. Finally, `fromfile` is given the value 2 for any observations for which `fromfile` is currently missing. Since `fromfile` == 1 for all observations in *filename1*, this means that `fromfile` is set to 2 for all observations in *filename2*.

merge

Merging data is used more commonly than appending data. In order to merge two datasets, you need to specify which record(s) in one dataset correspond to which record(s) in the other. Therefore, before merging two files, you must ensure that there is a variable or group of variables that are identical in each dataset.

For example, suppose that you wish to merge the files `file1` and `file2`, and that the variable `idno` exists in both datasets and is a unique identifier for each subject. Then before merging, the datasets would look like this:

In order to merge these two files, you need to make sure that they are both sorted by `idno`:

```
use file2
sort idno
save, replace
use file1
sort idno
```

Then you can give the command to merge the files:

```
merge 1:1 idno using file2
```

ID	common_1	common_2	file1_1	file1_2	file2_1	file2_2
1	a_1	b_1	c_1	d_1	.	.
2	a_2	b_2	c_2	d_2	.	.
3	a_3	b_3	c_3	d_3	.	.
4	a_4	b_4	.	.	e_4	f_4
5	a_5	b_5	.	.	e_5	f_5
6	a_6	b_6	.	.	e_6	f_6

Table 1.5: Appending Data: Combined Files

idno	var1	var2
1	a_1	b_1
2	a_2	b_2
3	a_3	b_3

Table 1.6: Merging Data: File 1

Note the syntax of the command:

```
merge [1:1]|[1:m]|[m:1]|[1:m] variable_name using filename
```

After merging, the dataset would look like this:
Things to note:

- All subjects who appear in either of the files will appear in the merged file.
- Subjects who do not appear in one of the files will have missing values for those variables.
- Stata has created a new variable called `_merge`, which contains the values

  1 if the value of idno exists in file1, but not in file2

  2 if the value of idno exists in file2, but not in file1

  3 if the value of idno exists in both files

The most common error in merging data is that one or other datasets is not sorted. In this case, you will get the error message

```
master data not sorted
```

or

```
using data not sorted
```

If it is the master data not sorted, you can simply enter the command

```
sort matching_variable
```

and try again. If it is the using data that is not sorted, save the current master file in its sorted state, then `use` and `sort` the second dataset.

The above example would be slightly more complex if there was no unique identifier for each subject. Imagine that there are a number of different centres participating in the study, and

idno	var3	var4
1	c_1	d_1
3	c_3	d_3
4	c_4	d_4

Table 1.7: Merging Data: File 2

idno	var1	var2	var3	var4	_merge
1	a_1	b_1	c_1	d_1	3
2	a_2	b_2	.	.	1
3	a_3	b_3	c_3	d_3	3
4	.	.	c_4	d_4	2

Table 1.8: Merging Data: Combined Files

each numbered their subjects 1–n. You cannot merge be `idno`, since the same `idno` can exist in each centre, and correspond to different subjects. For this reason, you can use a number of variables to match on: in this case, the command to use is

```
merge 1:1 centre idno using file2
```

Of course, you would need to run the command

```
sort centre idno
```

on both files first.

Ensuring Uniqueness Very often, when merging, you will want to match a single record in one file to a single record in the other. If this is what you want, the command `merge 1:1` will verify that this is true, and produce and error if not. However, there are times that you want to match several observations in one file to a single observation in the other: maybe one file contains data on individuals, and the other contains data on the households to which they belong. Multiple household members would need to be matched to the same household. There are commands `merge 1:m` for the case where a single record from the data in memory should be merged to several records in the "using" file, and `merge m:1` if several records from the data in memory should be matched to a single record in the "using" file.

However, the `merge` command does not require this. If a subject appears twice in one of the files, there will be two records for that subject in the final dataset. This can be avoided by using the option `unique` to the `merge` command:

```
merge idno using file2, unique
```

This will produce an error message if either file contains more than one entry with the same `idno`.

1.6 Other Dataset Manipulation Commands

1.6.1 `browse` and `edit`

The command `browse` opens a data editor window, in which the dataset is presented in spreadsheet form. The data can be examined, but not changed, following a `browse` command. If you wish to examine only a subset of the data, you can list the variables you want to see after the `browse` command.

The command `edit` is similar to `browse` but does allow changes. **Do not use it**. Any changes you make to your data must be documented, so the best way to manage it is with a do-file.

1.6.2 `preserve` and `restore`

You may wish to change your data temporarily. For example, there is a command `collapse` which creates a new dataset, consisting of the means (or other statistics) of the variables in the current dataset, calculated for a number of subgroups. You may want to do some analysis of these means, then return to your original dataset. The command `preserve` saves a copy of your dataset to disk, so that you can get it back easily after analysing the `collapse`d dataset, and `restore` restores the previously saved dataset.

2 Practical on Essentials of Stata

2.1 Preliminaries

To start Stata, click on Start Button ⇒ All Programs ⇒ Stata 14 ⇒ StataIC 14 (64 bit)

This should work for most of you, but depending on your faculty and whether you are staff or a student, the exact route may vary. Let me know if you have any difficulty.

Solutions for all practicals can be found at
`http://personalpages.manchester.ac.uk/staff/mark.lunt/stats_course.html`

Choose a directory to hold the work you are going to do on this course. I suggest that `P:` is the best place, maybe `P:\statacourse`. Set the global macro to the name of your chosen directory:

```
global mydir my chosen directory
```

(Remember text in italics is not typed in as it is, but replaced with the name of the directory you are using). This way, you can all chose different directories, but if I give my instructions in terms of the macro `$mydir`, they will work for all of you.

Of course, the directory you are going to use must exist, before you can save any files in it. You can use

```
mkdir "$mydir"
```

provided that the new directory is only one level below an already existing directory. To make sure that it exists, type

```
cd "$mydir"
```

to change to it, and make sure you don't get an error message in response.

It is a good idea to start a log now.

2.2 Reading and Saving Files

Type the command

```
sysuse auto
```

This will search for a dataset called `auto` that is installed with stata. You now want to save this dataset to your own directory. You can type

```
save "$mydir/auto"
```

to save this dataset in your own working directory, or if you have already changed to `$mydir`, you can just type

```
save auto
```

Make sure that you are in you own working directory (use `pwd` to find out, and `cd` to change if necessary). Now type

```
dir
```

to ensure that you really do have a copy of the dataset saved in this directory. If you have, type

```
clear
```

to remove the dataset from memory, then

```
use auto
```

to read the version in your own directory back in.

Now save the dataset using a different name, say `myauto`. This is because we are going to modify the dataset, and you should always make sure that your original data cannot be lost by mistake: save the original and work on a copy.

```
save myauto
```

2.3 Creating and Modifying Variables

2.3.1 Using `generate`

We are now going to create a new variable, `wtkg`, to contain the weight of each vehicle in kg. Since 1kg is approximately 2.2046lbs, the command to do this is

```
generate wtkg = weight/2.2046
```

It is good practice to label every variable as soon as you have created it, before you have time to forget what it is, so let's do that:

```
label variable wtkg "Weight (kg)"
```

2.3.2 Creating indicator variables

Very often, you want to generate a variable that can only take two values, representing "true" and "false". As an example, we will create a variable `short`, which takes the value 1 for all cars less than 190 in long and 0 for longer cars. The conceptually simplest way to do this is

```
generate short = 0
replace short = 1 if length < 190
```

However, there is a more efficient way: in stata, any logical expression (such as `(length < 190)`) has a value 0 if the expression is not true and 1 if the expression is true. So we could identify short cars with the single command

```
generate short2 = (length < 190)
```

If you type `tab short short2`, you should see that both commands have had exactly the same effect.

2.3.3 Using *egen*

Suppose that you wish to divide the vehicles into tertiles. There are a number of ways to do it: the simplest is to use `egen`. The function `cut` can be used to categorize a continuous variable, and you can either choose the thresholds yourself, or divide the observations into a given number of (more-or-less) equal sized groups. The code for doing this is

```
egen wtt = cut(weight), group(3)
```

As always, we now need to label this variable

```
label variable wtt "Tertiles of weight"
```

If you type `tab wtt`, you will see that `wtt` takes three values, 0, 1 or 2. The lowest tertile contains 24 cars, the others 25.

So that you don't get confused as to which tertile is which, it is best to label them. First, you need to define a label for them:

```
label def tertiles 0 "Lowest tertile" 1 "Middle Tertile" 2 "Highest Tertile"
```

Then you can assign that label to the variable

```
label values wtt tertiles
```

If you now repeat the command

```
tab wtt
```

you should see more meaningful labels for the categories of `wtt`.

2.3.4 Creating a string variable

If you type

```
tab make
```

you will see that of the 74 cars in the dataset, several are made by the same manufacturer: 3 Toyotas, 4 VW's etc. However, there is no variable to identify the manufacturer, so we had better create one. We will do this by taking the first word of the variable `make`, i.e. all of the characters up to the first space. This requires two functions: `strpos(`*`string1, string2`*`)`, which gives the position within *`string1`* at which *`string2`* first occurs, and `substr(`*`string, num1, num2`*`)`, which returns the substring of *`string`* that starts at position *`num1`* and ends at position *`num2`*. So, to extract the manufacturer of each vehicle, we use the command

```
gen str20 company = substr(make, 1, strpos(make, " "))
```

if you now type

```
tab company
```

you should see the following table

```
    company |      Freq.     Percent        Cum.
------------+-----------------------------------
        AMC |          3        4.11        4.11
       Audi |          2        2.74        6.85
        BMW |          1        1.37        8.22
      Buick |          7        9.59       17.81
       Cad. |          3        4.11       21.92
      Chev. |          6        8.22       30.14
     Datsun |          4        5.48       35.62
      Dodge |          4        5.48       41.10
       Fiat |          1        1.37       42.47
       Ford |          2        2.74       45.21
      Honda |          2        2.74       47.95
      Linc. |          3        4.11       52.05
      Mazda |          1        1.37       53.42
      Merc. |          6        8.22       61.64
       Olds |          7        9.59       71.23
    Peugeot |          1        1.37       72.60
      Plym. |          5        6.85       79.45
      Pont. |          6        8.22       87.67
    Renault |          1        1.37       89.04
     Toyota |          3        4.11       93.15
         VW |          4        5.48       98.63
      Volvo |          1        1.37      100.00
------------+-----------------------------------
      Total |         73      100.00
```

Notice that we have lost a car: there are only 73 values of `company`. This is because the the Subaru has no model name, and hence there is no space in `make` for this car. There are a number of solutions for this: the simplest is

```
replace company = make if company == ""
```

You now need to save this dataset, as we may use it again later:

```
save, replace
```

2.4 Manipulating Datasets

In order practice combining datasets, we need to create two datasets to combine. We will use a simulated dataset of blood pressure measurements called `bplong`. This file contains two records of blood pressure for each subject: one made before some intervention, the second taken after the intervention. The variable `when` takes the value 1 for the "before" measurement and 2 for the "after" measurement. This dataset will be split into separate "before" and "after" datasets.

This is done with the following commands:

```
sysuse bplong
save "$mydir/bplong"
preserve
keep if when == 1
save "$mydir/bpbefore"
restore
```

```
keep if when == 2
save "$mydir/bpafter"
```

Now, all of the records for the first visit are in `bpbefore` and all those for the second visit in `bpafter`.

2.4.1 Append

First of all, we will see how to append the two datasets, to reproduce the structure of `bplong`. The code to do this is:

```
use bpbefore, clear
gen fromfile = 1
append using bpafter
replace fromfile = 2 if fromfile == .
```

Now you can check that `fromfile` agrees with `when` by typing

```
tab fromfile when
```

If you wish to save this file, make sure that you give a suitable label to `fromfile` first. You may also wish to label the values that `fromfile` takes.

2.4.2 Merge

Now we are going to merge the files, so that we end up with a single record for each subject, and two separate variables for the "before" and "after" measurements. Since we want two variables in the merged dataset, we will need to change the name of the variable `bp` in at least one of the files. In fact, it is easier to change the variable name in *both* files.

```
use bpbefore, clear
rename bp bp_before
save, replace
use bpafter
rename bp bp_after
save, replace
```

Note that you need the `replace` option to save, since the file already exists, but we want to change it. Now, you need to ensure that both files are sorted by `patient`, so that they can be merged:

```
use bpbefore
sort patient
save, replace
use bpafter
sort patient
save, replace
```

Yes, it would have been more efficient to do the sorting and renaming at the same time, rather than have to read the files in twice. You can do that in future. Of course, if you remember

to use `PageUp` and `PageDown` to get older commands, or took your commands from the review window, there was very little typing to do anyway.

Now, we can do the actual merge:

```
merge 1:1 patient using bpbefore
```

If you have followed the previous instructions, `bpafter` is still loaded in stata, so you just need to merge in `bpbefore`. You can make sure that the merge was successful by typing in `tab _merge`: you should see that _merge takes the value 3 for all 120 patients, showing that they had data in both files.

2.5 Further Exercises

1. Calculate the lengths of each car in metres, using the fact that 1 inch is 0.0254 metres.
2. Create a variable `heavy`, which takes the value 0 for cars weighing less than 3000 lbs and 1 for cars weighing more than 3000 lbs.
3. Create tertiles of `wtkg` from the auto dataset, and check that it produces exactly the same results as creating tertiles of `weight`.
4. A simpler way of creating the `company` variable would be to use `egen` with the `ends` function. Use `help egen` to find out how this function works, then create a new variable called `comp2`, containing the first word of `make`.
5. Using the dataset that you created in Section 2.4.2, calculate the change in blood pressure between the "before" and "after" visit.
6. Using the same dataset, create a variable that takes 6 different values, depending on which age group and sex the patient belongs to. (*Hint: the* `group` *option to* `egen` *will help: check it out*) Label the values so that you can tell which group is which.