Introduction to Statistical Computing: Stata

September-October

2022

Module taught by Kathy Baisley, Nick Magill & Manuela Quaresma
Module materials largely developed by Tim Collier

LONDON SCHOOL *of* HYGIENE &TROPICAL MEDICINE

Chapter 0: Introduction to Module

0.1 Overview of Module

This is a brief introduction to and overview of the Stata component of the Introduction to Statistical Computing module.

The purpose of these sessions is to introduce you to the statistical package Stata. We will be using Stata version 17, which was released in April 2021. We will assume no prior knowledge of Stata or indeed of any other statistical software. The goal is enable you to use Stata efficiently and to be able to produce datasets that are ready for analysis. This will involve becoming familiar with the Stata windows or working environment, understanding Stata's command syntax, being able to create and manage datasets and to identify and correct errors. We will look at commands for describing and summarising data but more advanced statistical methods will be introduced in other modules. If we have time, we will go through some of the material in the Stata Graphics manual during the last session.

Although no prior experience of using Stata is assumed, anyone who is already familiar with Stata will be able to move ahead more quickly to the more advanced material. However, there is nearly always benefit to be gained from going over the introductory sessions carefully.

Throughout the sessions the key principles of good practice for data management will be emphasised along with the importance of understanding the work-flow of data analysis and of developing a consistent and clear system or convention for data and file management.

Although there is not a formal assessment for this module, many of the other module assignments will require you to use Stata.

0.2 Style

The style of the Stata sessions will be different from most of the other modules on the MSc. Rather than pre-recorded one hour lectures and live practicals, we will be recording short (5-10 mins) screencasts introducing you to the Stata commands to be used in each session. You will be asked to view these in advance of each practical session. We will then have a 1.5 hour practical session where you will be asked to work your own way through the module notes and the accompanying exercises. Tutors will be on hand to answer any questions that you may have. We find that this style works best for this type of module. The best way to learn a new statistical software package is to work your way through the notes whilst sat in front of computer trying out the commands. This style also enables people with different levels of experience of statistical software packages to work along at their own pace.

0.3 Module Files

The data and other files for these sessions can be downloaded from the *U:/* drive. Copy the "*Med_St_Intro_to_Stata*" folder plus its contents from *U:/Download/Teach/* to your own computer. You might want to rename the top folder as Intro_to_Stata.

0.4 Additional Materials

You are encouraged to expand your learning outside these module materials during the course of your study. This module is intended only as a brief introduction to Stata's capabilities; it will get you started but you will almost certainly run into questions on topics that are not covered here, or you will have some difficulty using Stata with your data. Fortunately you will also almost certainly find that someone has encountered, and publicly queried, the same or similar issues before.

Online:

- The ***Stata PDF Documentation*** should be your first port of call. It is a series of e-books containing over 14,000 pages of documentation on Stata's functionality that you can access from within Stata. Click Help -> PDF Documentation from the toolbar and choose the appropriate manual from the hyperlinked bookmarks on the resulting PDF. The manual contains many helpful remarks, including statistical overviews, and examples of how to use commands and interpret their results. You can also find it online here: https://www.stata.com/features/documentation/

- ***Statalist***, the official Stata forum run and moderated by Stata users. Questions and their answers date back to 1994, with many useful tips and suggestions on all types of data, statistical and Stata topics. https://www.statalist.org/

- The ***Stata YouTube Channel*** has been active since 2011 and now contains hundreds of short how-to clips for various data management and analysis tasks. https://www.youtube.com/user/statacorp

- Stata collates and maintains a series of online ***third-party resources*** that are also extremely helpful. Check out UCLA's Institute for Digital Research and Education, in particular. https://www.stata.com/links/resources-for-learning-stata/

Paper:

- ***The Workflow of Data Analysis Using Stata*** (2009); J. Scott Long, Stata Press. "Aimed at anyone who analyses data, this book presents an effective strategy for designing and doing data-analytic projects." Several copies are stocked in the library.

- For those who want to extend beyond introductory topics: ***An Introduction to Stata Programming, Second Edition*** (2016). Christopher Baum, Stata Press. "Great for anyone who wants to learn Stata programming."

Chapter 1: An Introduction to Stata

Aims & Objectives of Chapter 1

By the end of this chapter you should:

- be aware of Stata's capabilities
- be able to load a Stata dataset
- be familiar with the Stata Windows interface
- know how to change the working directory

The following commands are used in this chapter: `use, cd, browse, dir`.

1.1 Overview of Capabilities

Stata is an integrated statistical package designed for research professionals. As well as a wide range of *statistical tools*, Stata also provides publication-quality *graphical capabilities* and powerful *data management* features.

Basic *STATISTICS* tools include summaries, cross-tabulations, correlations, t-tests, chi-square tests and tests of equality of variances plus much more. Tools for the analysis of epidemiological data include epidemiological tables, linear and logistic regression and survival time methods (including life-table, Kaplan-Meier and Cox regression) and much more. See http://www.stata.com/features/ and the *Statistics* drop down menu.

Stata's *GRAPH* capabilities allow the production of publication-quality distinctly styled graphics. The two-way family of graphs are particularly useful. Stata comes with its own built in Graph Editor with which it is possible to interactively edit Stata graphs. See https://www.stata.com/features/publication-quality-graphics/ and the *Graphics* drop down menu.

Stata has an extensive set of *DATA MANAGEMENT* commands which can deal with both string and numeric variables. Datasets can be split, combined, reshaped and collapsed. Variables can be transformed, recoded, categorised and combined. There are many commands for dealing with date and time variables. Advanced tools are provided for managing and analysing survival time, time-series, categorical, survey and longitudinal data. See https://www.stata.com/features/data-management/ and the *Data* drop down menu.

Stata is user-friendly; it has a comprehensive on-line help facility and it is possible to access nearly all of its commands through a Graphical User Interface (GUI). Stata comes in different *'flavours'* (e.g. SE and IC) which have slightly different specifications regarding the maximum number of variables. With *Stata/SE* a maximum of 32,767 variables can be stored in a single dataset. With *Intercooled Stata* (*Stata/IC*) the maximum number of variables is 2,048. With both flavours the number of observations is limited only by the size of the memory of the computer being used. The current largest computer can hold 2.14 billion observations.

1.2 Loading a Stata Dataset

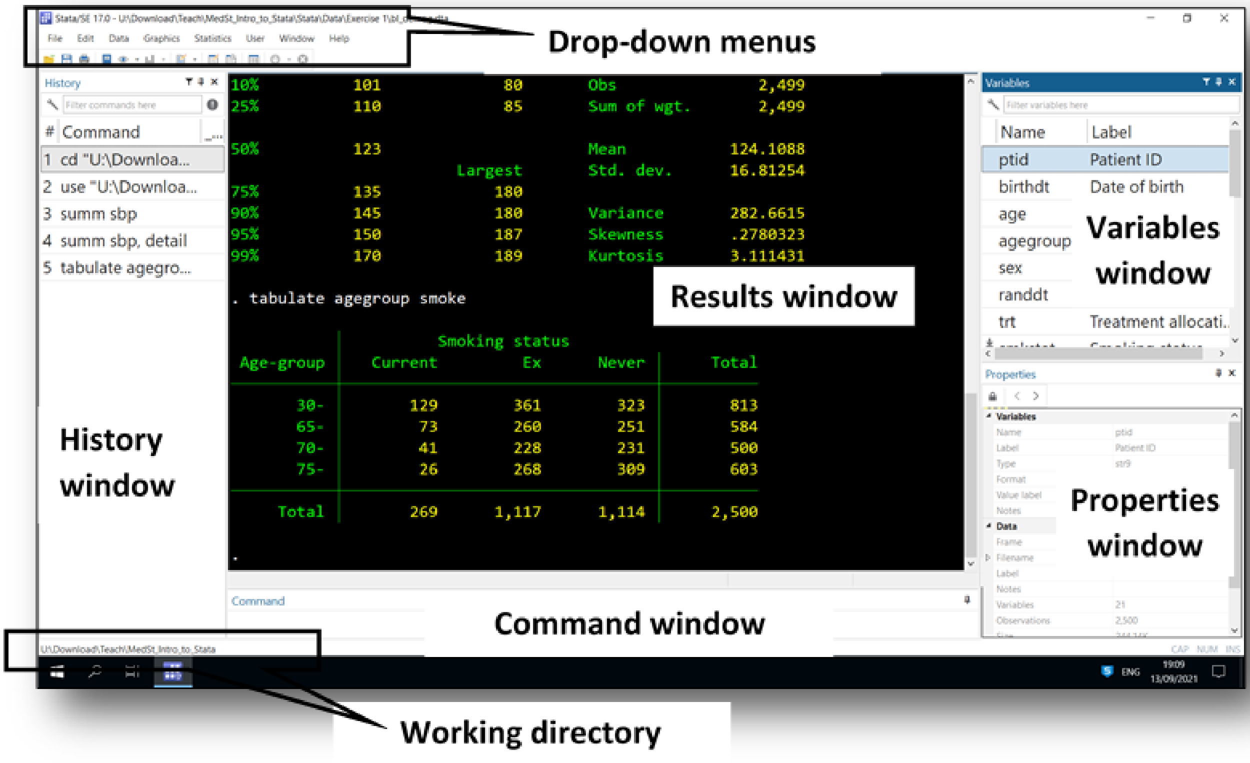
A Stata dataset can be loaded either by (i) double-clicking on a Stata data file, which automatically launches Stata and loads the dataset, or (ii) by launching Stata through the Start menu or an applications window and then opening the data file from within Stata.

Stata data files have the extension *.dta*. These are data files that have been created within Stata and have been saved in Stata's own format. Stata data files cannot be opened using any other statistical software. When you double-click on a Stata data file Stata is launched and the data file is loaded into Stata's memory.

Selecting Stata through the Start menu, or via a shortcut icon, will cause Stata to be launched, but without loading any data. The simplest way to load a Stata dataset from within Stata is to use the *File > Open* menu. We will load the *bl_demog.dta* dataset.

1.3 The Stata Windows Interface

On opening Stata you will be faced with 5 windows, a series of drop down menus and a shortcut tool bar (Figure 1.1). The 5 windows are the (i) Variables Window, (ii) Command Window, (iii) History Window, (iv) Results Window and (v) Properties Window. These, along with the drop-down menus and toolbar, are described below.



```
10%          101             80       Obs               2,499
25%          110             85       Sum of wgt.       2,499

50%          123                      Mean           124.1088
                        Largest       Std. dev.      16.81254
75%          135            180
90%          145            180       Variance       282.6615
95%          150            187       Skewness       .2780323
99%          170            189       Kurtosis       3.111431

. tabulate agegroup smoke

           |       Smoking status
 Age-group |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
       65- |        73        260        251 |       584
       70- |        41        228        231 |       500
       75- |        26        268        309 |       603
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500

.
```

#	Command
1	cd "U:\Downloa...
2	use "U:\Downloa...
3	summ sbp
4	summ sbp, detail
5	tabulate agegro...

Name	Label
ptid	Patient ID
birthdt	Date of birth
age	
agegroup	
sex	
randdt	
trt	Treatment allocati..

Figure 1.1: The default widescreen Stata windows layout

In Stata 17 the default windows layout is the *widescreen* layout as shown above. You can switch between layouts using the *Edit > Preferences > Load Preference Set >* menu.

(i) Variables Window

When a dataset is loaded in Stata's memory the *Variables Window* displays the name (e.g. smoke) of each variable and its label (e.g. "Smoking status") if a label has been created.

The Variables Window is interactive – variables can be selected into the Command Window (or Graphical User Interface) saving the need to type the variable name. A variable can be selected either by clicking the arrow that appears when the mouse hovers over its name or by double-clicking on the variable name.

A number of options, including choice of font and copying a variable list, can be accessed by right-clicking within the Variables Window. The width of each column can be varied using the cursor by clicking and dragging the vertical line which appears between each column on the bar at the top of the Variables Window.

By default the variables appear in the order in which they are stored in the dataset; the Variables Window can also be sorted alphabetically by variable name or variable label by clicking on the column headers. This does not change the order of the variables in the dataset, but can be useful when trying to locate a variable in a large dataset.

(ii) Command Window

Stata can be used interactively through the *Command Window*. Commands can be typed into the Command Window using the keyboard and submitted using the return (enter) key.

It is possible to scroll back/forwards through any commands that have already been submitted using the page up/down keys. These commands can then be submitted again or can be edited within the Command Window and then resubmitted. Previously submitted commands can similarly be selected from the History Window using the mouse left click.

Variables can be selected into the Command Window from the Variables Window using the mouse. Options such as font size can be changed by right-clicking within the Command Window.

(iii) History Window

The *History Window* keeps a record of all commands that have been submitted through either the Command Window or Graphical User Interface (GUI). (Note this window is called the *Review Window* in Stata 15 and earlier versions). The History Window is interactive so that selecting one of the commands by left clicking on it will cause it to appear in the Command Window. The command can then be edited within the Command Window and resubmitted. Commands that have resulted in an error message are displayed in red in the History Window.

Note that all contents of the History Window are lost when exiting Stata. However, it is possible to send or save all or some of the contents of the History Window to a do-file (more later) or to copy them to the clipboard. This can be done by right-clicking in the History Window and selecting the appropriate options. Commands can also be deleted from the History Window.

There are three column headers; # - the sequence number of the command as submitted in a particular session; *Command* – the command itself; *_rc* – contains any error codes. The History Window can be sorted by any of these columns (by clicking the header); this can be useful, for example when wanting to delete all commands containing errors. Once the erroneous commands have been deleted the History Window can then be resorted sequentially by clicking on the # header.

(iv) Results Window

Any command entered either in the command window or via the GUI appear in the *Results Window* along with any output (including error messages) arising from the command. The size of the font and general preferences can be changed by right-clicking within the Results Window.

The Results Window has a limited buffer size. Once the limit has been reached the earliest results from the current session disappear. The size of the Results Window buffer can be increased via the *Edit* drop down menu *Edit > Preferences > General Preferences* and then select the Results tab, or using the *set scrollbufsize* command. The buffer size can be changed to lie between 10,000 and 2,000,000 bytes. The default is 200,000.

All the contents of the Results Window are lost when exiting Stata. However, it is possible to keep a record of everything that appears in the Results Window in what Stata calls a log-file (more about this later). The status bar along the bottom of the results window indicates whether a log-file is open.

You can also copy-and-paste tables and other text directly from the Results Window into other software such as Microsoft Word or Excel (more later). Using the *File > Print* menu you can print either the whole, or a selection, of the Results Window.

Note that Stata will normally present one full screen of results at a time in the Results Window. If the command you submit requires more than one page of results Stata will pause after the first page and display ***–**more**–*** at the bottom of the Results Window. You can view the next line by pressing the *Enter* key, or scroll to the next page of results by pressing the *Space bar* or by the clicking on ***–**more**–*** . To stop the command press *q* or the *Break* button at the right hand end of the shortcut toolbar.

(v) Properties Window

The Properties Window shows properties for individual variables (e.g. name, label, type, value label) and for the dataset currently in memory (e.g. name of dataset, pathway, number of variables/observations). By default the properties for the first variable in the dataset are shown; properties for other variables are shown when they are selected in the Variables Window or you can use the arrow in the Properties Window to scroll through the variables.

By default the Properties Window is locked so that you cannot edit variable names etc. You can unlock the window by clicking on the small padlock symbol 🔒 at the top left of the Properties Window. It is then possible to rename variables or add labels interactively.

The Drop Down Menus

Above the main windows interface is the menu bar with eight drop-down menus: File, Edit, Data, Graphics, Statistics, User, Window and Help (Figure 1.2).

Through the *File* menu it is possible among other things to load or save a Stata dataset, open a Stata graph or log-file, execute a do-file, import data saved in non-Stata format (e.g. Excel, delimited text files created by spreadsheets, etc), change the working directory and print the contents of the results, viewer or graph windows.

The *Edit* menu can be used to copy and paste and to change various preferences e.g. general window and graph preferences.

Using the *Data*, *Graphics* and *Statistics* menus it is possible to access almost all of Stata's commands through the Graphical User Interface (GUI) or dialogue boxes. We will return to these menus and the GUI later.

Through the *Window* menu it is possible to reopen a window (e.g. the review window) if it is closed down during a session. It can also be used to open the data editor, the variables manager, a new or existing do-file or a new viewer.

The *User* menu is for use by programmers, providing a place for additional menu items.

Through the *Help* menu it is possible to search for and view help files for Stata commands, open the PDF documentation that comes with Stata, visit useful pages of the Stata Corp website (e.g. user support and frequently asked questions pages) and check for official updates.

The Shortcut Toolbar

12 shortcut buttons appear on the toolbar.

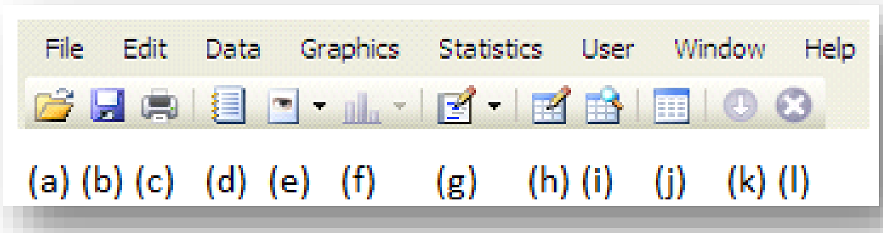


Figure 1.2: Stata drop-down menus and toolbar

From left to right these are: (a) open a Stata dataset, (b) save the current dataset (in Stata format), (c) print (either graph or contents of the viewer), (d) log begin/suspend/close, (e) open/bring to front the Stata viewer, (f) bring to front graph window, (g) open/bring to front do-file editor, (h) open/bring to front data editor, (i) open/bring to front data browser, (j) open/bring to front variables manager, (k) continue following pause, and (l) break, which stops the current action.

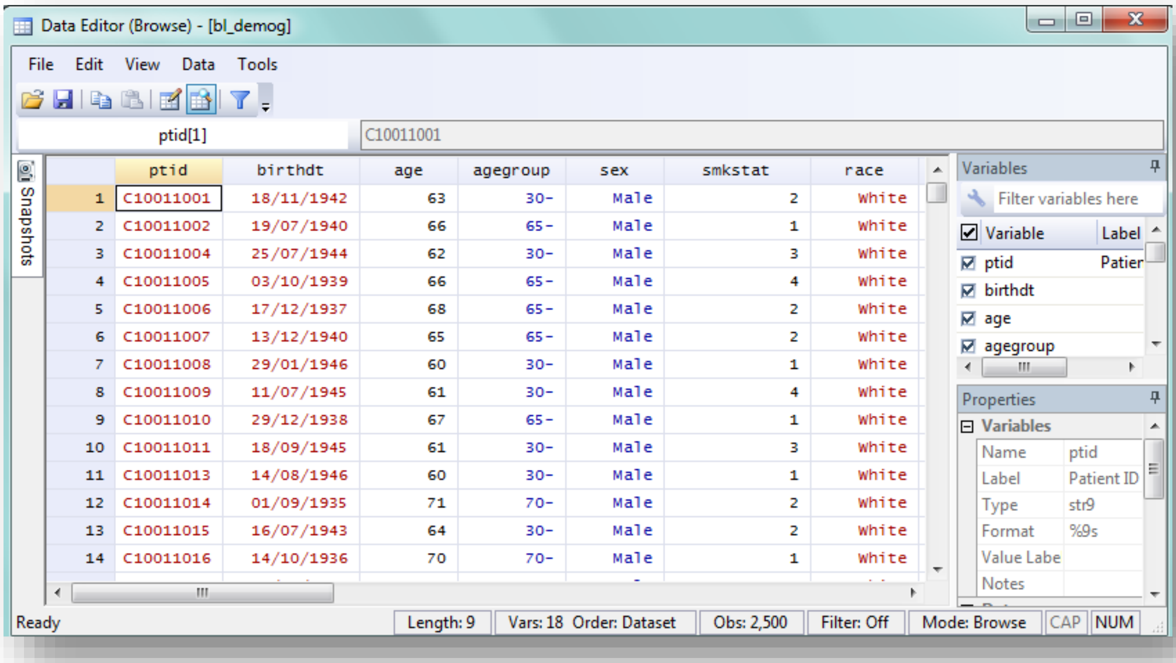
The Data Editor and Browser Windows

The data currently in memory can be viewed in either the *Data Editor* (in which the data can be viewed and edited) or in the *Data Browser* (in which the data can be viewed but not edited). The data editor and browser can be opened either via the *Data* drop down menu (*Data > Data Editor*) or through the short cut buttons (Figure 1.2).

The data are presented in a rectangular spreadsheet in which observations are stored in rows and variables in columns (Figure 1.3). Variable names appear at the top of each column.

Stata has a colour system to differentiate between variables of different formats:

> *Red = string variable*
> *Black = numeric variable*
> *Blue = numeric variable with value labels* attached



	ptid	birthdt	age	agegroup	sex	smkstat	race
1	C10011001	18/11/1942	63	30-	Male	2	White
2	C10011002	19/07/1940	66	65-	Male	1	White
3	C10011004	25/07/1944	62	30-	Male	3	White
4	C10011005	03/10/1939	66	65-	Male	4	White
5	C10011006	17/12/1937	68	65-	Male	2	White
6	C10011007	13/12/1940	65	65-	Male	2	White
7	C10011008	29/01/1946	60	30-	Male	1	White
8	C10011009	11/07/1945	61	30-	Male	4	White
9	C10011010	29/12/1938	67	65-	Male	1	White
10	C10011011	18/09/1945	61	30-	Male	3	White
11	C10011013	14/08/1946	60	30-	Male	1	White
12	C10011014	01/09/1935	71	70-	Male	2	White
13	C10011015	16/07/1943	64	30-	Male	2	White
14	C10011016	14/10/1936	70	70-	Male	1	White

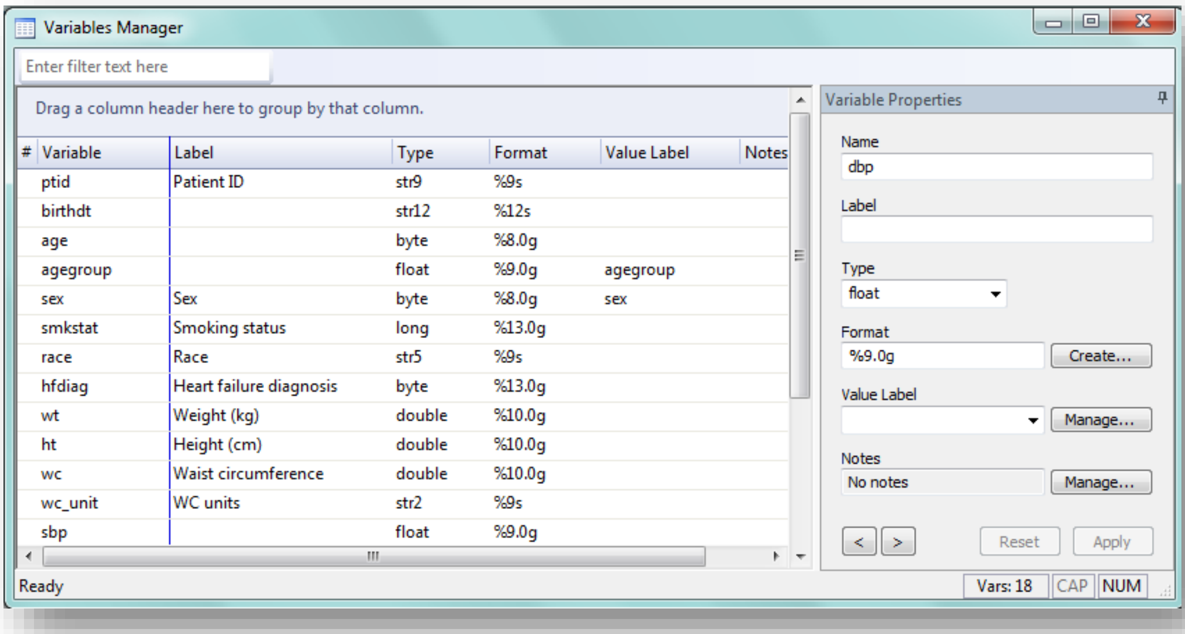
Figure 1.3: Stata's Data Browser Window

The data browser has its own set of menus and shortcut buttons which can be used to filter observations, sort variables, manage different variable properties and take a snapshot of the data currently in memory, which can be restored later if required.

The Data Editor also has a *Variables* and *Properties* window which displays the names and labels (if any exist) of the variables in memory.

The Variables Manager Window

The *Variables Manager* window can be opened using the *Window* menu or by using the Variables Manager button on the shortcut toolbar i.e. (j) in Figure 1.2.



# Variable	Label	Type	Format	Value Label	Notes
ptid	Patient ID	str9	%9s		
birthdt		str12	%12s		
age		byte	%8.0g		
agegroup		float	%9.0g	agegroup	
sex	Sex	byte	%8.0g	sex	
smkstat	Smoking status	long	%13.0g		
race	Race	str5	%9s		
hfdiag	Heart failure diagnosis	byte	%13.0g		
wt	Weight (kg)	double	%10.0g		
ht	Height (cm)	double	%10.0g		
wc	Waist circumference	double	%10.0g		
wc_unit	WC units	str2	%9s		
sbp		float	%9.0g		

Name	dbp
Label	
Type	float
Format	%9.0g
Value Label	
Notes	No notes

Figure 1.4 The Variables Manager Window

The Variables Manager is an interactive tool for managing the properties of the variables in the dataset in memory. It can be used to rename variables, create variable and value labels, change display formats and more.

When executing a command through the Variables Manager the command is echoed to the Results and History windows.

The list of variables in the Variables Manager Window can be sorted in ascending order by clicking on one of the column headers (a second click sorts in descending order). Clicking on the hash (#) symbol found at the far left of the column header bar will revert to the original ordering.

1.4 The Current Working Directory

It is important to note the *working directory* highlighted in Figure 1.1 above. This is where Stata will look to find files and is where data files, log files and graphs will be saved unless otherwise specified.

Depending on where (e.g. personal laptop, university network) you are using Stata there will be a default working directory e.g. H:/my documents as in Figure 1.1.

If you launch Stata by clicking on a Stata data file the current working directory will be the folder where the data file is located.

The current working directory can be changed either using the *File > Change Working Directory* menu, or on the command line using the `cd` (change directory) command e.g.

```
. cd "C:/Intro to Stata/data"
```

If one of the files or folders in the pathway or directory contains an embedded space or some special characters it is necessary to enclose the whole pathway in double quotes. This happens automatically when using the *File > Change Working Directory* menu.

To see what files are in your current working directory type `dir`.

```
. dir
  <dir>   9/25/17 12:01  .
  <dir>   9/25/17 12:01  ..
 251.1k   9/14/15 16:09  bl_demog.dta
1696.1k   9/27/14 12:03  vitals_long.dta
```

1.5 Data Dictionary for the Baseline Data

Dataset name: bl_demog.dta
Description: Stata dataset containing patient id, baseline demographic information plus some lifestyle, vital signs, anthropometric and laboratory measurements.

Variable name	*Variable description*	*Coding etc.*
ptid	Unique patient identifier	String length 9
birthdt	Date of birth	Stata date format
age	Age (years)	Numeric
agegroup	Age categories	0=30-64, 1=65-69, 2=70-74, 3=75+
sex	Sex	String: Female, Male
smkstat	Smoking status (5 levels)	1=Never, 2=Ex-Light, 3=Ex-Heavy, 4=Current-Light, 5=Current-Heavy
smoke	Smoking status (3 levels)	String: Current, Ex, Never
race	Ethnic group	String: Asian, Black, White, Other
hfdiag	Heart failure diagnosis	1=Ischemic, 2=Non-ischemic
wt	Weight (kg)	Numeric
ht	Height (cm)	Numeric
wc	Waist circumference	Numeric
wc_unit	Waist circumference	String: CM, M
sbp	Systolic Blood Pressure (mmHg)	Numeric
dbp	Diastolic Blood Pressure (mmHg)	Numeric
hrate	Heart Rate (bpm)	Numeric
egfr	Estimated Glomerular Filtration Rate (ml/min/1.73m)	Numeric
lvef	Left ventricular ejection fraction (%)	Numeric
diab	Diabetes	0=No, 1=Yes

Chapter 2: Stata Commands and Results

Aims & Objectives of Chapter 2

By the end of this chapter you should:

- know how to submit commands using the *Graphical User Interface*
- understand the general form of Stata's command syntax
- be able to enter commands in the command window
- be able to use Stata's help and search facilities
- know how to create and use a do-file
- understand good practice for do-files
- be able to save results in a log file
- know how to open and view a saved log file

The following commands are used in this chapter: `tabulate`, `summarize`, `list`, `sort`, `bysort`, `help`, `search`, `net search`, `log using` and `log close`

2.1 The Graphical User Interface (GUI)

Although throughout this session we will concentrate on learning the command syntax, a brief description of the GUI may be useful. The drop down menus give an idea of the breadth of Stata's statistical, data management and graphical capabilities and can help in learning, or remembering, the syntax for a particular command.

Almost all Stata commands can be accessed in a point-and-click fashion via the three drop-down menus: *Data*, *Graphics* and *Statistics*. Figure 2.1 shows the options available under the *Statistics* menu.

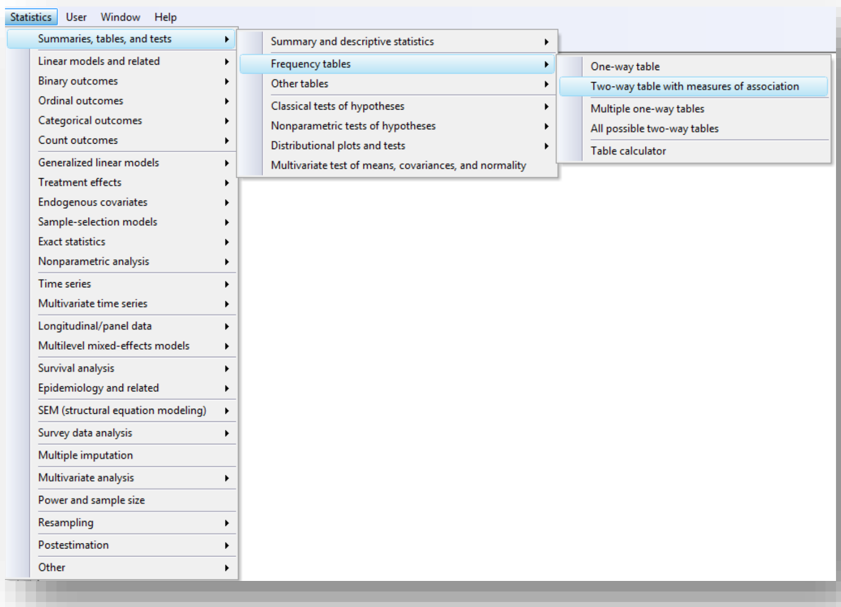


Figure 2.1: The Statistics menu and submenus

Selecting an option via the *Statistics* menu causes a *dialog box* for that command to open.

GUI Example 1: A Two-way Table

First we need to launch Stata and load the dataset (bl_demog.dta) from the Exercise 2 folder using the *File > Open* menu. From the *Statistics* menu select *Summaries, tables and tests > Frequency Tables > Two-way table with measures of association.* This should cause the *tabulate2* dialog box to open (Figure 2.2).

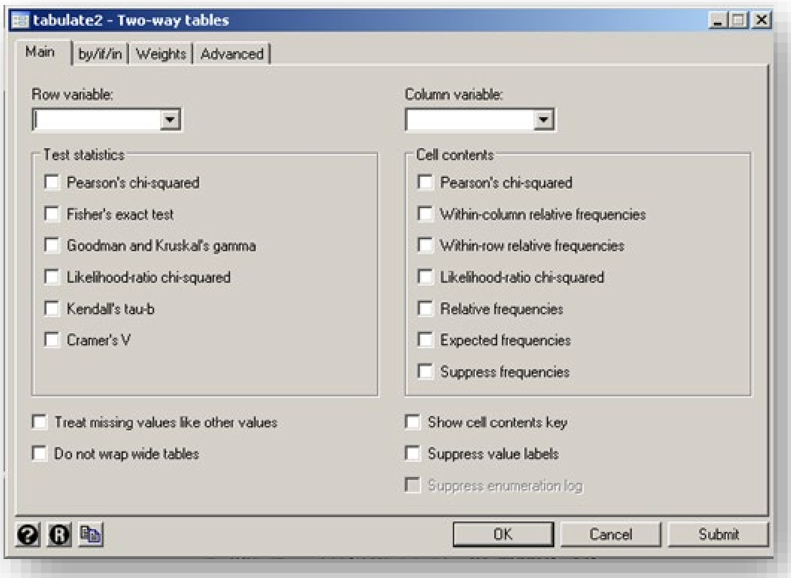


Figure 2.2: The tabulate2 dialog box

The *tabulate2* dialog box has 4 tabs; *Main*, *by/if/in*, *Weights* and *Advanced*. These tabs are used to select the variables to be tabulated and specify different options.

There are also six buttons on the toolbar at the bottom of the dialog box. These are:

?	Open help file for command
R	Reset the dialog box
⧉	Copy the command as it currently stands i.e. with variables and options as selected
OK	Executes the command and closes the dialog box
Cancel	Closes the dialog box without executing the command
Submit	Executes the command and leaves the dialog box open

The *Main* tab contains the *Variable* fields (Row and Column variable) and a number of 'tick box' options. The variable names can be either typed directly into the row and column fields or selected using the drop-down menu at the right of each field.

Select the variable *agegroup* as the row variable field and the variable *smoke* as the column variable field using the drop-down menus. Click *submit*.

Stata executes the command leaving the dialog box open. The command syntax and the resulting table appear in the Results Window. The command syntax also appears in the History Window. As we did not tick any options we get the default two-way table which contains just the cell and marginal totals along with the variable names (or labels if attached) and the row/column headings.

```
. tabulate agegroup smoke

 Age-group |          Smoking status
   (years) |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
       65- |        73        260        251 |       584
       70- |        41        228        231 |       500
       75- |        26        268        309 |       603
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500
```

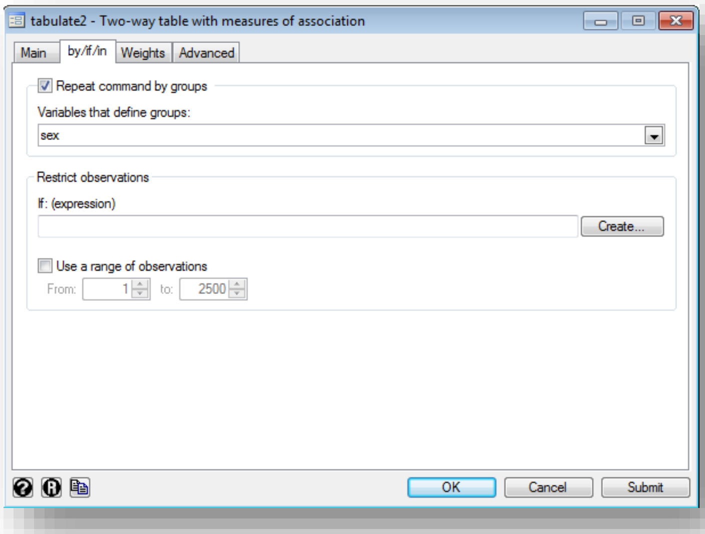
To obtain row percentages select *Within-row relative frequencies* option from the list of Cell contents options on the *Main Tab*. Click the *Submit* button. The results from this are shown below. Make sure you understand the output.

```
tabulate agegroup smoke  , row

+----------------+
| Key            |
|----------------|
|   frequency    |
| row percentage |
+----------------+

 Age-group |          Smoking status
   (years) |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
           |     15.87      44.40      39.73 |    100.00
-----------+---------------------------------+----------
       65- |        73        260        251 |       584
           |     12.50      44.52      42.98 |    100.00
-----------+---------------------------------+----------
       70- |        41        228        231 |       500
           |      8.20      45.60      46.20 |    100.00
-----------+---------------------------------+----------
       75- |        26        268        309 |       603
           |      4.31      44.44      51.24 |    100.00
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500
           |     10.76      44.68      44.56 |    100.00
```

GUI example 2: Repeating a command

The *by/if/in* tab allows a command to be repeated over each level of another variable (or combination of variables) or to be limited to a subgroup of the data. For example, we may wish to repeat the two-way tabulation of age-group and smoking status by sex, i.e. for males and females separately.

Select the by/if/in tab, and click the *Repeat command by groups* option and select the variable sex into the *Variables that define groups* box (Figure 2.3).


tabulate2 - Two-way table with measures of association

Main | by/if/in | Weights | Advanced

[✓] Repeat command by groups
Variables that define groups: sex

Restrict observations
If: (expression)

[ ] Use a range of observations
From: 1 to: 2500

Figure 2.3: Repeating a command over groups

The resulting output is shown below.

```
. by sex, sort : tabulate agegroup smoke

-> sex = Male
           |          Smoking status
 Age group |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       106        322        219 |       647
       65- |        61        234        155 |       450
       70- |        38        201        153 |       392
       75- |        18        241        194 |       453
-----------+---------------------------------+----------
     Total |       223        998        721 |     1,942

-> sex = Female
           |          Smoking status
 Age group |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |        23         39        104 |       166
       65- |        12         26         96 |       134
       70- |         3         27         78 |       108
       75- |         8         27        115 |       150
-----------+---------------------------------+----------
     Total |        46        119        393 |       558
```

Note that the command is now prefixed with `by sex, sort:`.

GUI example 3: Restricting the observations with if

We may wish to restrict the operation of the command to a subset of the observations in our dataset e.g. to males, those with BMI>30, etc. We can do this via the *by/if/in* tab.

Continuing with the tabulate2 dialog box, in the *by/if/in* tab deselect the *Repeat command by groups* option.

On the same tab we use *the If (expression)* box. We will restrict the tabulation of *agegroup* and *smoke* to just the males (sex = 1) in the study.

In the *if(expression)* box we type *sex*==1. Note the use of the double equals (==) which is Stata's convention when testing if something is equal to something else.

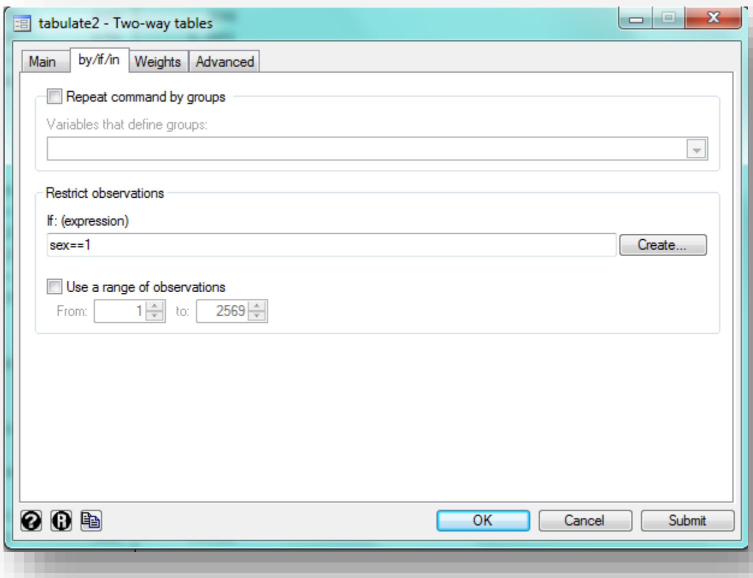


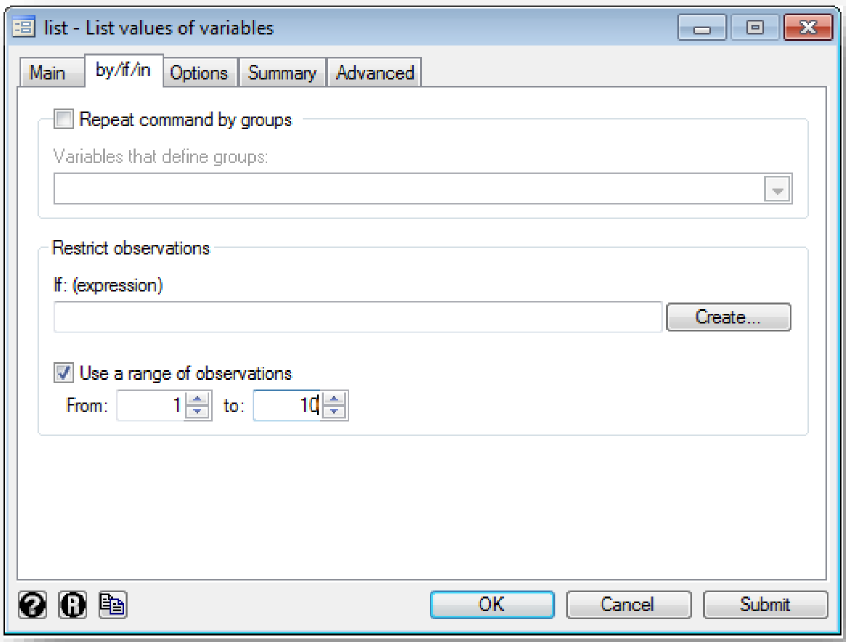
Figure 2.4: Restricting observations for a command

```
. tabulate agegroup smoke if sex==1

           |           Smoking status
 Age group |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       106        322        219 |       647
       65- |        61        234        155 |       450
       70- |        38        201        153 |       392
       75- |        18        241        194 |       453
-----------+---------------------------------+----------
     Total |       223        998        721 |     1,942
```

The resulting output is restricted to the 1,942 males in the study.

GUI example 4: Restricting the observations with in

We may wish to restrict the operation of the command to certain rows in our dataset e.g. to the first 10 rows. We will demonstrate this by using the `list` command. From the drop-down menus select *Data > Describe Data > List Data*. This will open up the `list` dialog box. On the main tab enter ptid, birthdt, age and smoke in the variables box.



Repeat command by groups: unchecked
Variables that define groups:
Restrict observations
If: (expression)
Use a range of observations: checked
From: 1 to: 10

Figure 2.5: Restricting observations using in

```
.
list ptid birthdt age smoke in 1/10

     +----------------------------------------+
     |      ptid     birthdt   age      smoke |
     |----------------------------------------|
  1. | C10011001   18nov1942    63    Current |
  2. | C10011002   19jul1940    66      Never |
  3. | C10011004   25jul1944    62    Current |
  4. | C10011005   03oct1939    66    Current |
  5. | C10011006   17dec1937    68    Current |
     |----------------------------------------|
  6. | C10011007   13dec1940    65    Current |
  7. | C10011008   29jan1946    60      Never |
  8. | C10011009   11jul1945    61    Current |
  9. | C10011010   29dec1938    67      Never |
 10. | C10011011   18sep1945    61    Current |
     +----------------------------------------+
```

We now have listed the values for *ptid*, *birthdt*, *age* and *smoke* in the first ten rows of the datatset – as it is currently sorted. We'll see later how to list the last 10 observations in a dataset.

2.2 The command syntax

One of the strengths of Stata is the consistency of its command syntax. Almost all of Stata's commands follow the same general form which makes it much easier to learn to use Stata. The basic form of the command syntax is:

```
[prefix:] command [varlist] [if] [in] [using] [ , options]
```

Square brackets indicate parts of the syntax which *may* be optional.

Command name

Each Stata command is invoked using the command name e.g. `summarize`, `regress`, `tabulate`, `histogram`. The command name appears first in the command syntax unless it is being modified by a prefix command.

From the general form of the command syntax above we can see that this is the only part of the command syntax not in square brackets i.e. not optional. Some commands (e.g. `summarize`, `describe`, `codebook`) do not require a variable list – when no variable list is specified Stata interprets this as meaning all variables in the dataset. For example;

```
summarize

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        ptid |         0
         dob |      2500   -7653.395    2867.442     -17807       1890
         age |      2500     68.6632    7.675033         43         95
         sex |      2500      1.2232    .4164747          1          2

[some output omitted]

       creat |      2500    211.9553    1042.164     15.912       9999
          hb |      2500    425.1898    1984.981        8.8       9999
         pot |      2500    108.2695     1014.15        2.2       9999
      sodium |      2500     483.181    1807.237      106.9       9999
-------------+--------------------------------------------------------
      totbil |      2500    636.6483    2415.778          0       9999
```

Using the GUI we created a two-way table. The syntax began with the command name which was `tabulate`:

```
 tabulate agegroup smoke

 Age-group |          Smoking status
   (years) |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
       65- |        73        260        251 |       584
       70- |        41        228        231 |       500
       75- |        26        268        309 |       603
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500
```

Command names can be abbreviated. The minimum abbreviation accepted is underlined in the help file for each command (see help with Stata commands later). For example, the command `summarize` can be abbreviated to `summ`.

Stata is case-sensitive with command names - all of Stata's commands are lower case. If you use the wrong case or misspell a command then Stata will return an error message (in red) reporting that the command has not been recognized.

```
sumarize sbp dbp
unrecognized command:  sumarize
r(199);
```

Such an error message tells you that the mistake is right at the start of the syntax.

Variable list

Following `command` in the command syntax is `[varlist]` the variable list. This is a list of one or more variable names that identify which variables that are to be used in the command. The number and order of the variable names may or may not be important.

As we saw above, for some commands, e.g. `summarize` the variable list is optional; Stata interprets no variables to mean all the variables in the dataset. By adding a variable name or names we restrict the operation of the command to those variables specified. For example, to obtain summary statistics for the variables *sbp* and *dbp*,

```
. summ sbp dbp

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sbp |      2499    124.1012    16.81271         72        189
         dbp |      2499    74.58944    10.20323         36        117
```

Here the order of the variable list is only important in that it affects the order in which the results are presented.

Sometimes the number and order are important. For example, `tabulate` requires a minimum of one variable (for a one-way tabulation) and a maximum of two variables. When two variables are specified with `tabulate` the order is important - the first variable is the row variable and the second the column variable.

```
tabulate  agegroup smoke

           |          Smoking status
    agegr |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
       65- |        73        260        251 |       584
       70- |        41        228        231 |       500
       75- |        26        268        309 |       603
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500

. tabulate  smoke agegroup

   Smoking |                    agegroup
    status |       30-        65-        70-        75- |     Total
-----------+--------------------------------------------+----------
   Current |       129         73         41         26 |       269
        Ex |       361        260        228        268 |     1,117
     Never |       323        251        231        309 |     1,114
-----------+--------------------------------------------+----------
     Total |       813        584        500        603 |     2,500
```

Specifying either none or more than two variables after `tabulate` would result in an error message as shown below.

```
. tabulate
varlist required

. tabulate  sex agegroup race
too many variables specified
```

With statistical modelling commands like `regress` (linear regression) and `logistic` (logistic regression) the order of the variables is very important. With such commands the first variable in the list is the dependent or response variable and any subsequent variables are explanatory variables.

As with command names, variable names can be abbreviated providing they are not ambiguous. Stata is case-sensitive with variable names; it is good practice to keep all variable names in lower case.

If you use the wrong case or incorrectly spell a variable name Stata will return an error message saying that the variable could not be found. For example:

```
. summarize spb
variable spb not found
```

This should immediately alert you that there is an error in the variable list, most probably an error in spelling or perhaps the wrong dataset has been loaded.

If and In

We saw above in *GUI example 3* that we can restrict the operation of a command to a subset of the data using an `if` expression. We can also use `in` to restrict a command to certain rows in the dataset. If neither `if` nor `in` are used then all observations in the dataset will be used (or at least all non-missing observations).

By using the `if` qualifier followed by some *expression* we restrict the operation of the command to the observations in the dataset for which the *expression* is true. For example, to obtain summary descriptive statistics of *sbp* for subjects where *weight* is less than 90 kg:

```
. summ sbp if wt<90

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         sbp |       1890    123.3648    17.00194         72        189
```

When using `if` to restrict to observations *equal* to some value (number or string) then Stata's convention is to use a *double equals*.

For example, for a summary of *sbp* for observations where *agegroup* equals 1:

```
summarize sbp if agegroup==1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         sbp |        584    124.2089    16.98261         85        180
```

For observations where *race* is recorded as White (note that we enclose the string in quotes):

```
summarize sbp if race=="White"

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         sbp |       2066    124.5714    16.50045         74        189
```

If you use a single equals rather than a double equals you will get an error message saying `invalid syntax`. This is not the most helpful error message as it does not tell you where in the syntax the error is found. However, if you do get such an error message when you have an `if` expression in the syntax it usually means that you have used a single rather than a double equals.

The `if` expression can involve more than one condition but only one `if` is required. For example, to get a summary of *sbp* for observations where *weight <90* and *race* is "White":

```
. summarize sbp if wt<90 & race=="White"

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         sbp |       1498    123.8959    16.73348         80        189
```

Note that `if` only appears once and "`&`" is used to join the two conditions.

If expressions can contain the following symbols

Symbol	*Meaning*
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equal to
!=	not equal to
~=	not equal to (alternative to above)
&	and
|	or

Note: the symbol | meaning "or" is found on the backslash key

Note that care needs to be taken when using `if` with `>` when there are missing data. We will return to this point later.

The `in` range qualifier is used to restrict the operation of the command to a specified range of rows (observations) in the dataset as currently sorted. For example, to list *ptid* and *age* for the first 5 observations in the dataset:

```
. list ptid age in 1/5
     +-----------------+
     |      ptid   age |
     |-----------------|
  1. | C10011001    63 |
  2. | C10011002    66 |
  3. | C10011004    62 |
  4. | C10011005    66 |
  5. | C10011006    68 |
     +-----------------+
```

The range `1/5` refers to rows 1 to 5 in the dataset. The range can be specified as either a single number e.g. `in 1` or as range between two numbers as in the example above.

Using

Some Stata commands require a filepath/filename to be specified. For example, in GUI example 6 when beginning a log-file we specified the location and name of the log-file with using.

```
. log using "H:\Stats Computing/Stata/logfiles\example1.log”
```

When importing data from non-Stata formats we will sometimes have to use using to tell Stata where the data are and the name of the file.

Options

Nearly all Stata commands allow a number of options which modify what the command does. These must be separated from the main command syntax by a comma. For example, the **`summarize`** command will, by default, display the number of observations, the mean, standard deviation and the minimum and maximum values.

```
. summarize age

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         age |      2500     68.6632    7.675033         43         95
```

Summarize has the option **`detail`** which requests a more detailed output of summary statistics. For example, compare the following with the output above.

```
. summarize age , detail

                         Age (years)
-------------------------------------------------------------
      Percentiles      Smallest
 1%           55             43
 5%           57             49
10%           59             50       Obs                2500
25%           63             51       Sum of Wgt.        2500

50%           68                      Mean            68.6632
                        Largest       Std. Dev.      7.675033
75%           74             91
90%           79             92       Variance       58.90613
95%           82             94       Skewness       .2433231
99%           86             95       Kurtosis       2.473595
```

When more than one option is specified the order is not important. However, they should be separated from the main command syntax by a single comma i.e. you do not need multiple commas for multiple options. For example, you will find that the commands,

```
tabulate  agegroup smoke  , chi row
tabulate  agegroup smoke  , row chi
```

produce the same output. A common mistake made with options is to omit the comma; generally this will result in an error message saying variable "*xxx*" not found, e.g.

```
tabulate  agegroup smoke chi
variable chi not found
r(111);
```

Type `help` *`commandname`* to find what options are available with each command.

Prefix commands
Most Stata commands may be preceded by a prefix command. The most commonly used prefix commands is `by` *`varlist`*`:` or `bysort` *`varlist`*`:` which causes the main command to be repeated over the groups defined in *`varlist`*.

We saw the use of the by prefix in *GUI example 2* where we obtained a two-way tabulation of age-group and smoking status by sex. In order to execute a command by levels of a variable or variables the dataset has to be sorted by that variable (or variables).

```
. sort sex
. by sex:tabulate agegroup smoke
```

This can be done in a single step either as

```
. by sex , sort: tabulate agegroup smoke
```

or

```
. bysort sex: tabulate agegroup smoke
```

The second is the simplest to remember.

2.3 Getting help in Stata

Stata has an extensive built-in and on-line help system providing a wealth of information to help you learn and use Stata.

The options under the *Help* menu (see Figure 2.6) include links to the PDF documentation, Advice, Contents, Search and Stata Command. These are described briefly below.

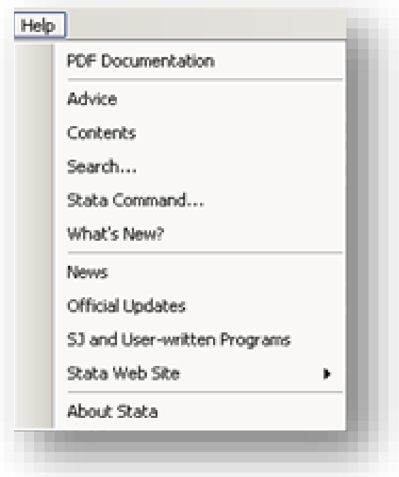


Figure 2.6: Stata's Help menu

Help > PDF documentation

Version 17 comes with complete PDF documentation, including *Getting Started with Stata, Base Reference Manual*, *User's Guide*, *Data-Management Reference Manual*, *Graphics Reference Manual*, and all the programming and specialized statistics manuals. The PDF documentation is linked into the existing interactive help file system.

Help > Advice

Advice provides a useful description of Stata's extensive help facilities as well as guidance on how to make best use of them.

Help > Contents

This provides help in the form of a category listing, which includes: Basics (language syntax, functions, etc.); Data management (inputting data, creating new variables, etc.); Statistics (tables, estimation commands, etc.); Graphics (scatter plots, bar charts, etc.) and Programming and matrices (do-files, matrices, etc.).

Help > Search

This enables the user to search for *keywords* in official help-files, Stata manuals, FAQs, the Stata Journal and Stata Technical Bulletin. There are also a number of trusted academic net resources that can be searched, including sites at UCLA, Boston College, Imperial College and UCL.

Searches can be carried out directly from within the Command window with the command `search` or `net search`. For example, to search for help on the keyword *meta* within the Stata documentation and FAQs;

```
. search meta
```

To search the net resources type;

```
. net search meta
```

You can find advice on how to carry out searches under *Help > Advice*.

Help > Stata Command

This provides help for specific Stata commands. Stata's help files provide information on the command syntax, a brief description of what the command does, a summary of the options available, links to the related dialog box and the PDF documentation, examples of the command syntax and links to related commands. Help files can be viewed either in a Stata viewer or in the Results window.

For example, to open the help file for `summarize` from within the Command window and view it in the Stata viewer;

```
. help summarize
```

Also try out the Stata YouTube channel.

2.4 Creating and Using Do-files

It may at first seem preferable to access Stata's commands through the Graphical User Interface. Why go to the trouble of learning the command syntax if you can produce exactly the same results by point-and-click? The reason why is the 'do-file'.

Whilst it is possible to use Stata interactively, i.e. to use the GUI or to submit commands in the command window, a far better alternative for serious research projects is use a do-file. A do-file is a text file containing a series of commands. The commands in the do-file can then be submitted, either in part or whole, as a sequence of commands. Stata has its own built-in file editor called the *Do-file Editor* and files created using Stata's editor are saved with the extension *.do*. It is also possible to use other text file editors to create do-files.

There are many advantages to using do-files. In particular, time is saved if it is necessary to repeat analyses e.g. routine reports during an ongoing study, or if an error is found in the data. In this case, the data are updated and the do-file is rerun.

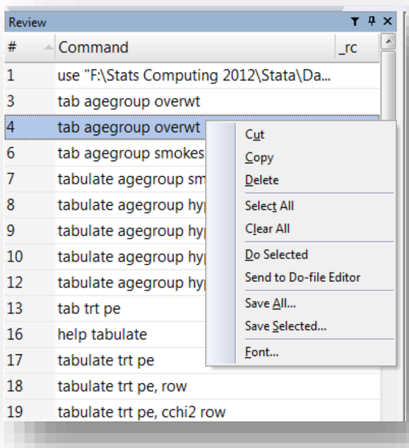
Also very importantly the do-file serves as a record of how the data were managed, processed and analysed. A great importance is placed on audit trails in medical research and the do-file is an excellent starting point.

Creating a New Do-file

Do-files can be created in a number of ways.

In chapter 1 we saw how to save the contents of the History window at the end of a Stata session. It is also possible to send some or all of the commands in the History window to a new do-file, at the end of, or during, an interactive session. This is done by selecting some or all the commands, right-clicking in the History window and selecting '*Send to Do-file editor*'.

However, remember that the History window contains a complete history of all commands submitted in the Command window, including those with errors, typos, etc. So creating a do-file in this way will require some discernment and editing.

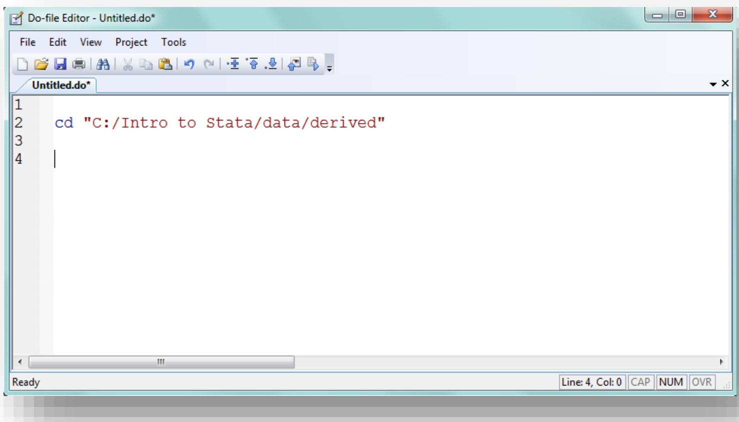


```
#   Command                                   _rc
1   use "F:\Stats Computing 2012\Stata\Da...
3   tab agegroup overwt
4   tab agegroup overwt
6   tab agegroup smokes
7   tabulate agegroup sm
8   tabulate agegroup hy
9   tabulate agegroup hy
10  tabulate agegroup hy
12  tabulate agegroup hy
13  tab trt pe
16  help tabulate
17  tabulate trt pe
18  tabulate trt pe, row
19  tabulate trt pe, cchi2 row
```

Figure 2.7 Creating a do-file from the History window contents

Using the *Save All* or *Save Selected* option will open the Save Review Contents dialog box. With this dialog box you can specify the location (*Save in*) and the name of the do-file. The default extension is *.do* but the files can be saved with other extensions, for example, *.txt*.

Using the *Send to Do-file Editor* option will automatically open the built in Do-file editor and dump the selected commands from the History window into a new do-file. This will initially be named *Untitled1.do* or something similar. The name appears on a tab just below the Do-file window menu bar. You will note that there is an asterisk next to the name, indicating that the file has changed since it was last saved - in fact the file has not yet been saved at all. We need to save the file to disk using the *File > Save as* menu within the Do-file Editor window.



```

cd "C:/Intro to Stata/data/derived"


```

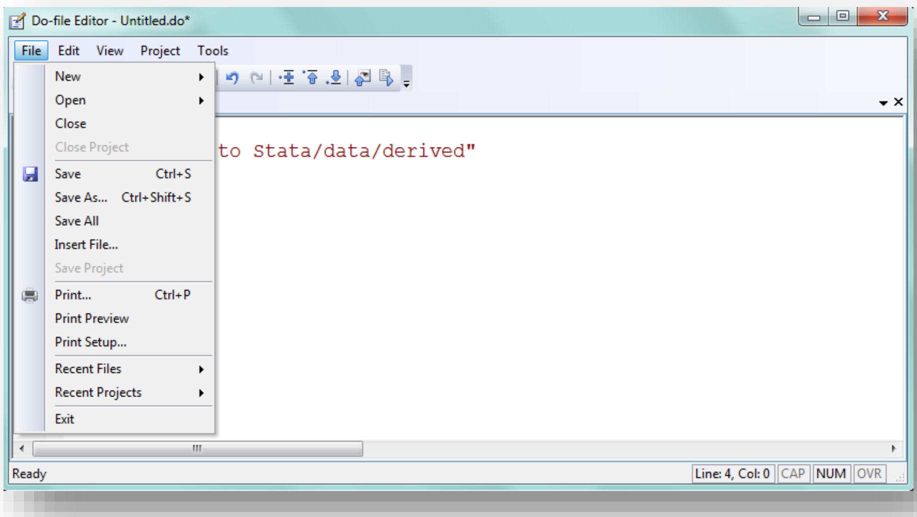
Figure 2.8: Stata's Do-file Editor window

Perhaps the best way to create a do-file for a new research project is to start from scratch using Stata's built in *Do-file editor*. The Do-file editor can be opened either through the *Window > Do-file editor* menu or the *Do-file editor short cut button*.

Commands are then typed directly into the do-file editor. The do-file can then be saved to disc using the *File > Save* menu from within Stata's Do-file editor. If building up a do-file in this way it is very important to save the file regularly as this is where most of your time is being invested.

Opening a previously created do-file

Previously created do-files are best opened from within Stata. This is done by opening the do-file editor (as described above) and using the *File > Open* menu within the do-file window.



```
to Stata/data/derived"
```

Figure 2.9: File menu in the Do-file editor window

It is possible to have several do-files opened within one Do-file Editor window – the names are displayed in tabs below the menu bar, as well as to have several Do-file Editors opened.

Executing Commands Saved in a Do-file

Commands saved within a do-file can be executed using the execute/do button on the toolbar.

To submit part of the do-file highlight the required section using the mouse and then click the *do* button. To execute all the commands contained in the do-file in sequence click the execute button without any of the commands or comments being highlighted.

Note that when commands are executed from within a do-file the History Window records only the do command (a single command) and not the individual commands executed within the file. However, both the commands and the output appears in the Results window and the log-file (if opened) if executed using the do command/button.

Note that when Stata encounters an error in a do-file, the error is reported and the do-file is stopped at the point of the error. This means that all the commands up to the point of the error will have been executed, but not commands after the point of error.

Adding Comments to Do-files

It is good practice to add comments to your do-files. For example, the first few lines of a do-file may contain information about the purpose of the do-file, the project name, the author, the date of creation and the date last edited.

It is possible to 'comment out' a single line in a do-file using an asterisk (*). E.g.,

```
* INTRODUCTION TO STATA: October 2022
```

If the asterisk was omitted from the first line above Stata would look for a command called INTRODUCTION and since no such command exists give an error message.

Comments can also be added at the end of a line using two forward slashes. E.g.,

```
tabulate agegroup diab , row // diab = diabetes 0=No 1=Yes
```

To comment out a series of lines in a do-file use a forward slash asterisk (/*) at the beginning of the section and an asterisk forward slash (*/) at the end of the section. E.g.

```
/*
The following errors were discovered:
Some heights were measured in metres
One case of date of birth & date of screening have been transposed
*/
```

Dealing with Long Commands in Do-files

Stata views a command as being submitted at the carriage return character. Sometimes, particularly with graphic commands, a command line may exceed the width of the visible page making it awkward to view and edit. In this case it may be wished to have the command run over several lines in the do-file.

A simple way of dealing with a moderately long command line is to place a triple forward slash at the end of an unfinished line. E.g.

```
summarize sbp dbp hrate qrsint gfrate ///
age wc if sex==2
```

2.5 Saving Results in Log Files

Stata has the ability to send a copy of everything that appears in the *Results window* to a file, called a log file. The log file is a record of everything that appears in the Results window, including commands, output, error messages, comments, etc. Results that appear in other windows, such as the Graph window, need to be saved separately.

Starting a Log File

Before a log file is started any output appearing in the Results window will not be captured. It is good practice to start a log file whenever you begin any serious work in Stata, particularly if you are working without a do file. Log files serve as a record or audit trail of all your work carried out during a Stata session. Once a log file is opened all subsequent results are immediately written to the file meaning that they can protect you from disasters such as sudden power failures. Log files can be started using either the *File > Log > Begin* menu or the `log using` command.

Log files can be created in two different formats. The default format for log files is *SMCL* (Stata Markup & Control Language). Log files in SMCL format preserve all the formatting and links from the Results window. These can be opened and viewed using the Stata Viewer but cannot be easily edited or copied and pasted into different software packages. Alternatively, it is possible to save the log file as a plain-text file; for keeping a record of your results that you can edit and then print, it is best to save your results as a text file. It is possible to translate a *.smcl* log file into a text file using the *File > Log > Translate* menu.

The command syntax to open a log file called *mylog* in the current working directory in text format is:

```
. log using mylog.log
```

or

```
. log using mylog , text
```

Unless you specify the directory, the log file will be saved in the current working directory. The *.log* extension and the option `text` are alternative ways of specifying text format rather than SMCL.

When using the File menu to begin a log file you will find the S*ave as type* drop down menu at the bottom of the dialog box. The default is Formatted Log (*.smcl) with Log (*.log) as the alternative.

Stata puts a header at the beginning of the log file that records the name of the log file (not the same as the filename), the pathway and filename, the log type, and a date-stamp showing the date and time the log file was opened.

```
name:  <unnamed>
       log:  h:\intro to stata\mylog.log
  log type:  text
 opened on:   2 Sep 2022, 10:55:35
```

When a log file has been started the Results window status bar (at the bottom of the Results window) indicates that the log file is on as well as what format (smcl or text). You can tell Stata to start logging at any point during a Stata session, but remember that only the output that appears on the screen *after* opening the log file will be captured.

It is possible to have multiple log files open. This requires using the `name(logname)` option. See `help log` for more details.

Closing a Log File

Stata will continue to save output to the log file until you close the log file with the *log close* command or exit Stata (when Stata automatically closes the file).

Note that you do not need to save the log file.

When you close the log file Stata adds a footer to the end of the file recording the name of the log file, the pathway and filename, the log type, and a date-stamp showing the date and time the log file was closed.

```
      name:  <unnamed>
       log:  h:\intro to stata\mylog.log
  log type:  text
 closed on:   2 Sep 2021, 11:22:59
```

Replacing and Appending Log Files

If the file *mylog* already exists, and you would like to overwrite it, use the `replace` option, i.e. submit the command,

```
. log using mylog , replace
```

If the file already exists and you would like to add more output to the end of it, use the `append` option, i.e. submit the command,

```
. log using mylog , append
```

Viewing and Printing a Log File

SMCL log files are best viewed using the Stata viewer. This can be accessed using the *File > Log > View* menu. The viewer has a print option.

Text format log files can be opened and viewed in text file editor or word processor such as Notepad or Word. Once in Word, the font may need to be changed to a non-proportional or monospaced font such as Courier New – tables and other output will not be aligned if you don't do this. Page breaks can also be put in appropriate places. It is good practice to edit the log file to remove any unwanted output before printing.

Adding Comments to Log Files

It is often helpful to add comments to your output during a session. Stata will treat any line starting with a '*' as a comment and will ignore it. For example, if you wanted to add the comment '*Stata session 1*' to your output you would enter the line

```
 * Stata session 1
```

Stata will ignore the comment, but the comment will appear in your log file.

Summary of Log File commands

Syntax	*Purpose*	*Example*
log using *filename*	Open a SMCL log file in the current directory	log using log1
log using "*pathway/filename*"	Open a SMCL log file in a specified location	log using "c:/temp/log2"
log using *filename* , text	Open a plain text log file	log using log3 , text
log using *filename*.log	Same as above	log using log4.log
log close	Close the log file	log close
log off	Temporarily stop sending results to the log file (without closing log file)	log off
log on	Resume sending results to the log file	log on
set logtype {text | smcl }	Sets the default log type to text or SMCL	set logtype text

Chapter 3: Inspecting Data

Aims and Objectives of Chapter 3

By the end of this chapter you should:

- know how to obtain a description of the data in memory
- be able to sort and view the data in the Data Browser and Results Window
- be able to obtain summaries of variables
- be able to check for errors and identify outlying or unusual values
- be able to check for duplicated values

The following commands are used in this chapter: `describe`, `browse`, `sort`, `gsort`, `list`, `summarize`, `tabulate`, `codebook`, `histogram`, `twoway scatter`, `duplicates report`.

3.1 Inspecting Data

Having loaded a dataset into memory it is very tempting to rush straight into the statistical analysis to get out the odds-ratio or p-value you are interested in. However, and this is really important, before you start any analysis you should take time to inspect your data.

Inspecting the data has two main purposes.

Firstly, and most importantly, the goal is to identify any errors or spurious values. You may have loaded your data successfully, carried out the appropriate statistical tests and used the appropriate statistical model, but, if there are serious errors in your data, the results may be completely invalid. Once identified the errors should be corrected where possible or dealt with in some other way if not. This is not always very exciting work but it is essential work.

The second goal in inspecting the data is to familiarise yourself with what data you have. What variables do you have? How many observations are there? Are there any missing values? How are the variables distributed? These are all key questions you should address at the beginning of any analyses.

In this chapter we will visit a number of commands that Stata has for looking at data, checking distributions of continuous and categorical variables, searching for spurious values or inconsistencies in the data and checking for duplicated values.

3.2 Describing a Dataset

The `describe` command gives a general summary of the dataset in memory or of a Stata data file saved on disk. It displays the filename and directory, the number of observations and variables, the date/time that the data file was created, the names of the variables and details of storage type, the display format, and details of any value and variable labels. For example, if we have loaded *bl_demog.dta* to obtain a description of the entire set the command syntax is:

```
describe

Contains data from H:\Stats Computing\Stata\Exercise 3\bl_demog.dta

  obs:         2,500
 vars:            21                          14 Sep 2015 16:09
 size:       250,000
----------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
----------------------------------------------------------------------------
ptid            str9    %9s                   Patient ID
birthdt         float   %d                    Date of birth
age             byte    %8.0g                 Age (years)
agegroup        float   %9.0g      agegroup   Age-group
sex             byte    %8.0g      sex        Sex
randdt          float   %d                    Date randomized
trt             str7    %9s                   Treatment allocation
smkstat         long    %13.0g                Smoking status
smoke           str7    %9s                   Smoking status
race            str5    %9s                   Race
hfdiag          byte    %13.0g                Heart failure diagnosis
wt              double  %10.0g                Weight (kg)
ht              double  %10.0g                Height (cm)
wc              double  %10.0g                Waist circumference
wc_unit         str2    %9s                   WC units
sbp             float   %9.0g
dbp             float   %9.0g
hrate           int     %8.0g
egfr            double  %10.0g                eGFR (ml/min/1.73msq)
lvef            double  %10.0g                LVEF (%)
diab            byte    %10.0g
----------------------------------------------------------------------------
Sorted by:  ptid
```

As we have not specified a variable list following the command name Stata's default is to describe all variables. Typing `describe` followed by a list of variable names describes only those variables in the list. For example,

```
describe ptid age sex
```

When starting out on a set of analyses for a new research project it can be very useful to have a hard copy of the output from the `describe` command for each of the data files. These can be kept in a folder for easy reference and annotated as necessary.

It is also possible to obtain a description of a Stata data file saved on disk without loading the data. For example, to obtain a description of the Stata data file *fup_vitals1.dta,*

```
describe using fup_vitals1.dta
```

The command can also be accessed using the *Data > Describe data > Describe data in memory* menu. See `help describe` for a description of the options available.

3.3 Viewing the Data

One of the first things to do having loaded a dataset into memory is to actually look at the data. Do not underrate the value of looking at the data with your eyes! You can pick up many issues by simply 'eyeballing' the data.

Data can be viewed either in the *Data Editor* using the `browse` command or in the *Results window* using the `list` command. We will consider both these commands along with the `sort` command which enables us to view the ends of distributions.

The Data Editor Window

The *Data Editor* window can be opened in either *Browse* or *Edit* mode and provides a live view of the data. In *Browse* mode the data can be viewed but not altered, whereas in *Edit* mode the data can be edited. So, if you are simply interested in looking at the data it is best to open the Data Editor in *Browse* mode. We can do this either using the *Data > Data Editor > Data Editor (Browse)* menu or the `browse` command or the *Data Editor (Browse)* shortcut button, shown in Figure 3.1. Note that the shortcut buttons for Edit and Browse mode are next to each other, with the Browse button on the right.

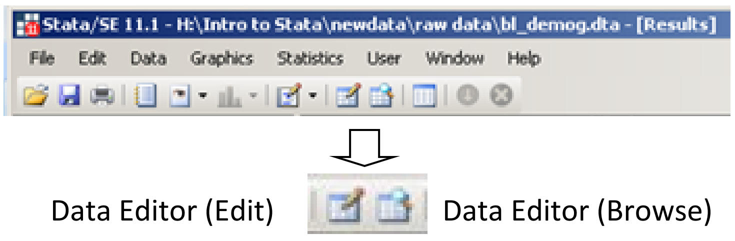


Figure 3.1 Data Editor Shortcut buttons

The syntax for the `browse` command is:

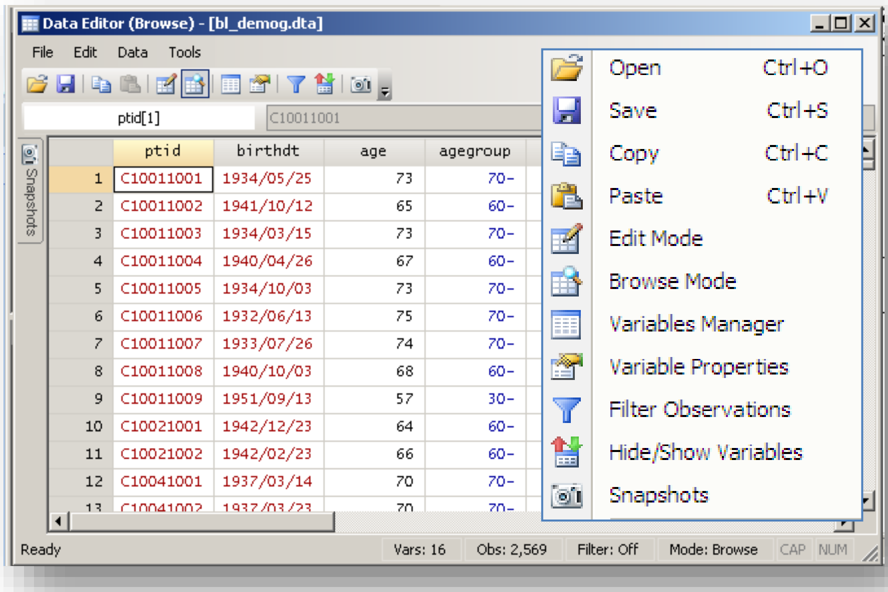
```
browse [variable list] [if] [in]
```

If no variable list is specified then the entire dataset is viewed.

A very useful feature of the `browse` command is that we can view a selection of variables. For example, we can view two variables next to each other in the Data Editor that may be some distance apart in the dataset. This saves scrolling back and forth along the spreadsheet or having to hide columns. If a variable list is specified then only these variables will be shown in the *Data Editor*. Note that the variables will appear in the order they are found in the dataset and not in the order in the variable list.

Clicking on the *Data Editor (Browse)* shortcut button will open the Data Editor window in browse mode and display all the variables in the dataset. The data are viewed in a rectangular matrix with observations appearing in rows and variables in columns.

The Data Editor window has its own drop down menus and toolbar (see Figure 3.2). From within this window we can limit the data being viewed. The number of observations (rows) can be reduced using the *Filter observations* and the number of variables can be reduced using the *Hide/Show Variables* buttons on the data browser shortcut toolbar (see Figures 3.2 and 3.3).

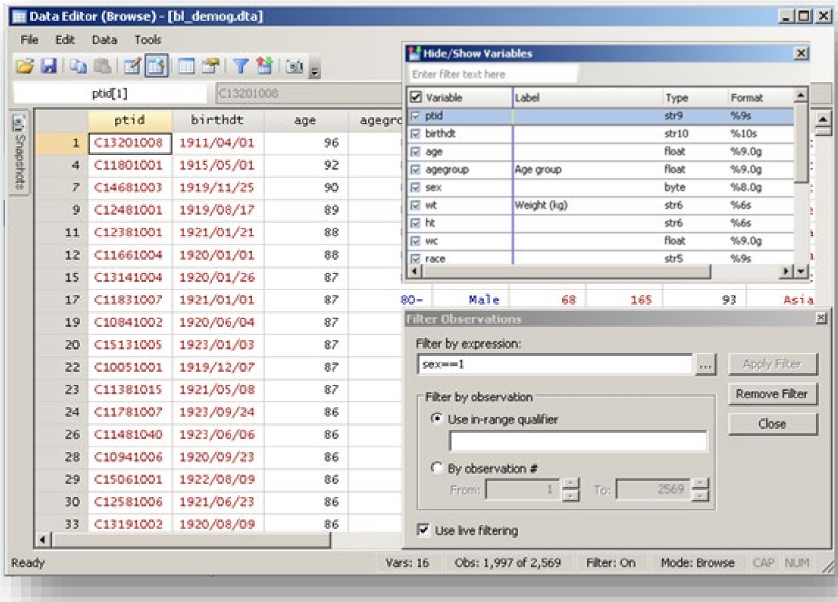


ptid[1] C10011001

	ptid	birthdt	age	agegroup
1	C10011001	1934/05/25	73	70-
2	C10011002	1941/10/12	65	60-
3	C10011003	1934/03/15	73	70-
4	C10011004	1940/04/26	67	60-
5	C10011005	1934/10/03	73	70-
6	C10011006	1932/06/13	75	70-
7	C10011007	1933/07/26	74	70-
8	C10011008	1940/10/03	68	60-
9	C10011009	1951/09/13	57	30-
10	C10021001	1942/12/23	64	60-
11	C10021002	1942/02/23	66	60-
12	C10041001	1937/03/14	70	70-
13	C10041002	1937/03/23	70	70-

Ready Vars: 16 Obs: 2,569 Filter: Off Mode: Browse CAP NUM

Figure 3.2: The Data Editor (Browse mode)



ptid[1] C13201008

	ptid	birthdt	age	agegro
1	C13201008	1911/04/01	96	
4	C11801001	1915/05/01	92	
7	C14681003	1919/11/25	90	
9	C12481001	1919/08/17	89	
11	C12381001	1921/01/21	88	
12	C11661004	1920/01/01	88	
15	C13141004	1920/01/26	87	
17	C11831007	1921/01/01	87	80-
19	C10841002	1920/06/04	87	
20	C15131005	1923/01/03	87	
22	C10051001	1919/12/07	87	
23	C11381015	1921/05/08	87	
24	C11781007	1923/09/24	86	
26	C11481040	1923/06/06	86	
28	C10941006	1920/09/23	86	
29	C15061001	1922/08/09	86	
30	C12581006	1921/06/23	86	
33	C13191002	1920/08/09	86	

80- Male 68 165 93 Asia

☑ Variable	Label	Type	Format
☑ ptid		str9	%9s
☑ birthdt		str10	%10s
☑ age		float	%9.0g
☑ agegroup	Age group	float	%9.0g
☑ sex		byte	%8.0g
☑ wt	Weight (kg)	str6	%6s
☑ ht		str6	%6s
☑ wc		float	%9.0g
☑ race		str5	%9s

Filter by expression: sex==1

Filter by observation
[✓ Use in-range qualifier]
By observation # From: 1 To: 2569

☑ Use live filtering

Ready Vars: 16 Obs: 1,997 of 2,569 Filter: On Mode: Browse CAP NUM

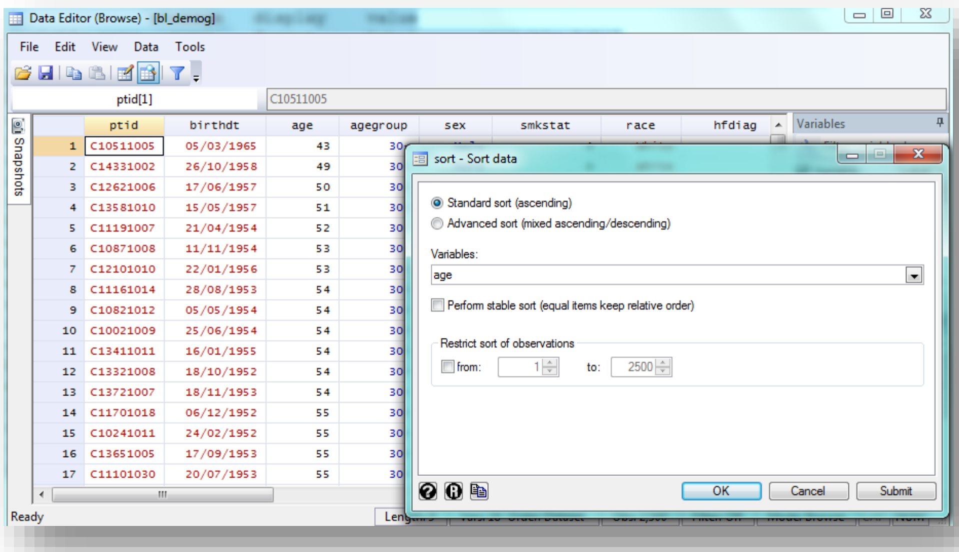
Figure 3.3: Filtering observations and Hiding variables in the Data Editor

Note that neither hiding variables nor filtering observations makes any changes to the dataset in memory. They simply change our view of the data.

Sorting data in the browser

When we open the *Data Editor* we get a live view of the data i.e. we view in the order in which the dataset is currently sorted. For example, the data may be sorted by a subject id number. When checking for errors it can be very useful to view the data sorted by another variable, say by date of birth or age. Note that when we sort the data all the observations for each subject stay together.

We can sort within the *Data Editor* by using the *Data > Sort* menu which opens the sort dialogue box (Figure 3.4). We can select *Standard* (ascending sort only) or *Advanced* (mixed ascending/descending sort). In Figure 3.4 we have selected the *Standard sort* option and sorted on the variable *age*. We can see that the youngest patient in the dataset is 43 years old.



	ptid	birthdt	age	agegroup
1	C10511005	05/03/1965	43	30
2	C14331002	26/10/1958	49	30
3	C12621006	17/06/1957	50	30
4	C13581010	15/05/1957	51	30
5	C11191007	21/04/1954	52	30
6	C10871008	11/11/1954	53	30
7	C12101010	22/01/1956	53	30
8	C11161014	28/08/1953	54	30
9	C10821012	05/05/1954	54	30
10	C10021009	25/06/1954	54	30
11	C13411011	16/01/1955	54	30
12	C13321008	18/10/1952	54	30
13	C13721007	18/11/1953	54	30
14	C11701018	06/12/1952	55	30
15	C10241011	24/02/1952	55	30
16	C13651005	17/09/1953	55	30
17	C11101030	20/07/1953	55	30

[✓ Standard sort (ascending)] Advanced sort (mixed ascending/descending)

Variables: age

Perform stable sort (equal items keep relative order)

Restrict sort of observations
from: 1 to: 2500

Figure 3.4: Sorting in the Data Editor

Under the *Ascending and descending sort* option we can specify a descending sort by inserting a minus sign (-) before the variable name. A plus sign (+) specifies an ascending sort. We can sort by more than one variable and include both ascending and descending sorts.

If you look at the commands echoed to the *History Window* following the *Standard* and *Advanced* sort you will see that there are two different commands names; the command `sort` relates to the *Standard sort* and `gsort` to the *Advanced sort*.

Sort by variable *age* in ascending order:

```
. sort age
```

Sort by agegroup in ascending order and sbp in descending order:

```
. gsort + agegroup - sbp
```

As mentioned above, when sorting by one or more variables the order of all other variables will be changed accordingly i.e. the whole dataset is sorted not just the variables specified.

Listing Data in the Results Window

The Data Editor provides us with a live view of the data. The `list` command provides a data listing *in the Results window.* This means that we can capture a record of the observations in a log file, if one is open. The syntax for the command is:

```
. list [varlist] [if] [in]
```

Typing `list` without any subsequent variable list will result in a listing of the entire dataset as it is currently ordered; it's not often that you will want to do that. Typing `list` followed by a list of variable names results in a listing of all observations on those variables specified.

More often it may be desired to list only a limited number of observations on a restricted set of variables. For example, to list the first 5 observations in the dataset for variables *ptid*, *birthdt* and *wc*:

```
. list ptid birthdt wc in 1/5

     +-------------------------------+
     |      ptid       birthdt    wc |
     |-------------------------------|
  1. | C10011001   18/Nov/1942   116 |
  2. | C10011002   19/Jul/1940   107 |
  3. | C10011004   25/Jul/1944   112 |
  4. | C10011005    3/Oct/1939    70 |
  5. | C10011006   17/Dec/1937   118 |
     +-------------------------------+
```

To list the *last 5 observations* in the dataset we specify *negative* numbers which Stata treats as counting 'from the last row' of the dataset.

```
. list ptid birthdt wc in -5/-1

      +-------------------------------+
      |      ptid       birthdt    wc |
      |-------------------------------|
2496. | C20021033   13/Feb/1935   112 |
2497. | C20021035    3/May/1948   134 |
2498. | C20021036   10/Jul/1930   103 |
2499. | C20051001   19/Aug/1934    82 |
2500. | C20051002   30/Sep/1946    77 |
      +-------------------------------+
```

Note that the -5/-1 essentially means 1/5 counting from the last row of the dataset – as it is currently sorted.

3.4 Commands for summarising distributions and checking for errors

Codebook

The single most useful command for checking data is `codebook`. This is particularly useful in that it produces a summary for each variable that takes into account the format of the data e.g. numeric, categorical, string etc.

For numeric variables codebook produces a summary consisting of the range, number of unique values, units, mean and SD and the number of missing values. The range immediately identifies spurious values lying outside the expected range. The number of unique values can help identify problems with unique identifiers.

If the variable is categorical the same summary is produced, but with a frequency tabulation rather than a mean and SD (the default maximum number of unique values for Stata to regard a variable as categorical rather than continuous is 9).

For string variables Stata presents the number of missing values, number of unique values and examples of the various strings. For date variables (numbers stored as elapsed dates) then the summary will include the range, the median date and other percentiles. Note that in the output from codebook shown below we can see that the date is stored as a string and not as an elapsed date – more about dates later. The syntax for the command is:

```
. codebook [varlist] [if] [in]  [ , options]
```

As with a few other Stata commands if no variable list is specified then the command is executed on all the variables in the dataset in memory. Below we have executed the command on one variable at a time:

```
. codebook  ptid

-------------------------------------------------------------------------------
ptid                                                                 Patient ID
-------------------------------------------------------------------------------

                  type:  string (str9)

         unique values:  2500                     missing "":  0/2500

              examples:  "C10951001"
                         "C11861013"
                         "C12501006"
                         "C13491008"
```

The variable *ptid* (patient identifier) is the variable that uniquely identifies each patient in our trial and which links most of the data files for this project. The output from codebook shows us that it is a string variable consisting of 9 characters. Importantly we see that there are 2500 unique values and no missing values. If there were less than 2500 unique values this would indicate that there are duplicated patient ids.4 example strings are presented.

```
. codebook  age

-----------------------------------------------------------------------------
age                                                               Age (years)
-----------------------------------------------------------------------------

                  type:  numeric (byte)

                 range:  [43,95]                      units:  1
         unique values:  47                       missing .:  0/2500

                  mean:   68.6632
             std. dev:   7.67503

           percentiles:        10%       25%       50%       75%       90%
                                59        63        68        74        79
```

As the variable is a numeric variable, with more than 9 unique values, `codebook` presents the range, mean, standard deviation and a selection of percentiles including the median (50%). The ages of the patients in our dataset range from 43 to 95 (years) with a mean age of 68.7 and median age of 68 years. The variable is labelled and there are no missing values.

```
. codebook sex

----------------------------------------------------------------------------
sex                                                              (unlabeled)
----------------------------------------------------------------------------
                  type:  numeric (byte)
                 label:  sex

                 range:  [1,2]                        units:  1
         unique values:  2                        missing .:  0/2569

            tabulation:  Freq.   Numeric  Label
                          1997         1  Male
                           572         2  Female
```

The variable *sex* is a numeric variable stored as a byte. As it only takes two unique values (1 or 2) codebook deals with this as a categorical variable and presents a frequency tabulation rather than mean, standard deviation as above with *age*. The table shows the numeric values and the label attached to each value. These are called value labels (more later).

See `help codebook` for more details.

Histogram

Another good way of checking for outlying values for a continuous variable, and examining the shape and spread of the distribution, is to plot a histogram. We can do this using the `histogram` command. For example to plot a histogram of waist circumferences:

```
. histogram  wc  , normal freq
(bin=34, start=.70999998, width=5.3320588)
```

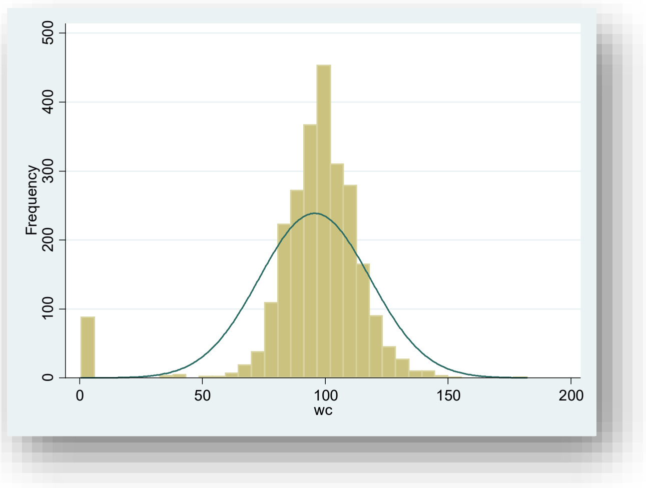


Figure 3.5 Histogram of waist circumference

The options specified in the command above are:

`normal`	requests a normal curve to be superimposed
`freq`	requests y-axis to show frequency rather than density.

The histogram shows that the variable *wc* is quite normally distributed, but that there is a blob of patients near to zero.

How could you plot a histogram for the group near zero?

Tabulate

Distributions of categorical variables can be checked with the `tabulate` command. By default `tabulate` ignores missing values but if we specify the option `missing` then `tabulate` presents a separate row for any missing values. For example,

```
. tabulate race  , missing

       Race |      Freq.     Percent        Cum.
------------+-----------------------------------
      Asian |        295       11.80       11.80
      Black |         60        2.40       14.20
      Other |         79        3.16       17.36
      White |      2,066       82.64      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

In this case there are no missing values and no strange values.

Two-way cross-tabulations of categorical variables might help identify inconsistencies in the data. For example, if we had smoking status (*smoke*) and years since stopped smoking (*smkstop*) we would expect *smkstop* to be missing for any *Never* smokers or any *Current* smokers.

Scatter Plots

Other graphs that might be useful for checking for errors include `twoway scatter` plots. These might be used to check for unusual combinations of values. For example if you have two variables that you expect to be strongly correlated (e.g. sbp and dbp) then a twoway scatter plot might help identify strange values. For example,

```
. twoway scatter sbp dbp  , ms(oh)
```

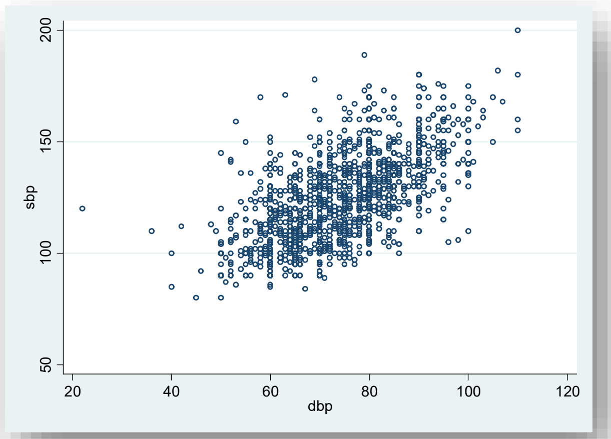


Figure 3.6: Scatter plot of sbp and dbp

In this case there are no obvious problems although there is one obvious outlier in terms of *dbp*. The option in the scatter plot command `ms(oh)` means use a hollow small circle for the marker symbol.

Summarize

For continuous numeric variables the command `summarize` with the option `detail` provides some additional descriptive statistics. For example,

```
sum  wc , d
                                wc
-------------------------------------------------------------
      Percentiles      Smallest
 1%          .82            .69
 5%          .95            .71
10%         1.09            .73       Obs                2369
25%           85            .73       Sum of Wgt.        2369

50%           96                      Mean           85.88955
                        Largest       Std. Dev.      35.65602
75%          106            143
90%          114            146       Variance       1271.351
95%          120            147       Skewness      -1.633495
99%          132            165       Kurtosis       4.513849
```

3.5 Checking for Duplicates

Above we saw that the command `codebook` can be helpful in identifying duplicated observations, i.e. check whether the number of unique values is the same as the number of observations. E.g.

```
---------------------------------------------------------------------------
ptid                                                             Patient ID
---------------------------------------------------------------------------

                   type:  string (str9)

          unique values:  2500                      missing "":  0/2500

               examples:  "C10951001"
                          "C11861013"
                          "C12501006"
                          "C13491008"
```

However, if there are missing values or if there is more than one variable by which we identify unique observations (e.g. patient id and centre id) then we need something more than `codebook`.

The `duplicates` set of commands enables us to investigate whether we have any duplicated observations in our dataset across any number of variables. It also helps to identify which are the duplicated records and can also be used to delete if appropriate.

The syntax for `duplicates` is similar to the `label` commands (`label variable`, `label define`, `label attach`) in that it has a series of subcommands. The syntax is:

```
. duplicates subcommand [varlist]
```

If no variable list is specified then Stata checks for duplicates across all the variables in the dataset. The subcommands include `report`, `list`, `tag` and `drop`

Duplicates Report

The command `duplicates report` produces a table showing observations that occur as one or more copies and indicating how many observations are *surplus* in the sense that they are the second, third, etc. copies of the first of each group of duplicates.

Example 1: Check whether there are any duplicate patient ids in the *bl_demog* data file:

```
. duplicates report ptid

Duplicates in terms of ptid

--------------------------------------
   copies | observations       surplus
----------+---------------------------
        1 |         2500             0
--------------------------------------
```

Here we see that there is only one copy of each value of the variable ptid i.e. there are 2500 unique patient ids out of 2500 observations (patients) in our dataset.

Example 2: Now load *fup_egfr.dta* and check for duplicate patient ids.

```
. use fup_egfr
. duplicates report ptid

Duplicates in terms of ptid

--------------------------------------
   copies | observations       surplus
----------+---------------------------
        1 |          339             0
        2 |          948           474
        3 |         1344           896
        4 |         1712          1284
        5 |         4040          3232
--------------------------------------
```

Here we find that we have only 339 observations where there is 1 unique value of *ptid*, 948 values of *ptid* where there are 2 copies (so 474 duplicated twice), 1344 values of ptid of which there are 3 copies (so 448 appearing three times each) etc.

Why are there so many duplicated values? If you look at the dataset you will see that there are multiple rows per patient because each visit for each patient is recorded on a separate row.

Repeat the command adding visit to the variable list.

```
. duplicates report  ptid  visit

Duplicates in terms of ptid visit

--------------------------------------
   copies | observations       surplus
----------+---------------------------
        1 |         8383             0
--------------------------------------
```

We now see that there is only one copy of each combination of *ptid* and *visit*.

Other subcommands

If we discover that there are some genuinely duplicated observations we can use `duplicates tag` to generate a new variable to identify the duplicate observations. The syntax requires us to specify a name for the new variable using the `gen(varname)` option. For example to generate a new variable called duplicates, which identifies duplicates in terms of ptid and visit, the syntax would be:

```
. duplicates tag ptid visit , gen(duplicates)
```

This new variable called duplicates will take the value 0 for all unique observations, the value 1 for all observations for which there is 1 extra copy, 2 if 2 extra copies, 3 if 3 extra copies, etc.

The command `duplicates drop` drops all but the first occurrence of each group of duplicated observations. This should be used with CAUTION.

Chapter 4: Creating and Combining Stata Datasets

Aims and Objectives of Chapter 4

By the end of this chapter you should:

- know how to enter data using the Data Editor or input command
- know how to import data into Stata from an Excel spreadsheet
- know how to save a Stata dataset
- be able to import data from tab or comma delimited text files
- be able to load data from a free format or fixed format text file
- know how Stata deals with data in memory and on disk
- know how to combine Stata datasets using append and merge

The following commands are used in this chapter: `input, save, import excel, import delimited, infile` and `infix, list, type, append, merge, drop.`

4.1 Entering Data using the Data Editor

Data can be entered interactively using the *Data Editor*.

Submitting the command `edit` or clicking on the *Data Editor* button opens up the data editor which looks like a standard spreadsheet. Data can be entered into the spreadsheet using the keyboard. Using the mouse select the cell into which the data is to be entered, type the data and press the *Enter* key (↵). The cursor keys or mouse are used to move up/down columns and across rows. Variable are stored in columns and records are stored in rows.

The storage type for each variable is determined automatically: if you type a number for the first entry, the new variable will be numeric; if the first entry is a string, the variable will be a string. By default the first variable will be called var1. We can rename the variable using the *Variable Properties* window which is on the right of the Data Editor window (see Figure 4.1).

Example: Here we will enter data from a published trial. Here is some text from the results section of the paper.

> *"The primary prespecified outcome at 1 year was observed in 28 of the 205 patients who received active treatment compared to 43 of the 197 patients who received placebo (p=0.03)."*

We can use Stata as a calculator to work out the number of patients who did not have an event.

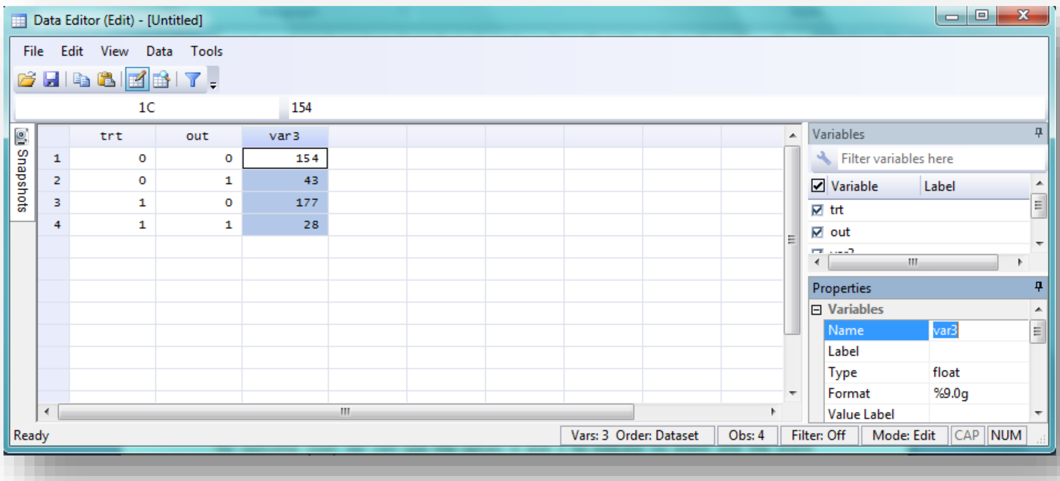
```
. display 205-28
177

. display 197-43
154
```

So in the active treatment group there were 28 patients with and 177 patients without the event; in the placebo group there were 43 with and 154 without the event.

This equates to the following two-by-two table.

Group	Outcome	
	No (0)	Yes (1)
Placebo (0)	154	43
Active (1)	177	28

We will need to create a dataset with three variables (one to indicate the treatment group, a second for the outcome and a third for the number of patients in each category) and four rows (one for each combination of treatment group and outcome). For treatment group (trt) we can use the values 0 and 1 for placebo and treatment respectively; for outcome (out) we can use the values 0 and 1 to indicate no event and the event respectively. We can name the third variable *freq* and put in the actual number of patients.



1C 154

	trt	out	var3
1	0	0	154
2	0	1	43
3	1	0	177
4	1	1	28

Variable	Label
trt	
out	

Variables

Name	var3
Label	
Type	float
Format	%9.0g
Value Label	

Vars: 3 Order: Dataset | Obs: 4 | Filter: Off | Mode: Edit | CAP | NUM

Figure 4.1 Entering data into the Data Editor

Clicking on a particular variable will cause the properties for that variable to appear in the *Variables Properties* window. We can type in a new name and press the Enter key.

We can now use Stata's frequency weight option to do some simple analyses of the data.

```
. tab trt outcome [fw=freq] , chi row

           |        outcome
       trt |         0          1 |     Total
-----------+----------------------+----------
         0 |       154         43 |       197
           |     78.17      21.83 |    100.00
-----------+----------------------+----------
         1 |       177         28 |       205
           |     86.34      13.66 |    100.00
-----------+----------------------+----------
     Total |       331         71 |       402
           |     82.34      17.66 |    100.00

          Pearson chi2(1) =   4.6098   Pr = 0.032
```

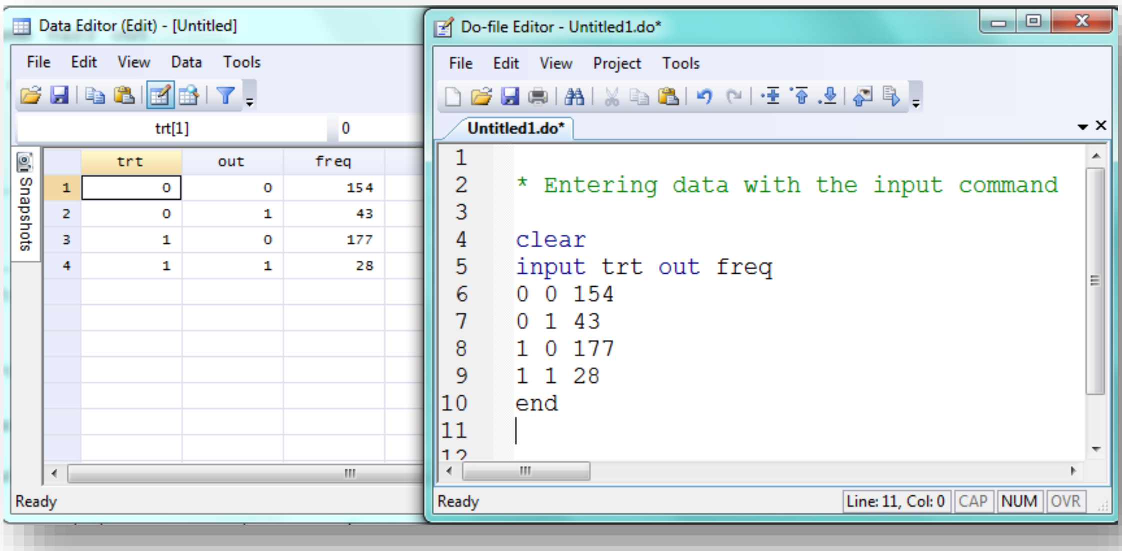
Note that `[fw=freq]` is not an option and comes before the comma. The `fw` means frequency weights and we use `=freq` to tell Stata the name of the variable containing the weights.

Having created the dataset in memory we can then save this to disk as a Stata dataset using the save command. E.g. `save dataset1`.

4.2 Entering Data using the input command

We could have entered the same data using the `input` command. This can be done either in the command window or more usually within a do-file (Figure 4.2).

The variable names are specified on the same line as the input command. This is then followed by the rows of data, each on a separate line. The command `end` tells Stata when the data entry is complete. The first observation in each row corresponds to the first variable named on the input command line.


Data Editor (Edit) - [Untitled]

	trt	out	freq
1	0	0	154
2	0	1	43
3	1	0	177
4	1	1	28

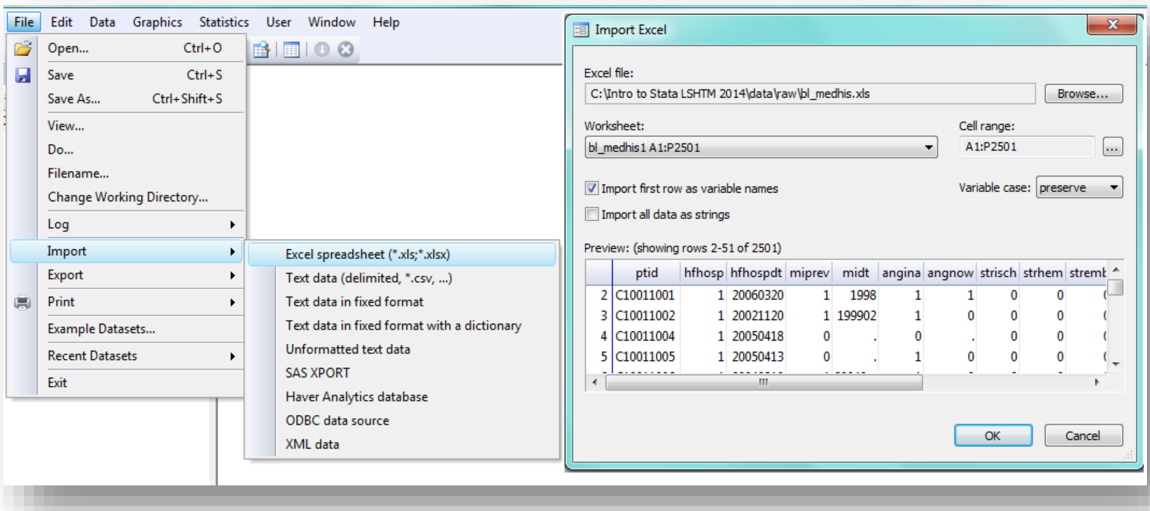
Do-file Editor - Untitled1.do*

```

* Entering data with the input command

clear
input trt out freq
0 0 154
0 1 43
1 0 177
1 1 28
end
```

Figure 4.2 Entering data with the input command

4.3 Importing Data from Excel

We can import data directly into Stata from an Excel spreadsheet either through the menus using *File > Import > Excel Spreadsheet* (Figure 4.3) or with the `import excel` command.



Excel file: C:\Intro to Stata LSHTM 2014\data\raw\bl_medhis.xls

Worksheet: bl_medhis1 A1:P2501

Cell range: A1:P2501

[✓] Import first row as variable names

[ ] Import all data as strings

Variable case: preserve

Preview: (showing rows 2-51 of 2501)

	ptid	hfhosp	hfhospdt	miprev	midt	angina	angnow	strisch	strhem
2	C10011001	1	20060320	1	1998	1	1	0	0
3	C10011002	1	20021120	1	199902	1	0	0	0
4	C10011004	1	20050418	0	.	0	.	0	0
5	C10011005	1	20050413	0	.	1	0	0	0

Figure 4.3 *Importing Data from Excel*

Note in Figure 4.3 that we specified the directory and Excel filename, we ticked the *Import first row as variable names* box and that the worksheet to be imported is *bl_medhis1*.

The equivalent Stata command to do this (assuming that this file is the current directory) is:

```
. import excel using bl_medhis.xls, firstrow sheet(bl_medhis1)
```

The Excel file is specified with `using` along with the appropriate file extension (either .xls or .xlsx).

The `firstrow` option tells Stata that the first row of the Excel file should be treated as variable names rather than data. If this option is not specified then the first row will be imported as data.

Data can only be imported from one sheet at a time. The option `sheet("sheetname")` indicates which sheet to import. If the `sheet()` option is not specified the default is to import the first worksheet.

Data can only be imported if there is no data in memory, or if the option `clear` is specified.

See `help import excel` for more details.

Saving a Stata Data File

Having imported the data from Excel into Stata we now need to save to disk as a Stata dataset. We can do this either through the *File > Save as* menu or the `save` command. E.g.

```
. save bl_medhis1
```

If the data file *bl_medhis1.dta* already exists on disk then if you are using the *File > Save as* menu Stata will give you the option of replacing the data file on disk with the file currently in memory or cancelling the save. If you are using the `save` command then Stata will return an error message saying:

```
. save bl_medhis1
file bl_medhis1 already exists
```

The `save` command has the option `replace` which forces Stata to overwrite the data file. When working within a do-file it is generally ok to add the option replace to any save command. This will stop the do-file breaking down. Remember it is important to keep at least one copy of the original 'raw' data.

```
. save bl_medhis1, replace
```

Once the Stata dataset has been created it can be loaded either through the *File > Open >* menu or with the `use` command. You can also double-click on a Stata data file, which automatically launches Stata and loads the dataset.

Data in Memory and Data on Disk

It is very important to be aware of how Stata works with data. Above we loaded data that existed on disk as an Excel file. As we do this Stata loads a copy of the data contained in those files into its memory (RAM). Any work carried out on the dataset currently in memory e.g. adding labels, generating new variables, etc. affects only the data in memory. The file from which the data was loaded is completely unaffected and remains in its original state. It is good practice to always keep (at least) one copy of the 'raw' data.

The process carried out above for the Medical History data is explained below.

Step 1: We started with the single Excel file (*bl_medhis.xls*) containing two worksheets saved on disk; this remains unchanged at the end of the process and can be regarded as our original or raw data. Keep this safe.

Step 2: We then loaded the first worksheet into Stata's memory. This only existed in Stata's memory i.e. it was not saved on disk.

Step 3: We then saved this file to disk (using `save` command) as a Stata data file. This file contains exactly the same information as the Excel file, but in a different format.

4.4 Importing Delimited Text Files

The `import` command can also be used to import data from other formats including tab delimited or comma delimited text files – often called CSV files. These are files that have been created by a spreadsheet style package.

We will demonstrate here using two files: *bl_labs1.txt* (a tab-delimited text file) and *bl_rand.txt* (a comma delimited text file).

Firstly we will view both files. We can do this either in a text editor e.g. notepad or alternatively within the *Results Window* by using the `type` command. E.g.

```
. type bl_labs1.txt, lines(5)

ptid    regid   creat   hb      pot     sodium  totbil
"C10161001"     1       71      12.6    4       140     15.4
"C10161002"     1       150     17.4    3.3     142     12
"C10161003"     1       115     11.7    4.7     132     23.9
"C10161004"     1       115     9.5     3.5     140     15.4
```

We have used the option `lines(#)` to restrict to the first few lines of the dataset. If we use the `showtabs` option we will see that the gaps between the variable names and the observations are tabs rather than multiple spaces.

```
. type bl_labs1.txt, lines(5) showtabs

ptid<T>regid<T>creat<T>hb<T>pot<T>sodium<T>totbil
"C10161001"<T>1<T>71<T>12.6<T>4<T>140<T>15.4
"C10161002"<T>1<T>150<T>17.4<T>3.3<T>142<T>12
"C10161003"<T>1<T>115<T>11.7<T>4.7<T>132<T>23.9
"C10161004"<T>1<T>115<T>9.5<T>3.5<T>140<T>15.4
```

For the bl_rand.txt file you should see that the first line or row contains the variable names and that names and values are separated (or delimited) by commas.

```
. type bl_rand.txt, lines(5)

rcode,indx_day,indx_mon,indx_year,consdt,randdt
2073,2,7,2006,"07-20-06","20060721"
2074,22,5,2006,"07-21-06","20060721"
2076,21,8,2006,"08-21-06","20060822"
2077,21,8,2006,"08-22-06","20060822"
```

Note in both these datasets that the first row contains the variable names and that each record is on a single separate line.

Importing a tab-delimited text file

To import the first of these files into Stata (assuming that the file is in the current working directory):

```
. import delimited using bl_labs1.txt, varnames(1)
(7 vars, 347 obs)
```

Stata reports that 7 variables and 347 observations have been imported.

Note that as before the syntax includes a command (`import`) and a subcommand (`delimited)` and again use `using` to specify the filename and extension. This time we use the option `varnames(#)` to specify which row (if any) contains the variable names. See what happens if you omit this option.
It is good practice to look at the data once it has been imported. We can do this using `browse` or `list`.

```
. list in 1/5

     +---------------------------------------------------------+
     |      ptid   regid   creat     hb   pot   sodium   totbil |
     |---------------------------------------------------------|
  1. | C10161001       1      71   12.6     4      140     15.4 |
  2. | C10161002       1     150   17.4   3.3      142       12 |
  3. | C10161003       1     115   11.7   4.7      132     23.9 |
  4. | C10161004       1     115    9.5   3.5      140     15.4 |
  5. | C10161005       1      97   13.6   4.9      140      6.8 |
     +---------------------------------------------------------+
```

It appears that the data have been imported correctly.

Importing a comma-delimited text file

To import the second file above i.e. the *bl_rand.txt* comma delimited text file we again use the `import delimited` command.

```
. import delimited using bl_rand.txt, varnames(1) delimiter(",")
(6 vars, 2500 obs)
```

With `import delimited` Stata expects that the delimiter will either be a comma or a tab so we don't really need to specify the `delimiter(",")` option. However, we include this option to demonstrate how other delimiters could be specified if required.

As before we should now look at the file and then save as a Stata file.

```
. list in 1/5

     +-----------------------------------------------------------+
     | rcode   indx_day   indx_mon   indx_y~r     consdt    randdt |
     |-----------------------------------------------------------|
  1. |  2073          2          7       2006   07-20-06  20060721 |
  2. |  2074         22          5       2006   07-21-06  20060721 |
  3. |  2076         21          8       2006   08-21-06  20060822 |
  4. |  2077         21          8       2006   08-22-06  20060822 |
  5. |  2078         28          8       2006   08-29-06  20060829 |
     +-----------------------------------------------------------+

. save bl_rand
file bl_rand.dta saved
```

4.5 Loading Free-Format Text Files

Free format text files are quite different to delimited text files. In the above two examples all the observations for a particular subject were found on a single line and the observations were separated by a distinct character (e.g. a comma or tab).

In free format files observations for a particular subject can go over many lines or indeed the observations for more than one subject can appear on a single line. Observations can be separated by single or multiple spaces. String variables must be enclosed in quotes, particularly if the strings contain spaces.

We will use the datafile bl_meds.txt to illustrate. As before the first thing we should do is to look at the file in a text editor or using the `type` command.

```
. type bl_meds.txt, lines(30)
"C10011001"  "Paroxetine"
"Aspirin"
"Enalapril"
"Risperidona"
"Furosemide"
"Carvedilol"
"Valsartan"
""
""
""
"C10011002"  "Losartan"
"Aspirin"
"Carvedilol"
"Digoxine"
"Rosiglitazone"
"Furosemide"
""
""
""
""
"C10011003"  "Losartan"
"Insuline"
"Aspirin"
"Furosemide"
"Salbutamol"
"Valsartan"
"Simvastatin"
"Amlodipine"
"Carvedilol"
"Clopidogrel"
```

The first thing we notice is that the data for each subject runs over several lines. All the observations here are strings and are contained in quotation marks including any missing or empty values. We can also see that there are no variable names.

At present the `import` command cannot deal with such data. The appropriate command for this type of dataset is `infile`.

With the `infile` command we will need to specify the names of the variables. Additionally we will need to indicate if any of the variables are strings and the length of the string. Finally we will need to specify the name of the file we are loading including its file extension.

We will load the bl_meds.txt dataset. The first variable is the patient identifier which we will call ptid. This variable is a string of 9 characters in length. The remaining 10 variables are also string variables. The length of the strings varies – it is not obvious what the longest string length for each of these variables so the best option is to specify a large number of characters and then compress afterwards. If we specify too short a length then the observations will be truncated.

```
infile str9 ptid str90 (med1 med2 med3 med4 med5 med6 med7 med8
med9 med10) using bl_meds.txt , clear
```

Note that *str9* is a string format meaning that the following variable is a string variable of maximum 9 characters. This comes immediately before the name of the variable *ptid* which it is being applied to. Rather than specifying a separate string format for each of the variables med1 to med10 we can specify a single string format (*str90*) and then apply this to all the following variables within the brackets. We chose a relatively large number 90 as the maximum length of the string to make sure none were truncated. We checked this was sufficient by using the compress command. As all the variables were then compressed to fewer than 90 characters we can be confident that no observations were truncated.

```
. compress
  med1 was str90 now str47
  med2 was str90 now str60
  med3 was str90 now str59
  med4 was str90 now str60
  med5 was str90 now str45
  med6 was str90 now str60
  med7 was str90 now str45
  med8 was str90 now str51
  med9 was str90 now str45
  med10 was str90 now str45
  (559,563 bytes saved)
```

Don't forget to look at the data.

```
. list ptid-med4 in 1/5

   +-----------------------------------------------------------------+
   |      ptid         med1         med2         med3            med4 |
   |-----------------------------------------------------------------|
1. | C10011001   Paroxetine      Aspirin    Enalapril       Risperidona |
2. | C10011002     Losartan      Aspirin   Carvedilol          Digoxine |
3. | C10011003     Losartan     Insuline      Aspirin        Furosemide |
4. | C10011004    Enalapril   Amiodarone    Thiazides       Acenocumarol |
5. | C10011005   Carvedilol    Enalapril   Amiodarone   Lanoxin Digoxina |
   +-----------------------------------------------------------------+
```

This command can also be accessed through the *File > Import > Unformatted Text Data* menu. Also see `help infile`.

4.6 Loading Fixed-Format Text Files

The final type of data we will consider is fixed format text files. These are rarely encountered but are dealt with here in case you should come across such a file.

For this example we will use the trtcodes.txt file. This contains the randomisation code, the randomisation date and the treatment group.

```
. type trtcodes.txt , lines(5)

2073200607211
2074200607212
2076200608221
2077200608221
2078200608291
```

The randomisation code is 4 characters in length and occupies the first 4 columns of the dataset. The randomisation date is 8 characters in length and occupies columns 5 to 12 of the dataset. The treatment group is just a single character (1 or 2) and occupies column 13.

Note that for each variable the observations are the same length and therefore occupy the same columns in the dataset. Note also that there are no separating characters and that the dataset does not contain any variable names.

As with the infile command we will need to specify the names of the variables, the format type if required (compulsory for strings) and to specify the data file to be loaded. Additionally we will need to specify the columns that each variable occupies.

```
infix rcode 1-4 str randdate 5-12 trtcode 13 using trtcodes.txt, clear
(2500 observations read)
```

Unlike the `infile` command we do not need to specify the length of the string since this is implicit from the column numbers that are specified.

```
. list in 1/5

     +----------------------------+
     | rcode   randdate   trtcode |
     |----------------------------|
  1. |  2073   20060721         1 |
  2. |  2074   20060721         2 |
  3. |  2076   20060822         1 |
  4. |  2077   20060822         1 |
  5. |  2078   20060829         1 |
     +----------------------------+
```

Note that if we had decided to import the randomisation date as a number rather than a string it would be best to specify the type as `long` format because of the size of that number. E.g.

```
infix rcode 1-4 long randdate 5-12 trtcode 13 using trtcodes.txt, clear
(2500 observations read)
```

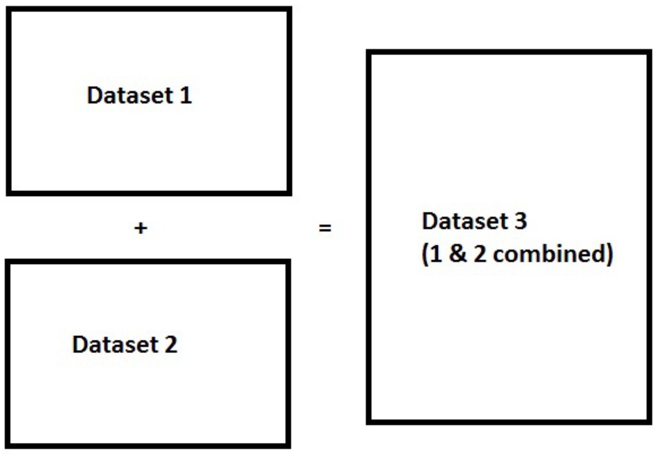
As we finish this section on creating datasets it is very important to again recall how Stata works with data. Above we loaded data that existed on disk as delimited files. As we do this Stata loads a copy of the data contained in those files into its memory (RAM). Any work carried out on the dataset currently in memory e.g. adding labels, generating new variables, etc. affects only the data in memory. The file from which the data was loaded is completely unaffected and remains in its original state. It is good practice to always keep (at least) one copy of the 'raw' data.

4.7 Appending Datasets

It is often the case in large clinical trials or studies involving a long period of follow-up that data will be stored in more than one file. For example, data on baseline demographics may be recorded and saved in one dataset, baseline blood measurements in another and follow-up measurements in yet more. It is then often necessary to combine these data files in to one large dataset in order to carry out a full analysis.

Stata has two main commands for combining datasets (`append` and `merge`). Which to use depends on whether you are adding more observations (making the dataset longer) or adding more variables (making the dataset wider). We start here with appending.

With the `append` command we are combining data files by stacking them one beneath the other. This means that the combined dataset will be longer (i.e. more observations) but will usually be the same width (i.e. same number of variables) as the individual data files as shown in the figure below.



The datasets being combined will generally contain the same variables (though some datasets may contain additional variables) but measured either at different time-points or at different centres.

For example, in a single trial we may have data files created at different centres, collecting information on the same variables (say age, sex, height, etc.) but for subjects recruited at each centre. The variables being collected are the same, but the subjects are different. Providing that the variable names are the same, e.g. age is called age in all datasets, we can combine these files by stacking or appending them one under the other.

For example, if we look at the first 5 observations in each of *bl_labs1.dta*, *bl_labs2.dta*, *bl_labs3.dta* and *bl_labs4.dta* we find that they contain the same 7 variables, but on different patients i.e. from different regions.

```
. use bl_labs1, clear
. list in 1/5, nolab

     +-------------------------------------------------------------+
     |      ptid   regid    creat     hb   pot   sodium   totbil |
     |-------------------------------------------------------------|
  1. | C10161001       1    70.72   12.6     4      140    15.39 |
  2. | C10161002       1   150.28   17.4   3.3      142    11.97 |
  3. | C10161003       1   114.92   11.7   4.7      132    23.94 |
  4. | C10161004       1   114.92    9.5   3.5      140    15.39 |
  5. | C10161005       1    97.24   13.6   4.9      140     6.84 |
     +-------------------------------------------------------------+

. use bl_labs2, clear
. list in 1/5, nolab

     +------------------------------------------------------------+
     |      ptid   regid   creat     hb   pot   sodium   totbil |
     |------------------------------------------------------------|
  1. | C10011001       2     129   12.5   4.9      148       10 |
  2. | C10011002       2     105   14.8   3.9      147       10 |
  3. | C10011004       2      98   13.5   4.7      144       16 |
  4. | C10011005       2     122   13.9   4.9      137       23 |
  5. | C10011006       2      98   13.8   4.7      145        4 |
     +------------------------------------------------------------+

. use bl_labs3, clear
. list in 1/5, nolab

     +---------------------------------------------------------------+
     |      ptid   regid    creat     hb    pot   sodium   totbil |
     |---------------------------------------------------------------|
  1. | C10371001       3    97.24   12.5    4.6      142     6.84 |
  2. | C10371002       3   123.76   13.9   4.49    137.4    23.94 |
  3. | C10371003       3   123.76   15.3    4.4    142.3   30.267 |
  4. | C10371004       3   106.08   11.7    4.5      138   21.033 |
  5. | C10371005       3   114.92   10.9   4.42    136.4   13.167 |
     +---------------------------------------------------------------+
. use bl_labs4
. list in 1/5, nolab

     +-----------------------------------------------------------+
     |      ptid   regid   creat     hb   pot   sodium   totbil |
     |-----------------------------------------------------------|
  1. | C10321001       4      67   15.3   4.5      142        8 |
  2. | C10321002       4     153   10.4   4.6      137        9 |
  3. | C10331001       4      71   13.4   4.5      141        6 |
  4. | C10331002       4     101   15.4     4      138       22 |
  5. | C10331003       4     123   13.1   4.6      142       16 |
     +-----------------------------------------------------------+
```

Note that the variable names in each dataset are identical: *ptid*, *regid*, *creat*, *hb*, *pot*, *sodium* and *totbil*.

Hopefully it should be clear that we want to stack these 4 datasets to produce a longer dataset with the total number of variables still being 7 and the total number of observations equal to the sum of the observations in the 4 datasets.

To combine these datasets we can load one dataset into memory and then append the other datasets e.g.

```
. use bl_labs1 , clear
. append using bl_labs2 bl_labs3 bl_labs4
```

Alternatively we can clear any data from memory and then specify all the datasets to be appended on the append command, e.g.

```
. clear
. append using bl_labs1 bl_labs2 bl_labs3 bl_labs4
```

If required we can generate a variable that indicates which dataset the observation came from we can use the `gen(newvarname)` option, e.g.

```
. append using bl_labs1 bl_labs2 bl_labs3 bl_labs4, gen(datasetid)
```

This will generate a new variable called datasetid in the combined dataset which will take values 1 for all observations from the first dataset in our list (bl_labs1), 2 for all observations from the second dataset in the list (bl_labs2) and so on.

In our example above this is not necessary as there is already a variable, *regid*, that identifies which dataset the observation comes from.

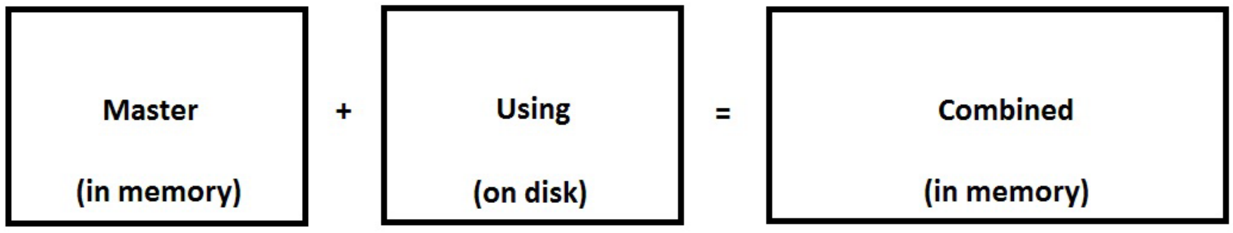
Having done used the append command we should check that things have worked out as we expected (e.g. browse the dataset and do a tabulation of regid) and then, if it has worked, save the new dataset with an appropriate name, e.g.

```
. save bl_labsall
```

The `append` command can also be accessed via the *Data > Combine datasets* menu.

4.8 Merging Stata Datasets

With the `merge` command we are able to add to the dataset in memory (called the master dataset) a second dataset (called the using dataset) which contains additional variables. When combining datasets in this way we do not want to stack them but rather add them on to the 'right-hand side' of the 'master' dataset. This time we end up with a wider dataset containing more variables as shown in the figure below (for one-to-one merges).



The number of observations in the merged dataset may be different if the master and using datasets are not perfectly matched or if we are not doing a one-to-one merge (more later).

Example of a one-to-one merge

Below we have listed the values for some variables in the first five rows of *bl_demog.dta* and for the first five rows of *fup_endpoints.dta* (follow-up dataset for the primary endpoint).

```
. use bl_demog

. list ptid age sex in 1/5

     +-------------------------+
     |      ptid   age     sex |
     |-------------------------|
  1. | C10011001    63    Male |
  2. | C10011002    66    Male |
  3. | C10011004    62    Male |
  4. | C10011005    66    Male |
  5. | C10011006    68    Male |
     +-------------------------+
. use fup_endpoints
. list in 1/5

     +------------------------------------------------------------+
     |      ptid   rcode   pep      pep_dt   acdeath     acdth_dt |
     |------------------------------------------------------------|
  1. | C10011001    2073     1   26nov2007         0    15jan2009 |
  2. | C10011002    2074     1   23mar2009         0    08sep2010 |
  3. | C10011004    2076     0   02sep2010         0    02sep2010 |
  4. | C10011005    2077     1   06apr2010         0    09aug2010 |
  5. | C10011006    2078     0   13sep2010         0    13sep2010 |
     +------------------------------------------------------------+
```

When merging datasets it is important that we have an identification variable (or variables) which link the observations in each dataset. We can see above that the variable *ptid* appears in both datasets. This is the patient identifier which we will use to link the records in the two datasets. Other than *ptid* the two datasets contain different variables. The linking variable name should be identical in each dataset. It cannot be *ptid* in one and *patid* in another.
To merge the two data files we first load one of them into Stata - this is called the **master** dataset in Stata terminology. In this case it makes sense to start with the baseline data file. We then use the `merge` command to add on or merge the second data file – called the **using** dataset in Stata terminology.

On the command line, following the command name, we need to specify the type of merge. In this case we are carrying out a one-to-one merge i.e. there is one unique record for each subject in both the master and using data files. We indicate a one-to-one merge by `1:1`.
Next on the command line we specify the linking variable (or sometimes variables). In our case the variable *ptid* uniquely identifies the records that are to be linked.

Finally we indicate where the file to be merged is located with `using` *`filename`* which might just be the filename if it is located in the current working directory or the directory and filename if located elsewhere.

So to merge *bl_demog.dta* and *fup_endpoints.dta* the syntax would be:

```
. use bl_demog , clear

. merge 1:1 ptid using fup_endpoints

    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                             2,500  (_merge==3)
    -----------------------------------------
```

Each time two files are merged Stata automatically creates a new variable which is named ***_merge***. This variable can take values 1, 2 or 3, indicating 1 – the observation was in the master dataset but was not found in the using data; 2 – the observation was found in the using data but was not found in the master data; 3 – the observation was found in both master and using data. It is possible to carry out more advanced merges involving updating variables, in which case *_merge* can take values 4 or 5.

Immediately after the merge command Stata presents a tabulation of *_merge* (see output above) so we can see how well the matching has gone. In our example above we see that we have complete matching for all 2500 observations. The *_merge* variable remains in the dataset which means that it can be tabulated later. We have to rename or drop the variable before carrying any more merges. If we don't we will get an error message saying that *_merge* is already defined.

Alternatively we can use the `gen(varname)` option with the merge command by which we provide the name for the *_merge* variable e.g.

```
. merge 1:1 ptid using fup_endpoints , gen(pe_merge)
```

This will generate a new variable called *pe_merge* instead of *_merge*.

As the matching was complete we could simply drop the *_merge* variable from our dataset using the `drop` command:

```
. drop _merge
```

Or if would prefer to keep but rename the variable we can use the `rename` command. The command syntax is `rename` ***oldvarname newvarname.*** E.g.

```
. rename _merge pe_merge
```

Note that following the merge command we now have the two files *bl_demog.dta* and *fup_pe.dta* exactly as they were on disk and the combined file in memory. We can save the dataset in memory under a new name or continue adding more data files.

Non-Unique Merges

Here we will look at merging where the linking variable is not unique, either in the master or using datasets or both. Recall that when carrying out a merge the dataset currently in memory is called the master dataset and the dataset being merged on to the dataset in memory is called the using dataset.

There are 3 possible scenarios:

(i) The linking id variable(s) is unique in the master dataset but not in the using dataset. This is called a ***one-to-many*** (1:m) merge,
(ii) The linking id variable is duplicated in the master dataset but is unique in the using dataset. This is called a ***many-to-one*** (m:1) merge,
(iii) Or the linking id variable is duplicated in both variables. This is a ***many-to-many*** (m:m) merge.

Here we will demonstrate the second of these, a many-to-one merge.

Merging part of a Dataset

We will also look at how to merge only part of a dataset, i.e. bring in a subset of the variables from the using dataset. Quite often, when carrying out a merge, we may only wish to bring in a in a subset of variables from the 'using' dataset i.e. not the whole dataset. We can do this easily in Stata with the `keepusing(varlist)` option.

Below are listed the first 5 observations from the *fup_egfr.dta* dataset having sorted on the patient id variable and visit.

```
. use fup_egfr, clear
. sort ptid visit
 . list in 1/5

     +-------------------------------------------+
     |      ptid       visit     visdate    egfr |
     |-------------------------------------------|
  1. | C10011001   Screening   20jul2006   52.82 |
  2. | C10011001     Month 5   19dec2006   58.34 |
  3. | C10011001    Month 13   05sep2007   56.37 |
  4. | C10011001    Month 21   30apr2008   53.67 |
  5. | C10011001    Month 29   08jan2009   56.92 |
     +-------------------------------------------+
```

We can see that the values of the linking variable *ptid* are duplicated e.g. C10011001 appears 5 times, once for each visit.

We want to add on to this dataset the variables sex, age and treatment group from *combined1.dta*. Below we list the first 5 observations from *combined1.dta* to show that *ptid* is not duplicated, e.g. C10011001 appears only on the first row.

```
. use combined1, clear
. sort ptid
. list ptid age sex trt in 1/5

     +----------------------------------+
     |      ptid   age    sex       trt |
     |----------------------------------|
  1. | C10011001    63   Male    Active |
  2. | C10011002    66   Male   Placebo |
  3. | C10011004    62   Male    Active |
  4. | C10011005    66   Male    Active |
  5. | C10011006    68   Male    Active |
     +----------------------------------+
```

To merge these two files we first load fup_egfr.dta and then merge on *hfs_combined1.dta*.

```
. use fup_egfr, clear
. merge m:1 ptid using combined1, keepusing(sex age trt)

    Result                           # of obs.
    -----------------------------------------
    not matched                             3
        from master                         0  (_merge==1)
        from using                          3  (_merge==2)

    matched                             8,383  (_merge==3)
    -----------------------------------------
```

The merge command includes the following:

m:1	indicating a many-to-one merge
ptid	the linking variable
using combined1	indicating which dataset we are merging; the using file is in the current working directory otherwise we would also need to specify the directory
keepusing(sex age trt)	bring in only these three variables

Save this dataset as *fup_egfr1.dta*.

Chapter 5: Housekeeping

Aims and Objectives of Chapter 5

By the end of this chapter you should:

- Be aware of the value of housekeeping
- Know how to attach labels to variables
- Know how to define value labels and attach them to variable values
- Be able to amend value labels
- Know how to rename variable names
- Know how to get rid of unwanted variables or observations
- Know how to add and manage notes to variables
- Know how to manage labels, names and notes in the Variables Manager

The following commands are used in this chapter: `label variable, label define, label values, label list, label book, rename, drop, keep.`

5.1 Housekeeping

By housekeeping we mean the part of data management that is related to keeping your dataset in good order, i.e. not cluttered with redundant variables, clearly labelled, etc. This can be incredibly helpful for you when it comes to descriptive statistics and statistical analyses. A messy dataset often leads to messy analyses.

Additionally, as medical research is often collaborative, it is likely that at some point you will pass on your data to other people. If the dataset is clean, clearly labelled etc. this makes this much less painful than it might otherwise be.

In this chapter we will look at a few of the essential housekeeping commands. There are many more. Much of what we will look at can also be done through the *Variables Manager*.

5.2 Labelling Variables

It is good practice to keep *variable names* as short as possible. This helps to keep the dataset simple, and simplifies the process of referring to variables during analyses, e.g. it is much simpler to refer to *`sbp`* than to *`systolic_blood_pressure`*.

However, keeping variable names short can make it difficult to identify what each variable is e.g. what is *wc* or *lvef*? To overcome this we can attach a lengthier more detailed label (up to 80 characters in length) to each variable. These will be saved with the data file and will appear when describing, tabulating and graphing the variables. For example, when we do a describe of the *bl_demog.dta* dataset:

```
describe

Contains data from H:\Stats Computing\Stata\Exercise 5\bl_demog.dta

  obs:         2,500
 vars:            21                          14 Sep 2015 16:09
 size:       250,000
-------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-------------------------------------------------------------------
ptid            str9    %9s                   Patient ID
birthdt         float   %d                    Date of birth
age             byte    %8.0g                 Age (years)
agegroup        float   %9.0g      agegroup   Age-group
sex             byte    %8.0g      sex        Sex
randdt          float   %d                    Date randomized
trt             str7    %9s                   Treatment allocation
smkstat         long    %13.0g                Smoking status
smoke           str7    %9s                   Smoking status
race            str5    %9s                   Race
hfdiag          byte    %13.0g                Heart failure
diagnosis
wt              double  %10.0g                Weight (kg)
ht              double  %10.0g                Height (cm)
wc              double  %10.0g                Waist circumference
wc_unit         str2    %9s                   WC units
sbp             float   %9.0g
dbp             float   %9.0g
hrate           int     %8.0g
egfr            double  %10.0g                eGFR (ml/min/1.73msq)
lvef            double  %10.0g                LVEF (%)
diab            byte    %10.0g
-------------------------------------------------------------------
Sorted by:  ptid
```

Here we see that in general the variable names (first column) are much shorter than the variable labels (fifth column). The variable ptid has the label “Patient ID”, the variable wt has the label “Weight (kg)”. The syntax for labelling a variable is:

```
label variable varname “variable label”
```

Here `label` is the command and `variable` is a subcommand indicating the type of label being defined. The word *varname* tells Stata to which variable the label should be attached.

In the data description above the variable *sbp* has not been labelled. To add a variable label the syntax is:

```
. label variable sbp “Systolic Blood Pressure (mmHg) ”
```

The variable label will now appear when we describe the data, plot the data, etc. E.g.

```
. describe sbp

              storage   display  value
variable name   type    format   label  variable label
--------------------------------------------------------------
sbp             float   %9.0g           Systolic Blood Pressure (mmHg)
```

Variable labels can be amended without specifying modify, replace or overwrite. For example, if we wished to abbreviate the above label we could amend our command and rerun as

```
. label variable sbp "SBP (mmHg)"
```

5.3 Labelling Values

Above we added labels to variable names. Here we will learn how to add labels to the values of a variable. Stata calls these *value labels*.

Often a categorical variable (e.g. sex) will be recorded as real numeric values (e.g. 1, 2) which indicate some category (e.g. Male, Female). For example, in our dataset the variable *hfdiag* records the type of heart failure which the patient was diagnosed with on entry to the study and takes the values 1 or 2 where 1 = Ischaemic and 2 = Non-Ischaemic.

```
. tab hfdiag

      Heart |
    failure |
  diagnosis |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |      1,730       69.20       69.20
          2 |        770       30.80      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

It can be very useful to attach labels to these values so that we can easily identify what the values refer to without continually having to look this information up.

Adding value labels to a variable in Stata is a two-stage process. Firstly, a value label is *defined*, and then secondly, the value label is *attached* to the appropriate variable.

(i) Defining a Value Label

Each value label has a name, and a series of associations, between integer values and text. To define a value label we use the `label define` command. We will start by defining a value label called *hf_lab*

```
. label define hf_lab 1 "Ischaemic" 2 "Non ischaemic"
```

The words `label define` are the Stata command for defining a value label, *hf_lab* is the name we have decided to call this value label. The rest are the values and the labels for each value.

When we run this command the value label is created or defined but is not yet attached to any variable. When the dataset is saved the value label would be saved as an unattached value label along with the data.

(ii) Attaching a Value Label to a Variable

Having defined the value label we now need to tell Stata to which variable this value label should be attached.

To attach the value label *hf_lab*, created above, to the variable hfdiag:

```
. label values hfdiag hf_lab
```

Here *label value* is the Stata command, *hfdiag* is the name of the variable we are attaching the value label to, and *hf_lab* is the name of the value label we defined above.

```
. tab hfdiag

Heart failure |
    diagnosis |      Freq.     Percent        Cum.
--------------+-----------------------------------
    Ischaemic |      1,730       69.20       69.20
Non-ischaemic |        770       30.80      100.00
--------------+-----------------------------------
        Total |      2,500      100.00
```

Amending or replacing a value label

To change or modify an existing value label we need to specify the option `modify` along with the modified label define command. For example, if we wanted to modify the labels of the hf_lab value label,

```
. label define hf_la 1 "Ischaemic HF" 2 "Non ischaemic HF", modify
```

We can completely overwrite a value label by specifying the option `replace`.

Dropping a Value Label

To drop an existing value label:

```
. label drop labelname
```

Where `label drop` is the Stata command and *`labelname`* is the name of the value label.

Detaching a Value Label

To detach a value label from a particular variable, without deleting the value label itself, we simply use the label values command specifying the variable name but no value label. For example,

```
. label values agegroup
```

Will break the association between agegroup and its current value label, but the value label itself will not be deleted.

Getting a list of value labels

There are two useful commands for getting summaries of one, several or all the value labels that exist in the dataset in memory.

The command `label list [labelname]` can be used to obtain a list of any value labels defined, including the value label name, the values and what they map to. E.g.

```
. use bl_demog , clear

. label list
agegroup:
           0 30-
           1 65-
           2 70-
           3 75-
           4 100-
sex:
           1 Male
           2 Female
```

The command `labelbook [labelname]` can be used to obtain a codebook style report for the specified value label or, if none are specified, for all value labels in the dataset.

```
. labelbook agegroup

-------------------------------------------------------------------
value label agegroup
-------------------------------------------------------------------

      values                                    labels
       range:  [0,4]                     string length:  [3,4]
           N:  5                 unique at full length:  yes
        gaps:  no                  unique at length 12:  yes
  missing .*:  0                           null string:  no
                               leading/trailing blanks:  no
                                    numeric -> numeric:  no
  definition
           0   30-
           1   65-
           2   70-
           3   75-
           4   100-

   variables:  agegroup
```

Adding or Removing Numeric Prefixes to Value Labels

The command `numlabel` can be used to add (or remove) prefix numeric values to the value labels. The syntax for the command is:

```
numlabel [labelname],  {add/remove}
```

For example:

```
 . tab sex

         Sex |      Freq.     Percent        Cum.
------------+-----------------------------------
        Male |      1,942       77.68       77.68
      Female |        558       22.32      100.00
------------+-----------------------------------
       Total |      2,500      100.00

 . numlabel sex, add
 . tab sex

         Sex |      Freq.     Percent        Cum.
------------+-----------------------------------
     1. Male |      1,942       77.68       77.68
   2. Female |        558       22.32      100.00
------------+-----------------------------------
       Total |      2,500      100.00
 . numlabel sex  , remove
 . tab sex

         Sex |      Freq.     Percent        Cum.
------------+-----------------------------------
        Male |      1,942       77.68       77.68
      Female |        558       22.32      100.00
------------+-----------------------------------
       Total |      2,500      100.00
```

Saving value labels to a do-file

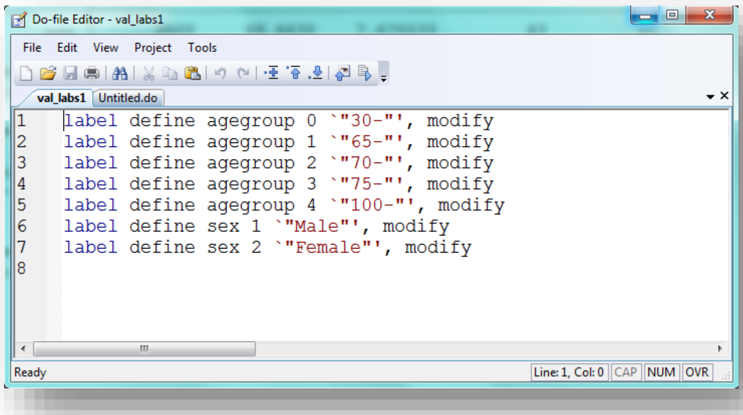
The command `label save` can be used to save value labels to a do-file i.e. to save a series of commands that can redefine the value labels. Value labels currently not attached to at least one variable are not saved. The file to which the value labels are to be saved is specified with using filename. The syntax for the command is:

```
  label save using filename, [options]
```

For example to save the value labels from the bl_demog dataset:

```
 . label save using val_labs1
 file val_labs1.do saved
```

The do-file created is shown below.



Do-file Editor - val_labs1
File Edit View Project Tools
val_labs1 Untitled.do
Ready
Line: 1, Col: 0 CAP NUM OVR

```
label define agegroup 0 `"30-"', modify
label define agegroup 1 `"65-"', modify
label define agegroup 2 `"70-"', modify
label define agegroup 3 `"75-"', modify
label define agegroup 4 `"100-"', modify
label define sex 1 `"Male"', modify
label define sex 2 `"Female"', modify
```

Figure 5.1: Do-file created by label save

5.4 Dropping Variables

Sometimes it can be helpful to reduce the size of your dataset by dropping either certain variables (columns) or observations (rows) from a dataset. For example, one or more variables that have been collected may not be required or perhaps an error has been made when generating a new variable. Alternatively you may want to drop all subjects that are under a certain age. We can use the command `drop` for either case.

For example, if you have made a mistake when creating a variable called *smoke2* and you wish to recreate it correctly, first you will need to drop the erroneous variable

```
. drop smoke2
```

In the situation where the number of variables you want to drop is greater than the number that will be left in the dataset we can use the `keep` command. This does the opposite to the `drop` command in that it keeps the variables specified and drops all others. For example:

```
describe

Contains data from bl_demog.dta
  obs:         2,500
 vars:            18                          23 Sep 2014 14:15
 size:       232,500
---------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
---------------------------------------------------------------------------
ptid            str9    %9s                   Patient ID
birthdt         str12   %12s
age             byte    %8.0g
agegroup        float   %9.0g      agegroup
sex             byte    %8.0g      sex        Sex
smkstat         long    %13.0g                Smoking status
race            str5    %9s                   Race
hfdiag          byte    %13.0g                Heart failure diagnosis
wt              double  %10.0g                Weight (kg)
ht              double  %10.0g                Height (cm)
wc              double  %10.0g                Waist circumference
wc_unit         str2    %9s                   WC units
sbp             float   %9.0g
dbp             float   %9.0g
hrate           int     %8.0g
egfr            double  %10.0g                eGFR (ml/min/1.73msq)
lvef            str5    %9s                   LVEF (%)
diab            str7    %9s
---------------------------------------------------------------------------
Sorted by:  lvef

keep ptid age sex
describe

Contains data from bl_demog.dta
  obs:         2,500
 vars:             3                          2 Sep 2014 19:20
 size:        27,500
-----------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
-----------------------------------------------------------------------
ptid            str9    %9s                   Patient ID
age             byte    %8.0g
sex             byte    %9.0g      sex        Sex
-----------------------------------------------------------------------
Sorted by:  ptid
     Note:  dataset has changed since last saved
```

5.5 Dropping Observations

You may wish to carry out a series of analyses which are restricted to just a subset of all the observations in a dataset. You could do this using an `if` expression and keeping all the observations in the dataset – but this can add to the complexity of an analysis. An alternative is to create a separate dataset that contains just the observations that you want to include in your analysis. We can do this with the `drop` or `keep` commands combined with an `if` expression.

For example if you wanted to carry out a series of analyses on people aged 50 or above we could drop all observations/records where age was less than 60 years as follows.

```
. use bl_demog, clear

. summ age

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         age |       2500     68.6632    7.675033         43         95

. drop if age<60
(272 observations deleted)

. summ age

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         age |       2228    70.10727    6.819218         60         95
```

This command has dropped the entire record (row) for any person age under 60 years. We could have done the same thing using `keep if age>=60`.

There is not an undrop or unkeep command so take care. However, remember any changes you make such as dropping variables or observations are only made to the dataset in memory and not to the data on disk.

5.6 Changing Variable Names

Sometimes we may wish to modify variable names. We can do this using the `rename` command.

The basic syntax is:

```
  rename oldname newname
```

Where *oldname* is the original name of the variable and *newname* is the new variable name. For example, to change the variable name *diab* to *diabetes*:

```
rename diab diabetes
```

We can also rename groups of variables. Some examples are given below.

1. To rename two variables at same time (birthdt as dob and randdt as dor):

```
  rename (birthdt randdt) (dob dor)
```

2. To swap two variable names:

```
rename (a b) (b a)
```

3. To rename all variables beginning/ending with the same prefix/suffix:

(i) To rename all variables beginning with bl_ by removing bl_ and adding _1 to end of name:

```
rename bl_* *_1
```

(ii) To remove the prefix bl_ from all variable names:

```
rename bl_* *
```

(iii) To remove the suffix _fup and replace with _2

```
rename *_fup *_2
```

We can also use `rename` to change the case of variable names. As Stata is case sensitive with variable names it is generally simplest to have all variable names in lower case. The syntax for this is:

```
rename varname, lower
```

Where *varname* is the name of the variable to be changed. We could also specifiy `upper` or `proper` for upper and proper case respectively – though this is not advised for variable names.

Sometimes many or all of the variable names in a dataset contained upper case characters and we wish to make them all lower case. We can do this simply using _all to indicate the variable list. E.g.

```
rename _all, lower
```

5.7 Adding Notes

The command `notes` attaches notes to the dataset in memory. As with labels these notes become a part of the dataset and are saved when the data file is saved. Also as with labels we can attach notes generically to the dataset or to specific variables within the dataset.

The note facilities can be accessed using the *Data > Data utilities > Notes utilities* menu.

Notes are numbered sequentially, with the first note added being 1, the second being 2 etc. When one note is deleted the numbering is not reordered. Up to 9,999 notes can be added to the dataset generically, and up to 9,999 notes added to each variable! Each note can be up to 67,784 characters in length! Note that notes are not saved until the dataset is saved.

Some examples are given below:

1. Adding a note to the dataset in memory

```
. notes: Data used for Intro to Stata Module
```

2. Adding a note to a variable

```
. notes wc: Note that some waist circumferences are measured in
meters
```

3. Listing notes

```
. notes

_dta:
  1.  Data used for Intro to Stata Module

wc:
  1.  Note that some waist circumferences are measured in meters
```

Just specifying the command `notes` requests a listing of all notes. To specify a list of notes for the dataset

```
. notes _dta

_dta:
  1.  Data used for Intro to Stata Module
```

And for a particular variable:

```
. notes wc

wc:
  1.  Note that some waist circumferences appear to be measured in
meters
```

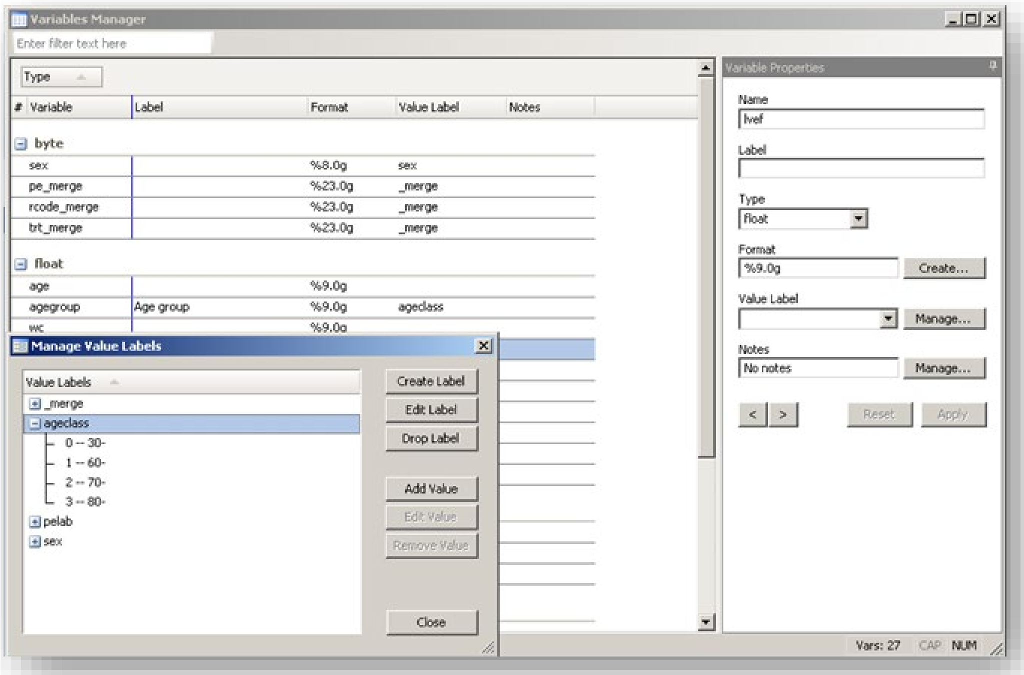
4. Deleting notes

Notes can be deleted using the notes drop command. For example, to drop the first note attached to wc

```
. notes drop  wc in 1
  (1 note dropped)
```

5.8 Managing Labels and Notes in the Variables Manager

If you forget the command syntax for labelling, notes etc, then variable labels, value labels and notes can be managed very easily through the Variables Manager (Figure 7.2) . Remember that the Variables Manager can be opened using the *Window > Variables Manager* menu or through the shortcut on the toolbar.



# Variable	Label	Format	Value Label	Notes
byte				
sex		%8.0g	sex	
pe_merge		%23.0g	_merge	
rcode_merge		%23.0g	_merge	
trt_merge		%23.0g	_merge	
float				
age		%9.0g		
agegroup	Age group	%9.0g	ageclass	
wc		%9.0g		

Name: lvef
Label:
Type: float
Format: %9.0g
Value Label:
Notes: No notes

```
_merge
ageclass
  0 -- 30-
  1 -- 60-
  2 -- 70-
  3 -- 80-
pelab
sex
```

Figure 5.2 Variables Manager and Manage Value Labels dialog box

To add a variable label, selecting the variable from the left frame and fill in the details in Variable properties frame. Click on the Manage... button next to the Value Label field to open the Manage Value Labels dialog box. With this it is possible to create, edit, drop, and attach value labels. Notes are dealt with similarly.

Chapter 6: Essential Data Processing

Aims and Objectives of Chapter 6

By the end of this chapter you should:

- Know how Stata deals with data in memory and on disk
- Be able to correct or change values of existing variables
- Be able to convert string variables to numeric
- Understand how Stata stores missing values
- Be able to create new variables

The following commands are used in this chapter: `replace`, `encode`, `destring`, `recode`, `generate`, `egen` and `mvdecode`.

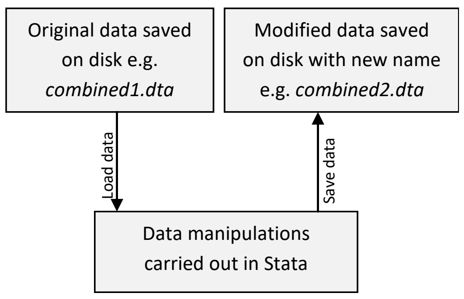
6.1 Data Processing

So far in this module we have learnt how to create Stata datasets, to examine data for errors and to carry out basic housekeeping. The goal of this process is to create a dataset that is ready for analysis. In this chapter we will continue along this process and look at some of the essential data processing commands for correcting errors and creating new variables ready for our analysis.

6.2 Processing Existing Variables

It is often the case that we will want to modify some existing variables. For example, we may wish to change the units (e.g. from days to years or grams to kilograms), or correct errors, or convert from string to numeric. Here we will look at a few of the most essential commands you will need: `replace`, `encode`, `destring` and `recode`.

It is important to recall that when we make changes e.g. drop variables, rename variables, generate new variables we are only affecting the data currently in memory, not the data saved on disk. The modifications are only saved when the dataset is saved. It is important to keep one completely unmodified copy of your original or raw data.


Original data saved on disk e.g. *combined1.dta*
Modified data saved on disk with new name e.g. *combined2.dta*
Load data
Save data
Data manipulations carried out in Stata

1. Replacing values

The **`replace`** command can be used to change the values of an existing variable. The syntax for the command is:

```
  replace varname=expression [if] [in]
```

where *expression* could be a number or a missing value, or a function of some variables.

Replace example

The variable *wc* has values that were recorded in metres or centimetres. We want the units to be consistent e.g. to create a new variable where all the values are in cm. First we will generate a new variable, wc_cm, which is an exact copy of wc, using the `generate` command (we will come back to this command shortly) and then replace the values that were measured in metres.

```
. generate wc_cm = wc
. replace wc_cm = wc_cm*100 if wc_unit==”M”
(319 real changes made)
```

We have replaced *wc_cm* with itself times 100 if *wc_unit* equals “M”. Note use of single equals for assigning the values and the double equals for if condition.

We should now check that the changes have worked.

```
. summ wc wc_cm

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          wc |      2369    85.88955    35.65602        .69        165
       wc_cm |      2369    99.27214    13.28178         58        165
```

2. Encoding Categorical String Variables

Most of Stata's analytical commands require numeric rather than string variables. When strings are encountered Stata treats them as if they were missing values. By a categorical string variable we mean variables such as sex containing values “Male” or “Female” or smoke containing “Never”, “Ex” or “Current” or race containing “Asian”, “Black”, “White” etc.

We can convert such categorical string variables into numeric variables using the `encode` command. This command allocates numeric values to the different strings (default is in alphabetical order). A value label is automatically created.

The basic syntax for the command is:

```
    encode varname , gen(newvarname)
```

where *varname* is the name of the variable to be encoded; `gen(newvarname)` must be specified.

Encode example 1

The variable *sex* is a string variable recorded as “Female” or “Male”. To encode this into a numeric variable with value labels attached:

```
. encode sex , gen(sex2)

. tab sex2

        Sex |      Freq.     Percent        Cum.
------------+-----------------------------------
     Female |        558       22.32       22.32
       Male |      1,942       77.68      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

Specifying the option `nolabel` we see that *sex2* is a numeric variable with value labels attached

```
. tab sex2  , nolab

        Sex |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |        558       22.32       22.32
          2 |      1,942       77.68      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

By default with `encode` the new variable takes values 1, 2, etc. based on the alphabetical ordering of the string values.

Encode example 2

If we wish the variable to take some ordering other than alphabetical we need to first define the *value label* indicating the ordering and then include the option `label(labelname)` with the command. For example, we may wish to encode the variable *race* but due to the numbers in the different categories it makes sense to use an ordering other than alphabetical.

```
. tab race

       Race |      Freq.     Percent        Cum.
------------+-----------------------------------
      Asian |        295       11.80       11.80
      Black |         60        2.40       14.20
      Other |         79        3.16       17.36
      White |      2,066       82.64      100.00
------------+-----------------------------------
      Total |      2,500      100.00

. label define race_lab 1"White"2"Asian"3"Black"4"Other"
. encode race , gen(race2) label(race_lab)
. tab race2

       Race |      Freq.     Percent        Cum.
------------+-----------------------------------
      White |      2,066       82.64       82.64
      Asian |        295       11.80       94.44
      Black |         60        2.40       96.84
      Other |         79        3.16      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

3. Converting “Number” Strings to Numeric

With the `encode` command above we converted categorical string variables into numeric variables. The original categories contained strings which consisted of non-numeric characters e.g. Female, Black. In each case there was a discrete number of categories.

In this example we will be converting numeric variables that have been stored as strings into a numeric storage type. These are variables that may have all numeric characters but have been imported as strings or could be variables that are mostly numeric but have a few non-numeric characters. For these examples we will use the `destring` command. The syntax is:

```
destring varlist , {replace/gen(newvarlist)} [force ignore(“”)]
```

One of the options `replace` or `gen(newvarlist)` must be specified. If `replace` is used the existing variable is overwritten.

Destring example 1

First we will look at the variable lvef (Left ventricular ejection fraction (%)). This is a numeric variable that can in theory take values between 0 and 100. In the bl_demog dataset this variable has been imported as a string.

```
. codebook lvef

-------------------------------------------------------------------------------
lvef                                                                    LVEF (%)
-------------------------------------------------------------------------------

                  type:  string (str5)

         unique values:  80                       missing "":  0/2500

              examples:  "20"
                         "25"
                         "28"
                         "30"
```

We can see from the codebook command that *lvef* is stored as a string, but the examples are just numeric characters. If you browse lvef you will see that the variable contains just numeric characters but that they are displayed in red i.e. they are stored as a string.

We therefore cannot calculate a mean or other summary statistics since Stata does not allow this for string variables e.g. see summarize output below.

```
. summarize lvef

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        lvef |         0
```

To convert this to a numeric variable:

```
. destring lvef , replace
lvef has all characters numeric; replaced as double
(137 missing values generated)
```

Stata reports that all the characters were numeric and that lvef has been replaced as double i.e. as one of Stata's numeric storage types. Now we can use summarize to obtain some summary statistics.

```
. summ lvef

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        lvef |      2363    26.07964    4.671837          6         40
```

Destring example 2

In this second example we have a variable diab that has been recorded as 0 or 1 i.e. as a numeric variable. However, for one observation the entry is recorded as "Missing". This is obviously not a number and therefore the variable *diab* has been stored as a string.

```
. codebook diab

------------------------------------------------------------------
diab                                                   (unlabeled)
------------------------------------------------------------------

                  type:  string (str7)

         unique values:  3                      missing "":  0/2500

            tabulation:  Freq.  Value
                          1716  "0"
                           783  "1"
                             1  "Missing"
```

If we try the same approach as for lvef Stata will report that this variable contains some non-numeric characters and therefore that it cannot replace. A variable can only be numeric if all the characters are numeric. To force Stata to convert this to a numeric variable we either must specify which non-numeric characters to ignore or specify the `force` option.

```
. destring diab , replace
diab contains nonnumeric characters; no replace

. destring diab , replace ignore("missing")
diab: characters m i s n g removed; replaced as byte
(1 missing values generated)

. tab diab

       diab |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      1,716       68.67       68.67
          1 |        783       31.33      100.00
------------+-----------------------------------
      Total |      2,499      100.00
```

4. Recoding numeric categorical variables

We may sometimes wish to combine categories of an existing variable (e.g. combine Current and Ex smokers into Ever smokers or change a binary variable coded as 1=No 2=Yes to be 0=No and 1=Yes).

The most appropriate command for this in Stata is `recode`. The basic syntax for recode is:

```
recode varname rules , gen(newvarname) label(labname)
```

Where `(rules)` takes the form `(oldvalue(s)= newvalue)`. Some examples of rules are shown in the table below.

Recode rules

Example	**Meaning**
(3 = 1)	3 recoded to 1
(4/6 = 2)	4 to 6 (inclusive) recoded to 2
(7 9 = 3)	7 and 9 recoded to 3
(min/2=0)	Minimum value to 2 (inclusive) recoded as 0
(11/max=4)	11 to maximum value (inclusive but not including missing values) recoded to 4
(* = 5)	All other values (including missing) recoded to 5
(. = 99)	Missing values recoded to 99

The option `gen(`*`newvar`*`name)` indicates that we want to create a new variable (*newvarname*) which contains the recoded values i.e. we do not recode the values of the original variable. If this option is not specified then the values of the original variable (*varname*) will be recoded. We should only do this if *varname* is a new variable we have previously copied using `generate`.

Recode example 1

Suppose we wish to create a new variable *evsmk* that takes the value 0 for never smokers and 1 for those who have ever smoked. In our dataset we have a variable *smoke* that takes value 1 = Never, 2 = Ex-Light, 3 = Ex-Heavy, 4 = Current-Light and 5 = Current-Heavy.

```
.recode smoke(1=0"Never smoker")(2/5=1"Ever smoker"),gen(evsmk)
label(evsmk_lab)
(2500 differences between smoke and eversmoke)
```

Note that within the brackets we have specified the recode rules followed by value labels. In the options we have specified the name of the value label with `label(`*evsmk_lab*`)`.

We can check the distribution of the new variable with a table.

```
. tab evsmk

   RECODE of |
       smoke |
    (Smoking |
     status) |      Freq.     Percent        Cum.
-------------+-----------------------------------
Never smoker |      1,114       44.56       44.56
 Ever smoker |      1,386       55.44      100.00
-------------+-----------------------------------
       Total |      2,500      100.00
```

It is worth cross-tabulating the two variables to check we have not made a mistake in the recoding:

```
. tab smoke evsmk

              |   RECODE of smoke
      Smoking |   (Smoking status)
       status | Never smo  Ever smok |     Total
--------------+----------------------+----------
        Never |     1,114          0 |     1,114
     Ex-Light |         0        576 |       576
     Ex-Heavy |         0        541 |       541
Current-Light |         0        132 |       132
Current-Heavy |         0        137 |       137
--------------+----------------------+----------
        Total |     1,114      1,386 |     2,500
```

See `help recode` for more details.

Recode example 2

Here we will create a categorical variable called *agegroup* by recoding the variable age; as above we do not want to change the values of the variable age. The minimum age is 43 years and maximum is 95 years.

```
. recode age (40/59=1 "<60 years" ) ( 60/69=2  "60-69 years" )
(70/max=3  "70+ years" ) , gen(agegroup) label(age_lab)
(2500 differences between age and agegroup)

. tab agegroup

  RECODE of |
   age (Age |
   (years)) |      Freq.     Percent        Cum.
------------+-----------------------------------
  <60 years |        272       10.88       10.88
60-69 years |      1,125       45.00       55.88
  70+ years |      1,103       44.12      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

6.3 Creating New Variables

We will now look at two key commands for creating new variables: `generate` and `egen`.

Generate

The basic syntax for the `generate` command is:

```
generate [type]  newvar = expression [if] [in]
```

`newvar` is the name of the new variable being created;

`expression` could be a number, a string, a variable or some function to be evaluated;

`type` is the storage type for the new variable. Type can be one of:

byte	integer (range from -127 to +100)	1 byte
int	Integer	2 bytes
long	integer (range -2,147,483,647 to 2,147,483,620)	4 bytes
float	$-1.7014117 \times 10^{38}$ to $+1.7014117 \times 10^{38}$	4 bytes
double	$-8.9884656 \times 10^{307}$ to $+8.9884656 \times 10^{307}$	8 bytes
str#	string of maximum length # where $1 \leq \# \leq 244$	1-244 bytes

If `type` is not specified and the new variable is numeric Stata will default to float. If `type` is not specified and the new variable is a string Stata will automatically select *str#* where *#* is the maximum length of the string variable.

Generate example 1: creating a variable that is a function of two variables

Suppose we wish to create a new variable called *bmi* containing the body mass index of each individual (defined as kg/m^2). Note that in our dataset weights are recorded in kilograms but heights are recorded in centimetres rather than metres. We therefore need to divide height by 100 within the expression:

```
. generate bmi = wt/(ht/100)^2
(14 missing values generated)
```

Note that 14 missing values are generated since we have some missing values for *ht* and/or *wt* and therefore *bmi* cannot be calculated.

Once a new variable has been created it is always worthwhile checking its distribution using `histogram` or `summarize`. It is possible that we might uncover problems with *ht* and *wt* that were not obvious when considered individually. There are 2 values of *bmi* over 50 kg/m^2.

```
. list pt wt ht bmi wc sex if bmi>50 & bmi~=.

       +------------------------------------------------------+
       |      ptid       wt     ht         bmi      wc      sex |
       |------------------------------------------------------|
1342.  | C13721007      130    150    57.77778       .   Female |
1941.  | C14791001    149.9    173     50.0852    1.46     Male |
       +------------------------------------------------------+
```

A twoway scatter plot can be helpful. The darker markers relate to the 2 cases where bmi>50.

```
. twoway   (scatter wt ht if bmi>50, ms(o) mc(gs1)) ///
           (scatter wt ht if bmi<50, ms(oh) mc(gs12)), legend(off)
```

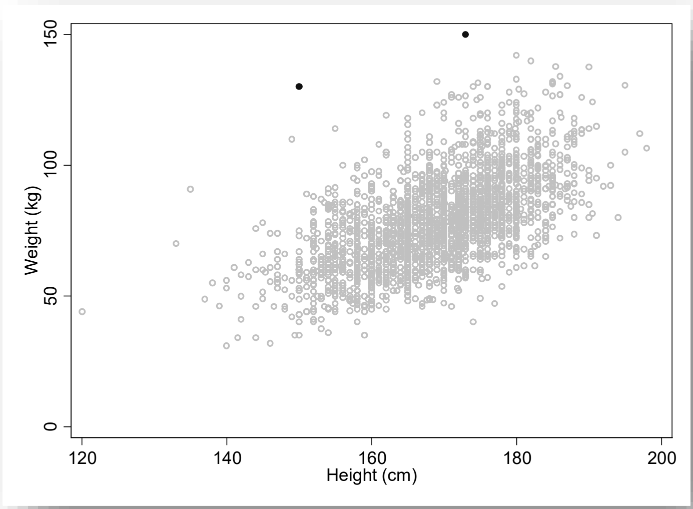


Figure 6.1 Scatter plot of weight versus height

Generate example 2: generating a log-transformed variable

Quite often we will want to create a new variable equal to the log-transformation of another, generally where the original variable is skewed.

```
. gen log_egrf = log(egfr)
```

Creates a new variable called *log_egfr* that is the natural logarithm of *egfr*. We can use `histogram` to compare the two distributions.

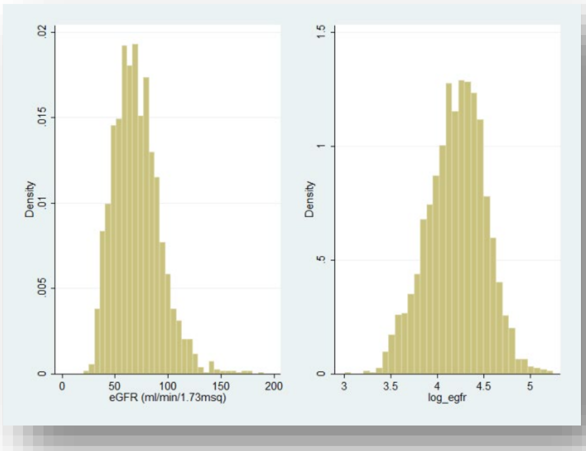


Figure 6.1: Histogram of egfr and log_egfr

The log transformed variable looks slightly less positively skewed though there may be some negative skew.

```
. summ egfr log_egfr, detail

                     eGFR (ml/min/1.73msq)
-------------------------------------------------------------
      Percentiles      Smallest
 1%         32.6           20.2
 5%           39             25
10%         44.5           26.3       Obs                2491
25%         55.2           26.8       Sum of Wgt.        2491

50%         68.7                      Mean           70.70273
                        Largest       Std. Dev.      21.89467
75%         83.8          172.5
90%         98.3          176.2       Variance       479.3766
95%        109.2            178       Skewness       .8156039
99%        137.7          190.5       Kurtosis       4.494325

                           log_egfr
-------------------------------------------------------------
      Percentiles      Smallest
 1%     3.484312       3.005683
 5%     3.663562       3.218876
10%     3.795489       3.269569       Obs                2491
25%     4.010963       3.288402       Sum of Wgt.        2491

50%     4.229749                      Mean            4.21123
                        Largest       Std. Dev.      .3101085
75%     4.428433       5.150397
90%     4.588024        5.17162       Variance       .0961673
95%     4.693181       5.181784       Skewness       -.163848
99%     4.925077       5.249652       Kurtosis       2.964581
```

Generate example 3: generating a binary variable

A useful feature of the `generate` command is that when combined with a condition it is possible to generate a binary 0, 1 variable depending on whether the condition is satisfied or not. For example, to generate a variable called *age70* taking the value 1 if the subject is greater than or equal to 70 and 0 otherwise.

```
. gen age70 = (age>=70)

. tab age70

      age70 |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      1,397       55.88       55.88
          1 |      1,103       44.12      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

CAUTION: Care needs to be taken when using > or < conditions where there are missing values. This is because of the way Stata handles missing values. Stata stores numeric missing values as "large positive values" which will be greater than any non-missing value and which will therefore certainly satisfy the condition "> some real value" and certainly not satisfy the condition "< some real value". So to avoid having a subject with a missing bmi returned as being obese we need to add ***if bmi~=.*** as follows.

```
. gen obese = (bmi>=30) if !missing(bmi)
(14 missing values generated)


. tab obese

      obese |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      1,821       73.25       73.25
          1 |        665       26.75      100.00
------------+-----------------------------------
      Total |      2,486      100.00
```

If we had omitted the !missing(bmi) condition we would have got the following:

```
. gen obese = bmi>=30

. tab obese

      obese |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      1,821       72.84       72.84
          1 |        679       27.16      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

Note that this time Stata does not report any missing values generated and that we have an extra 14 observations in category 1 (the BMI 30+ group).

More than one condition can be specified, e.g.

```
. gen hyper = (sbp>=140 | dbp>=90) if !missing(sbp,dbp)
(1 missing value generated)

. tab hyper

      hyper |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      1,913       76.55       76.55
          1 |        586       23.45      100.00
------------+-----------------------------------
      Total |      2,499      100.00
```

However, you need to think carefully when doing this. If sbp was missing and dbp was 95 then by our definition the person has hypertension no matter what the missing sbp value is. However if sbp was missing and dbp was 89 the missing sbp is important.

Egen

The command `egen` is an extension to the `generate` command and has many useful functions which can be used to create a new variables.

A particularly useful function specific to `egen` is `cut(varname)` which can be used to 'cut up' or categorize a continuous variable. We can either specify the exact cut-points to be used with the option `at(`*`numlist`*`)` or specify the desired number of equal-sized groups (e.g. quarters or fifths) using the option `group(#)`.

Egen example 1: Categorising a continuous variable at specific cut-points

Here we will create a new variable *sbpcat* based upon an existing variable *sbp*. We will create four categories: <120, 121-129, 130-139 and 140+. The lowest value we will accept is 80mmHg and maximum value will be 200mmHg i.e. any values below 80 or above 200 will be treated as missing. The command syntax for this is:

```
. egen sbpcat = cut(sbp), at(80, 120, 130, 140, 201) label
```

The option `at()` tells Stata where we want the cut-points. The first number is the lowest value of the first category; the next value is the lowest value in the second category and so on up to the final number which is the value at and beyond which we will treat as missing. So in our example here the first category will extend from 80 to 119 mmHg inclusive; values below 80 would be returned as missing. The second category will include values from 120 to 129 mmHg and the third 130 to 139 mmHg. The fourth category will extend from 140 to 200 mmHg and any values of 201 or above will be returned as missing.

The option `label` requests that the categories are given integer-coded values and that a value label (using the left-hand ends of each interval) is automatically defined and attached. If the label option is not used then the by default the categories will be coded with values equal to the left-hand end of the intervals e.g. 80, 120 etc. in the example above.

```
. tab sbpcat

     sbpcat |      Freq.     Percent        Cum.
------------+-----------------------------------
        80- |        880       35.21       35.21
       120- |        581       23.25       58.46
       130- |        521       20.85       79.31
       140- |        517       20.69      100.00
------------+-----------------------------------
      Total |      2,499      100.00
```

Note that one missing value was generated because we have one value missing for *sbp*.

The number list in the `at()` option could also be specified as #1(#2)#3 meaning from #1 to #3 in steps of #2. For example, `at(40(10)90)` would be create categories from 40 to 90 in steps of 10 where the top category would be 80-89.

Egen example 2: Categorising a continuous variable into equal sized groups

To categorize a variable into equal sized groups (e.g. quarters, fifths) we use the option `group(#)` rather than specifying the actual cut-points. For example, to categorise *bmi* into four equal size groups:

```
. egen bmi4 = cut(bmi), group(4) label
 (14 missing values generated)


. tab bmi4

       bmi4 |      Freq.     Percent        Cum.
------------+-----------------------------------
  13.21179- |        621       24.98       24.98
   24.4418- |        620       24.94       49.92
   27.0538- |        623       25.06       74.98
  30.27371- |        622       25.02      100.00
------------+-----------------------------------
      Total |      2,486      100.00
```

Note that the groups are not exactly equal-sized because there are a few tied values.

Egen example 3: Creating a row summary variable

In the bl_medhis1.dta dataset there are four binary (0/1) variables (*strisch strhem stremb stroth*) that record whether or not the patient had previously experienced any of four different types of strokes. The four variables are tabulated below.

```
. tab1 strisch strhem stremb stroth

    strisch |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,297       92.47       92.47
          1 |        187        7.53      100.00
------------+-----------------------------------
      Total |      2,484      100.00


     strhem |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,413       99.38       99.38
          1 |         15        0.62      100.00
------------+-----------------------------------
      Total |      2,428      100.00


     stremb |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,397       98.56       98.56
          1 |         35        1.44      100.00
------------+-----------------------------------
      Total |      2,432      100.00

     stroth |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,404       99.05       99.05
          1 |         23        0.95      100.00
------------+-----------------------------------
      Total |      2,427      100.00
```

We want to create a single variable that takes the value 1 if any of these four variables is 1 and 0 otherwise. As a patient could have had more than one type of stroke a row total would not necessarily work. What we could use for this task is a row maximum.

```
. egen stroke_any = rowmax(strisch strhem stremb stroth)
(1 missing value generated)


. tab stroke

 stroke_any |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,249       90.00       90.00
          1 |        250       10.00      100.00
------------+-----------------------------------
      Total |      2,499      100.00
```

1 missing value has been generated. This will be a patient who has missing values for all four of the stroke types.

Egen example 4: calculating the number of missing values in a set of variables

In example 3 we saw that 1 missing value had been generated. This was for a patient who had missing values for all four of the stroke types. However, you may have noted from the tables above that each of the stroke type variables has more than one missing values. The `rowmax` function will calculate a row maximum as long as at least one value is not missing. The missing values do not count towards the row maximum. Essentially we have to assume here that if one stroke type has been reported then all other missing values can be treated as a 0. We might be interested in exploring how serious this issue might be by finding out how many missing values each person had.

To calculate the number of missing values among these four stroke variables for each patient we can use the `rowmiss()` function of `egen`.

```
. egen stroke_miss = rowmiss(strisch strhem stremb stroth)

. tab stroke_miss

stroke_miss |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      2,418       96.72       96.72
          1 |          9        0.36       97.08
          3 |         72        2.88       99.96
          4 |          1        0.04      100.00
------------+-----------------------------------
      Total |      2,500      100.00
```

Here we see that 1 patient has all four missing, 72 have three missing and 9 have one missing value. The majority have all 4 values present.

We could explore further by creating a cross-tabulation of *stroke_any* and *stroke_miss* to see how many of the 81 with partially missing data had previously had a stroke.

```
. tab stroke_miss stroke_any, miss

stroke_mis |            stroke_any
         s |         0          1          . |     Total
-----------+---------------------------------+----------
         0 |     2,226        192          0 |     2,418
         1 |         8          1          0 |         9
         3 |        15         57          0 |        72
         4 |         0          0          1 |         1
-----------+---------------------------------+----------
     Total |     2,249        250          1 |     2,500
```

We can see that 58 out of the 81 with some missing value had previously experienced a stroke. The missing data is therefore not important – since whatever values they would take the row maximum would still be 1. There are 23 patients (8+15) who we are uncertain about. We must either assume that missing is equivalent to “no” for these patients or perhaps code them as missing for the *stroke_any* variable.

6.4 Missing values

Missing data are common in medical research. It is rare to find a dataset with complete data on all subjects on all variables, even from the most carefully conducted trials.

Stata has 27 numeric missing values. They are,

.   called the "system missing value",

and

.a, .b, .c, ... , .z   called "extended missing values".

The "system missing value" is generated by Stata when it is not able to assign a specific value. The "extended missing values" can be specified by the user when it is important to keep track of reasons for a missing value. Value labels can be attached to .a, .b etc. The advantage of having more than one missing value code is that measurements that are missing because of say non-response can be distinguished from those missing due to withdrawal from the study.

As described above, numeric missing values are represented by very large positive values. The ordering is:

"all non-missing numbers" < . < .a < .b < .c < ... < .x < .y < .z

Stata ignores any of these missing values when carrying out analytical commands e.g. when summarizing a variable. However, it *is very important* to note that sometimes missing values can be recorded in a database as a real number e.g. 9999 or -999. When this is the case Stata will treat these as real values and so they would, for example, contribute towards the calculation of a mean. Such values need to be recoded as Stata missing values. We can do this with the `recode` or `mvdecode` commands.

Missing Values Example 1: recoding numeric values to missing

First look at a summary of hemaglobin (hb).

```
. summm hb

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
          hb |       2500    425.1898    1984.981        8.8       9999
```

We see that the maximum value is 9999 which was the numeric value chosen to indicate a missing value. Although we know that this is not a real value Stata does not and had treated it as a genuine value and used it in calculating the mean and standard deviation (SD). Such values need to be recoded as missing values. We can do this is a number of ways, including using `replace`, `recode` or `mvdecode`.

For example to use the `recode` command:

```
. recode hb 9999=.
(hb: 103 changes made)
```

If we do a summary now we see a big change in both the mean and SD.

```
. summm hb

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          hb |      2397    13.79959    1.560654        8.8         20
```

Missing values Example 2: recoding missing values for multiple variables

Alternatively Stata has a specific command (`mvdecode`) for decoding missing values across multiple variables and if required multiple variables.

Here we just have one numerical missing value (9999) that has been recorded in four variables.

```
. mvdecode hb pot sodium totbil  , mv(9999)
          hb: 103 missing values generated
         pot: 26 missing values generated
      sodium: 87 missing values generated
      totbil: 156 missing values generated
```

Missing values Example 3: taking care with greater or less than

Since Stata stores missing values as (impossibly) large values we need to be careful when generating new variables based on values of existing variables using > or < statements. For example,

```
. generate sbp140=(sbp>=140)
```

will return 0 if sbp<140, and 1 if sbp≥140 OR if sbp is missing.

To overcome this problem it is necessary to add a condition outside the brackets:

```
. generate sbp140=(sbp>=140) if sbp~=.
```

If we have extended missing values then we would need to use exclude observations by conditioning on whether the value is less than “.”. For instance,

```
. generate hyper=(sbp>=140) if sbp<.
```

Alternatively we could use the ‘not missing’ function e.g.

```
. generate hyper=(sbp>=140 | dbp>=90) if !missing(sbp, dbp)
```

Exploring missing data patterns

Above in egen example 4 we encountered some issues with missing data. A neat way of exploring that example further is with the `misstable` command along with its various subcommands.

The `summarize` subcommand produces a basic summary table for the variables specified.

```
. misstable summarize  strisch strhem stremb stroth
                                                     Obs<.
                                            +------------------------------
             |                              | Unique
    Variable |     Obs=.     Obs>.     Obs<.  | values        Min         Max
-------------+------------------------------+------------------------------
     strisch |        16              2,484  |      2          0           1
      strhem |        72              2,428  |      2          0           1
      stremb |        68              2,432  |      2          0           1
      stroth |        73              2,427  |      2          0           1
  -----------------------------------------------------------------------------
```

We see here the number of missing values for each variable plus a brief summary of the non-missing values.

The `patterns` subcommand allows an exploration of any patterns of missing values among the set of variables.

```
. misstable patterns strisch strhem stremb stroth, freq

    Missing-value patterns
      (1 means complete)

              |   Pattern
    Frequency |  1  2  3  4
  ------------+-------------
        2,418 |  1  1  1  1
              |
           58 |  1  0  0  0
            8 |  0  0  0  1
            8 |  1  1  1  0
            5 |  0  1  0  0
            1 |  0  0  0  0
            1 |  0  0  1  0
            1 |  0  1  1  1
  ------------+-------------
        2,500 |

  Variables are  (1) strisch  (2) stremb  (3) strhem  (4) stroth
```

This allows us to see that for 2418 people all the values are non-missing. The most common missing pattern among the four variables is for number 1 (see key below = strisch) to be present but for the remaining 3 to be missing.

Chapter 7: Descriptive Statistics and Simple Hypothesis Tests

Aims and Objectives of Chapter 7:

By the end of this chapter you should be able to:

- Obtain univariable summaries of location and spread for continuous variables
- Produce one-way tables summarising categorical variables
- Produce simple graphs for describing univariable distributions
- Describe associations between categorical and continuous variables
- Describe associations between continuous variables
- Carry out a two-sample t-test and chi-square test
- Copy and paste output from the Results Window as text or tables

The following commands are used in this chapter: `summarize, table, tabulate, tabstat, graph (histogram, box, twoway scatter, matrix), pwcorr, ttest, tabodds`

7.1 Summarising and describing data

Having checked for errors in the data the next step is to produce simple descriptive summaries of the data. It is tempting to overlook this stage and rush into more complicated statistical modelling. However a good initial description of the data is essential. Such summaries usually involve tables and/or figures. It is fairly easy to produce both tables and figures, though not so easy to produce tables and figures that convey information effectively. It can be helpful to look at how data are presented in tables and figures in journals such as the New England Journal of Medicine, the BMJ or the Lancet.

How to summarize data is determined mostly by the format and distribution of the data. For example we would summarize age (in years) and sex (male/female) very differently, the first being a continuous numeric variable and the second being categorical (could be stored as string or numeric).

For continuous numeric variables it is important to check the shape of the distribution. Variables that are approximately normally distributed can be appropriately summarised using the mean and standard deviation of the distribution whereas skewed distributions may best be described using the median and perhaps the inter-quartile range or 5th and 95th centiles. The shape of the distribution is best checked visually using the **`histogram`** command (as described above) or the **`qnorm`** command.

For categorical variables we are generally interested in absolute numbers and relative frequencies e.g. row or column percentages.

7.2 Summarising continuous variables

For quantitative continuous variables, you will usually be interested in summary measures like means, medians and standard deviations, as well as maximum and minimum values and percentiles. These describe the location and spread of the distribution.

Most of this information can be obtained using the `summarize` command. Typing `summarize` on its own provides the number of observations, the mean, the standard deviation, the minimum value and the maximum value for all the variables in the dataset. You can restrict the number of variables which are summarised by including a list of variable names with the command.

For the following examples we will load *bl_combined2.dta.*

For example, to obtain summaries of the variables *wt* and *sbp*, submit the command,

```
. summ wt sbp

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          wt |      2497    79.19049    16.87359         31      149.9
         sbp |      2499    124.1012    16.81271         72        189
```

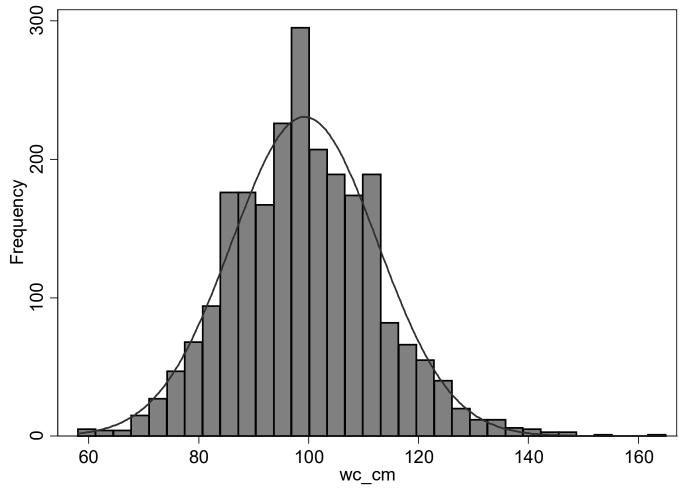
Additional information can be obtained by specifying the option `detail` (can be abbreviated to `d`). Stata will then present the number of observations, measures of location e.g. the mean and median (50th percentile), measures of spread e.g. the standard deviation, variance, various percentiles and the four smallest and four largest observations and measures of skewness and kurtosis. For example,

```
. sum sbp, detail

               Systolic Blood Pressure (mmHg)
-------------------------------------------------------------
      Percentiles      Smallest
 1%           90             72
 5%           98             74
10%          101             80       Obs                2499
25%          110             85       Sum of Wgt.        2499

50%          123                      Mean           124.1012
                        Largest       Std. Dev.      16.81271
75%          135            180
90%          145            180       Variance       282.6673
95%          150            187       Skewness       .2785866
99%          170            189       Kurtosis       3.111947
```

Distributional plots for continuous variables

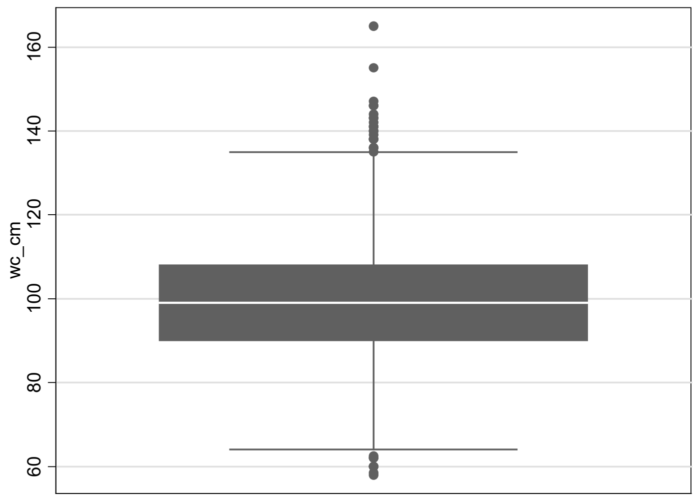
Useful graphical summaries for continuous numeric variables include histograms and box-and-whisker plots. Two examples are shown below.

```
. histogram wc_cm, freq normal
```



The option `freq` requests frequencies rather than density and `normal` overlays with a normal distribution given the mean and standard deviation of the variable.

```
. graph box wc_cm
```



The line within the box is the median value. The box extends from the 25th to 75th centiles of the distribution i.e. the interquartile range. The whiskers extend up to 1.5 times the interquartile range above the 75th centile and below the 25th centile. If there are no values at or beyond these points the whiskers extend to the lowest or highest value within this range. Values beyond these points i.e. outlying values, are plotted individually. See help graph box and the pdf documentation for more information on the history of box plots.

Other distributional plots can be found under the *Graphics > Distributional Plots* menu.

Summarising a continuous variable over levels of a categorical variable.

Often we will want to examine the association between a continuous variable and a categorical variable, e.g. does systolic blood pressure vary across categories of age-group or categories of over-weight?
Here we want to obtain a summary of *sbp* for patients (i) who are obese and (ii) are not obese.

As there are just two categories of obese (0/1 = no/yes) we could do this quite simply using an `if` statement e.g.

```
. summ sbp if obese==1

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sbp |       665    127.3564    16.16868         74        187

. summ sbp if obese==0

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sbp |      1820    122.9052    16.86602         72        189
```

Alternatively we could use the `bysort` prefix e.g.

```
. bysort obese: summ sbp

------------------------------------------------------------------------

-> obese = 0

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sbp |      1820    122.9052    16.86602         72        189

------------------------------------------------------------------------

-> obese = 1

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sbp |       665    127.3564    16.16868         74        187

------------------------------------------------------------------------

-> obese = .

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sbp |        14    124.9643    20.41536         90        168
```

Note that with the bysort prefix Stata treats patients with a missing obese value as a separate group and automatically returns a summary for this group of patients.

However, almost certainly it would be better to produce these summary statistics in a table format using the `table` command. This command can be used to produce basic frequency tabulations for categorical variables, but is most useful when used with the `statistic(…)` option which a range of statistics to be specified, including frequency, mean, median, standard deviation (sd), and percentiles (p1, p2 etc).

For example, to produce a table of the number, mean and standard deviation of systolic blood pressure (sbp) for each category of obese:

```
table (obese), statistic(count sbp) statistic(mean sbp) statistic(sd
sbp)
```

```
--------------------------------------------------------------------
          |  Number of nonmissing values       Mean   Standard deviation
----------+---------------------------------------------------------
BMI 30+   |
  BMI<30  |                        1,820   122.9137              16.8668
  BMI 30+ |                          665   127.3609             16.16739
  Total   |                        2,485   124.1038             16.79512
--------------------------------------------------------------------
```

Note: The `table` command must be typed in full. Tabulate may be abbreviated as `tab`.

We can reduce the number of decimal places reported with the nformat option, e.g.

```
table (obese), statistic(count sbp) statistic(mean sbp) statistic(sd
sbp) nformat(%4.1f mean sd)
```

```
--------------------------------------------------------------------
          | Number of nonmissing values    Mean   Standard deviation
----------+---------------------------------------------------------
BMI 30+   |
  BMI<30  |                       1,820   122.9                 16.9
  BMI 30+ |                         665   127.4                 16.2
  Total   |                       2,485   124.1                 16.8
--------------------------------------------------------------------
```

The % indicates that this is a format; the 4.1 specifies 1 decimal place and a minimum of 4 characters in width; the f indicates a fixed format. The format will be applied to the variables listed after the format in the brackets.

The command `tabstat` can also be used to get summaries of one or more continuous variables over levels of a categorical variable. In the following example we build up from a simple summary of *sbp* and *dbp*.

Firstly, to get the means of *sbp* and *dbp* in the whole dataset:

```
. tabstat sbp dbp

   stats |       sbp       dbp
---------+--------------------
    mean |  124.1012  74.58944
------------------------------
```

With the `stats()` option we can request various statistical summaries. Here we request counts, means and standard deviations:

```
. tabstat  sbp dbp   , stats(count mean sd)

   stats |       sbp       dbp
---------+--------------------
       N |      2499      2499
    mean |  124.1012  74.58944
      sd |  16.81271  10.20323
------------------------------
```

Using the `by()` option we can obtain these summary measures over levels of a categorical variable:

```
. tabstat sbp dbp, by(bmi4) stats(count mean sd)

Summary statistics: N, mean, sd
  by categories of: bmi4

  bmi4 |       sbp       dbp
-------+--------------------
   16- |       272       272
       |   119.318  71.97243
       |  17.50248  10.46185
-------+--------------------
   22- |       466       466
       |  120.9421  72.91309
       |  17.36223  10.07976
-------+--------------------
   25- |      1074      1074
       |   124.784  74.94274
       |  16.19471   9.87823
-------+--------------------
   30- |       665       665
       |  127.3564  76.25338
       |  16.16868  10.20251
-------+--------------------
 Total |      2477      2477
       |  124.1516   74.5866
       |   16.7733  10.16568
----------------------------
```

See `help tabstat` for more options.

Two-sample t-test

The two-sample t-test can be used to compare the mean levels of a normally distributed continuous variable in two independent groups.

Below we have carried out a t-test to calculate the mean difference in systolic blood pressure and its 95% confidence interval between patients who were/weren't obese.

```
. ttest sbp, by(obese)

Two-sample t test with equal variances
------------------------------------------------------------------------------
   Group |     Obs        Mean    Std. Err.   Std. Dev.   [95% Conf. Interval]
---------+--------------------------------------------------------------------
       0 |    1820    122.9052    .3953456    16.86602    122.1298    123.6806
       1 |     665    127.3564    .6269946    16.16868    126.1253    128.5875
---------+--------------------------------------------------------------------
combined |    2485    124.0964     .336914    16.79509    123.4357     124.757
---------+--------------------------------------------------------------------
    diff |           -4.451171    .7559186               -5.933467   -2.968875
------------------------------------------------------------------------------
    diff = mean(0) - mean(1)                                      t =  -5.8884
Ho: diff = 0                                     degrees of freedom =     2483

    Ha: diff < 0                 Ha: diff != 0                 Ha: diff > 0
 Pr(T < t) = 0.0000         Pr(|T| > |t|) = 0.0000          Pr(T > t) = 1.0000
```

There are four rows of results plus some hypothesis tests and p-values below.

The first two rows (0 and 1) are the means, 95% CIs etc. in the two groups separately. In this case 0 = not obese and 1 = obese; compare these to the tables on the previous page.

The third row (combined) presents the results for the whole sample pooled i.e. both groups combined.

The fourth row (diff) contains the results we are most interested in i.e. the mean difference and its 95% CI. Here the mean difference is estimated to be -4.45 with 95% CI (-5.93, -2.97). Since the 95% CI does not include 0 (no difference) we can infer that the difference is statistically significant.

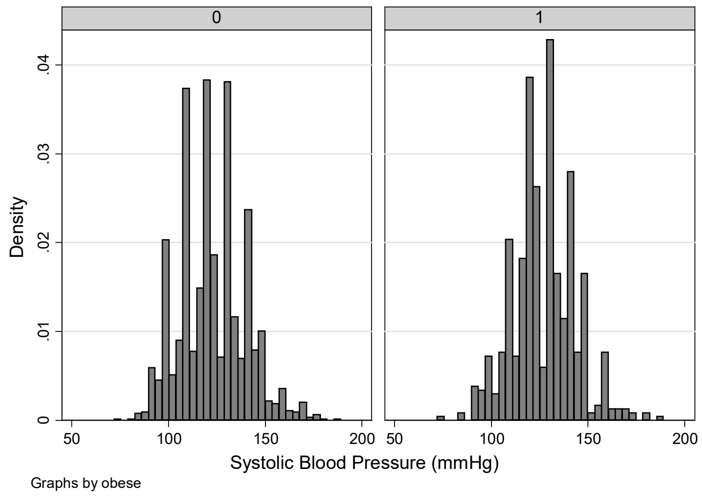
Stata presents three p-values – we are usually interested in the middle p-value which is from a two-sided test. The p-values on the left and right of the output are from one-sided tests. Here the two-sided p-value is <0.0001 which is strong evidence against the null hypothesis that the means in the two groups are equal.

An additional assumption is that the variances in the two samples are equal, though there is a correction to the t-test that can account for non-equal variances. In the example above we have assumed that the variances in the two samples. If we suspected that they were not equal we could add the option `unequal`.

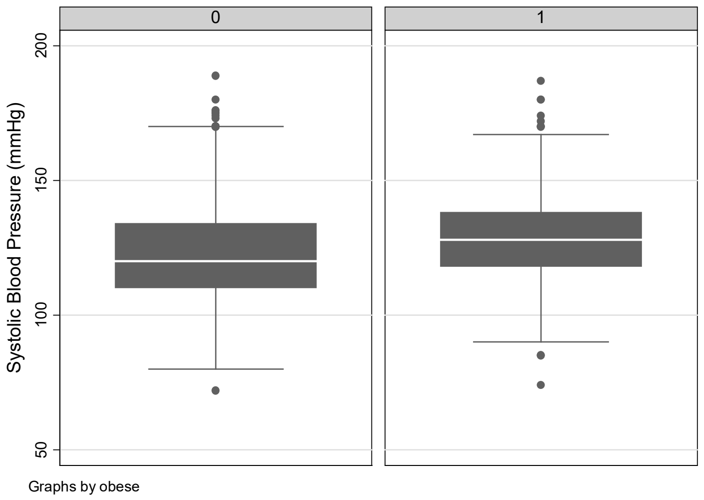
Graphs of associations between continuous and categorical variables

Both the histogram and box plots can be used to describe the distribution of continuous variables over categories of another variable.

```
. histogram sbp, by(obese)
```



```
. graph box sbp, over(obese)
```



Correlation - association between continuous variables

One common method used to describe the linear association between continuous variables is the correlation coefficient. The correlation coefficient can take values between -1 (perfect negative correlation) and +1 (perfect positive correlation). A value of 0 indicates no association between the two variables.

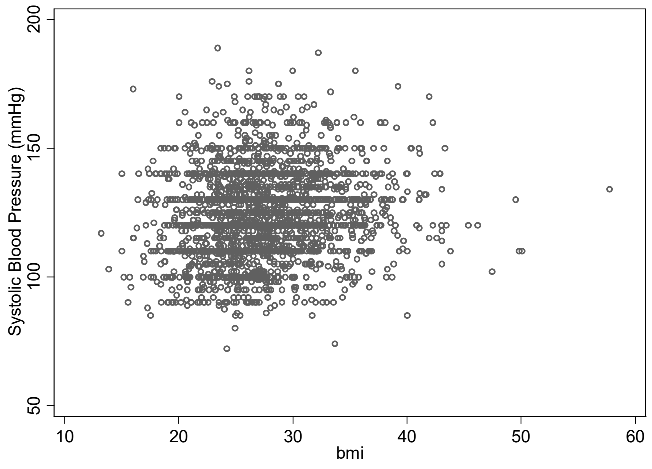
To calculate the correlation between sbp and bmi:

```
. correlate sbp bmi
(obs=2485)

             |      sbp      bmi
-------------+------------------
         sbp |   1.0000
         bmi |   0.1575   1.0000
```

Here the correlation between sbp and bmi is 0.16 (to 2dp). The number of patients with both measurements is 2485.

We could also produce a twoway scatter plot:

```
. twoway scatter sbp bmi, ms(oh)
```



We can add more variables to the variable list:

```
. corr sbp bmi age wt ht wc_cm hrate
(obs=2348)

          |      sbp      bmi      age       wt       ht    wc_cm    hrate
----------+---------------------------------------------------------------
      sbp |   1.0000
      bmi |   0.1579   1.0000
      age |   0.0669  -0.1532   1.0000
       wt |   0.1287   0.8485  -0.1903   1.0000
       ht |   0.0038   0.0791  -0.1085   0.5850   1.0000
    wc_cm |   0.1152   0.7847  -0.0895   0.8138   0.3313   1.0000
    hrate |  -0.0594  -0.0076  -0.0473  -0.0248  -0.0456   0.0332   1.0000
```

Note that the number of observations is now 2348. In calculating these correlation coefficients Stata has used only those patients with complete data on all the variables specified.

We can obtain all the pair wise correlations using the `pwcorr` command. With this command each correlation coefficient is calculated using the patients with complete data for that pair.

```
. pwcorr sbp bmi age wt ht wc_cm hrate

             |      sbp      bmi      age       wt       ht    wc_cm    hrate
-------------+---------------------------------------------------------------
         sbp |   1.0000
         bmi |   0.1575   1.0000
         age |   0.0737  -0.1524   1.0000
          wt |   0.1312   0.8488  -0.1839   1.0000
          ht |   0.0054   0.0755  -0.1057   0.5812   1.0000
       wc_cm |   0.1180   0.7845  -0.0852   0.8135   0.3314   1.0000
       hrate |  -0.0463  -0.0059  -0.0455  -0.0260  -0.0514   0.0307   1.0000
```

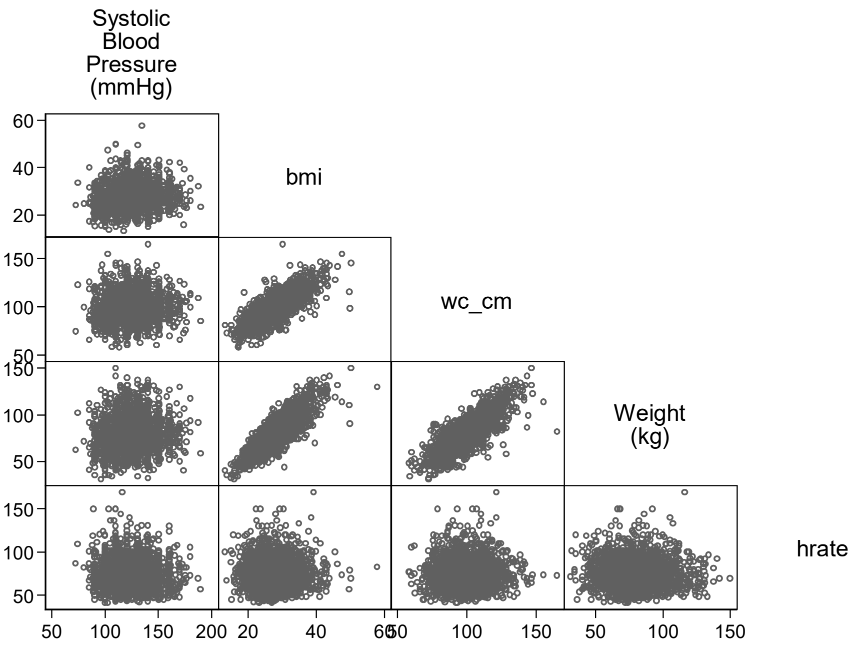
Options for `pwcorr` include `obs` (number of observation) and `sig` (p-value for test of correlation is equal to zero).

```
. pwcorr sbp bmi age wc hrate, obs sig

             |      sbp      bmi      age       wc    hrate
-------------+---------------------------------------------
         sbp |   1.0000
             |
             |     2499
             |
         bmi |   0.1575   1.0000
             |   0.0000
             |     2485     2486
             |
         age |   0.0737  -0.1524   1.0000
             |   0.0002   0.0000
             |     2499     2486     2500
             |
          wc |   0.1222   0.1792  -0.0321   1.0000
             |   0.0000   0.0000   0.1180
             |     2369     2360     2369     2369
             |
       hrate |  -0.0463  -0.0059  -0.0455   0.0318   1.0000
             |   0.0210   0.7706   0.0234   0.1227
             |     2486     2472     2486     2357     2486
```

Stata presents the p-values to 4 decimal places. For some correlations e.g. sbp and bmi the p-value is presented as 0.0000. You should report the p-values such as this as p<0.0001. Or you could be a bit more precise and report as p<0.00005 – since Stata would have presented anything greater than this rounded to 0.0001. e.g. 0.00008 would have been reported as 0.0001.

We can also produce multiple twoway scatter plots as a quick way of examining the association between multiple continuous variables.

```
. graph matrix sbp bmi wc_cm wt hrate, ms(oh) half msize(*0.85)
```



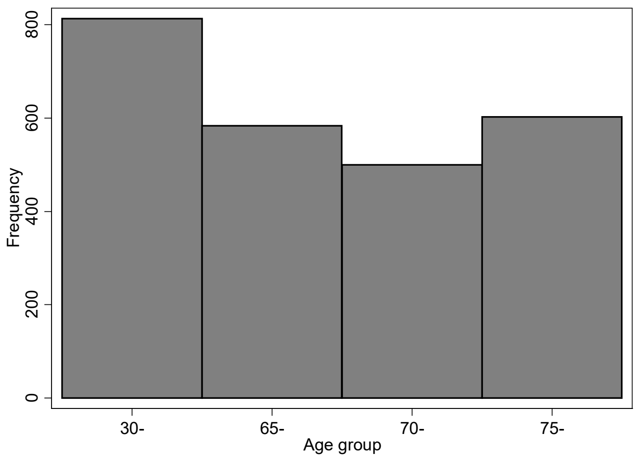
7.3 Summarising categorical variables

The distribution of categorical variables can be viewed in Stata using the `tabulate` command, which allows one-way or two-way tabulations. For example,

```
. tab agegroup

  Age group |      Freq.     Percent        Cum.
------------+-----------------------------------
        30- |        245        9.54        9.54
        60- |      1,121       43.64       53.17
        70- |        946       36.82       90.00
        80- |        257       10.00      100.00
------------+-----------------------------------
      Total |      2,569      100.00
```

The distribution can be displayed graphically using a graph twoway histogram with the discrete option.

```
. graph twoway histogram agegroup, discrete xlab(0(1)3, value) freq
```



This is a different command to the histogram command we have used before. The `graph twoway histogram` command has extra options.

The `xlab()` option relates to the x-axis labelling. The `0(1)3` is shorthand for label the values 0 to 3 in steps of 1. The option `value` within `xlab()` says use the value label rather than the numeric values.

Two-way associations between categorical variables

To obtain a *two-way* table of age-group and smoking, submit the command,

```
. tab agegroup smoke
```

```
           |          Smoking status
 Age group |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
       65- |        73        260        251 |       584
       70- |        41        228        231 |       500
       75- |        26        268        309 |       603
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500
```

To obtain row or column percentages, use the `row` or `col` options. For example,

```
. tab agegroup smoke, row

+----------------+
| Key            |
|----------------|
|   frequency    |
| row percentage |
+----------------+

           |          Smoking status
 Age group |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
           |     15.87      44.40      39.73 |    100.00
-----------+---------------------------------+----------
       65- |        73        260        251 |       584
           |     12.50      44.52      42.98 |    100.00
-----------+---------------------------------+----------
       70- |        41        228        231 |       500
           |      8.20      45.60      46.20 |    100.00
-----------+---------------------------------+----------
       75- |        26        268        309 |       603
           |      4.31      44.44      51.24 |    100.00
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500
           |     10.76      44.68      44.56 |    100.00
```

To carry out a general chi square test i.e. a test of the null hypothesis of no association between agegroup and smoke:

```
. tab agegroup smoke, row chi
[key omitted]

           |          Smoking status
 Age group |   Current         Ex      Never |     Total
-----------+---------------------------------+----------
       30- |       129        361        323 |       813
           |     15.87      44.40      39.73 |    100.00
-----------+---------------------------------+----------
       65- |        73        260        251 |       584
           |     12.50      44.52      42.98 |    100.00
-----------+---------------------------------+----------
       70- |        41        228        231 |       500
           |      8.20      45.60      46.20 |    100.00
-----------+---------------------------------+----------
       75- |        26        268        309 |       603
           |      4.31      44.44      51.24 |    100.00
-----------+---------------------------------+----------
     Total |       269      1,117      1,114 |     2,500
           |     10.76      44.68      44.56 |    100.00

          Pearson chi2(6) =  58.7493   Pr = 0.000
```

Note as pointed out above that Stata often presents p-values to just 3 or 4 decimal places. Here the p-value is 0.000 to 3dp. We could report this as p<0.0005.

The p-value is obtained from the chi-square test statistic which in the example above is 58.7493 on 6 degrees of freedom. We could use Stata's `chiprob()` function. This function takes two arguments; first the degrees of freedom and second the chi-square statistic. We can use the display command and use a format to get the p-value to 10 decimal places.

```
. disp %12.10f chiprob(6, 58.7493)
0.0000000001
```

So the first significant figure is at 10 decimal places. You should report such a p-value as p<0.0001 at most.

By default Stata omits any missing values. If we want to a table including a row/column for missing values i.e. treat missing values in the same way as non-missing values we must specify the `missing` option. For example,

```
. tab agegroup smoke, miss

           |                    smoke
 Age group |                 Current           Ex        Never |     Total
-----------+-----------------------------------------------------+----------
       30- |         0            48          102           95 |       245 
       60- |         1           151          504          465 |     1,121 
       70- |         0            64          432          450 |       946 
       80- |         0             9          111          137 |       257 
-----------+-----------------------------------------------------+----------
     Total |         1           272        1,149        1,147 |     2,569 
```

Here we see that there is one person missing smoking status who is in the 60-69 year age-group.

Association between a binary outcome and a categorical variable

Here we are interested in investigating the association between the outcome of all cause mortality (*acdeath*) and the categorical variable *sbpcat* (created above).

We start with a simple two-way tabulation with column percentages and a chi square test.

```
. tab acdeath sbpcat  , col chi

 All cause |                    sbpcat
     death |       70-       120-        130-        140- |     Total
-----------+----------------------------------------------+----------
         0 |       697        470         452         461 |     2,080
           |     79.20      80.90       86.76       89.17 |     83.23
-----------+----------------------------------------------+----------
         1 |       183        111          69          56 |       419
           |     20.80      19.10       13.24       10.83 |     16.77
-----------+----------------------------------------------+----------
     Total |       880        581         521         517 |     2,499
           |    100.00     100.00      100.00      100.00 |    100.00

          Pearson chi2(3) =  30.1937   Pr = 0.000
```

There is clearly a trend across the categories of *sbpcat* with the risk of dying decreasing with increasing *sbp*.

We now use the `tabodds` command to estimate the odds of dying in each category

```
. tabodds acdeath sbpcat

---------------------------------------------------------------------------
     sbpcat  |      cases     controls       odds      [95% Conf. Interval]
-------------+-------------------------------------------------------------
         70- |        183          697    0.26255        0.22311   0.30897
        120- |        111          470    0.23617        0.19204   0.29044
        130- |         69          452    0.15265        0.11849   0.19666
        140- |         56          461    0.12148        0.09205   0.16030
---------------------------------------------------------------------------
Test of homogeneity (equal odds): chi2(3)  =    30.18
                                  Pr>chi2  =   0.0000

Score test for trend of odds:     chi2(1)  =    28.91
                                  Pr>chi2  =   0.0000
```

The output shows the number of cases (deaths) and controls (alive), the odds of death and a 95% CI for the odds in each category. It also shows a test of homogeneity of the odds (same as the general chi square test above) and a test of trend in the odds. The odds of the outcome will always be greater than the risk of the outcome – compare the percentages and the odds in the two outputs above.

The test for trend suggests that there is a trend in the odds of dying across the categories of *sbpcat*.

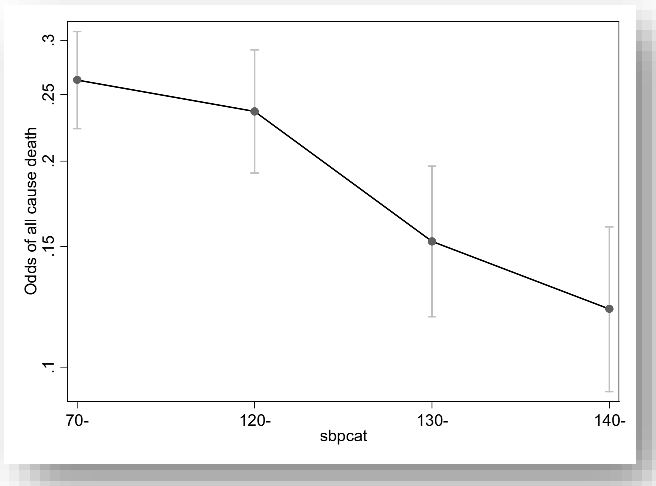
Using the `tabodds` command we can also get odds ratios:

```
. tabodds acdeath sbpcat , or

---------------------------------------------------------------------------
     sbpcat |  Odds Ratio       chi2       P>chi2     [95% Conf. Interval]
------------+--------------------------------------------------------------
        70- |    1.000000          .            .            .           .
       120- |    0.899512       0.62       0.4304     0.691240    1.170536
       130- |    0.581423      12.64       0.0004     0.429610    0.786882
       140- |    0.462667      22.78       0.0000     0.334496    0.639951
---------------------------------------------------------------------------
Test of homogeneity (equal odds): chi2(3)  =    30.18
                                  Pr>chi2  =   0.0000

Score test for trend of odds:     chi2(1)  =    28.91
                                  Pr>chi2  =   0.0000
```

The lowest level of *sbpcat* is used as the reference group. The odds ratio for dying in the 140- group compared to the 70- group is 0.46, 95% CI 0.33 to 0.64, i.e. the odds of dying in the highest category of *sbp* is 54% lower than that in the lowest category.

We can also produce a graph of the odds with the graph option.

```
. tabodds acdeath sbpcat, graph ci yscale(log) xlab(,value)
ytitle(Odds of death)

[output omitted]
```

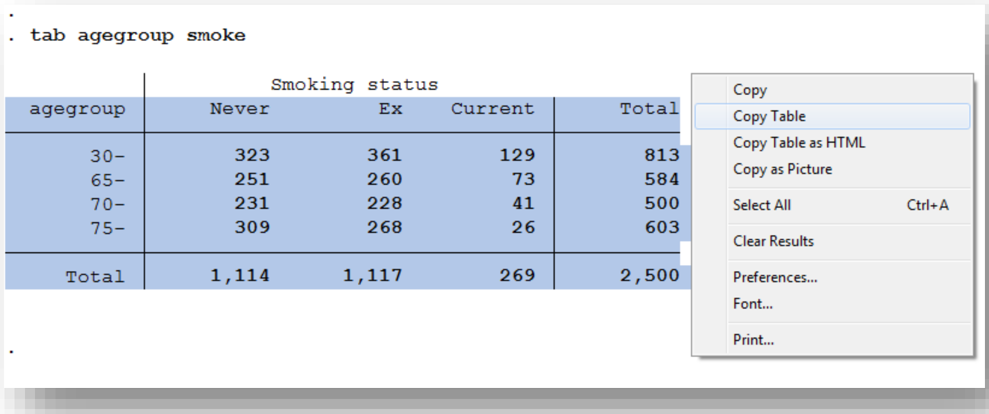


Extra graph options have been specified:

`graph`	plot of odds against categories
`ci`	plot the 95% CIs
`xlab(, value)`	use value labels for the x-axis
`ytitle(Odds of death)`	specifies the y-axis title
`yscale(log)`	use a log scale for the y-axis.

7.4 Moving tables into Word or Excel using Copy & Paste

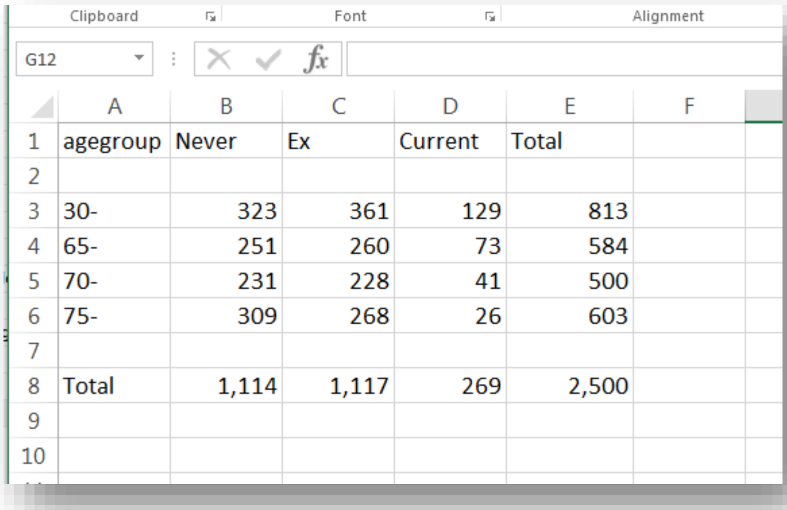
It is possible to copy and paste tables straight from the Results Window into Word or Excel. Use the mouse to highlight the table and right-click. The first two options are *Copy* and *Copy Table* (see figure below). Selecting 'Copy Table' will allow the table to be pasted as a tab-delimited table directly into Excel or Word. Note here that we have selected the column headers but not the overall columns title.



```
.
. tab agegroup smoke

           |        Smoking status
  agegroup |     Never        Ex   Current |     Total
-----------+---------------------------------+----------
       30- |       323       361       129 |       813
       65- |       251       260        73 |       584
       70- |       231       228        41 |       500
       75- |       309       268        26 |       603
-----------+---------------------------------+----------
     Total |     1,114     1,117       269 |     2,500

.
```

Copying a table directly from the results window

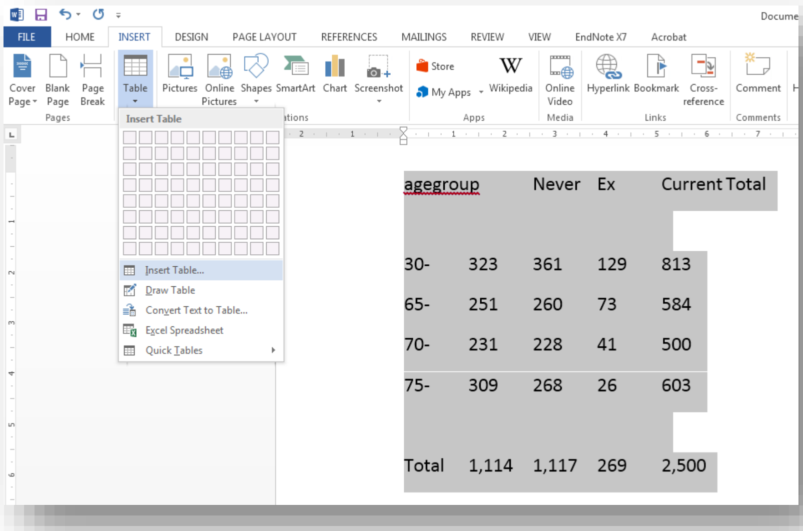
When pasting into an Excel worksheet the cells of the table will occupy separate cells of the spreadsheet.



	A	B	C	D	E	F
1	agegroup	Never	Ex	Current	Total	
2						
3	30-	323	361	129	813	
4	65-	251	260	73	584	
5	70-	231	228	41	500	
6	75-	309	268	26	603	
7						
8	Total	1,114	1,117	269	2,500	
9						
10						

Inserting a table into Excel

To create a table in Word, follow the same procedure as above i.e. select table in the Results Window, copy as a table and paste the table into Word. You then need to select the pasted table in Word and then select *Insert* > *Table* > *Insert Table* as shown in the figure below.



agegroup	Never	Ex	Current	Total
30-	323	361	129	813
65-	251	260	73	584
70-	231	228	41	500
75-	309	268	26	603
Total	1,114	1,117	269	2,500

Creating a table in Word

Tables can be pasted as plain text into a Word document using the *Copy* option rather than *Copy Table*. Having copied and pasted into Word you will need to select a non-proportional font such as Courier New or Consolas, otherwise the table columns will not correctly align – see figures below.

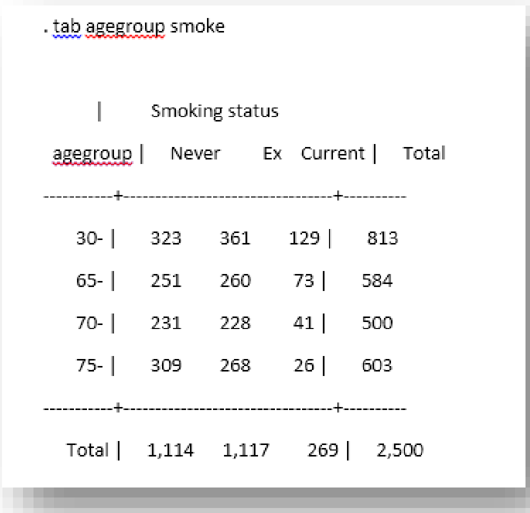
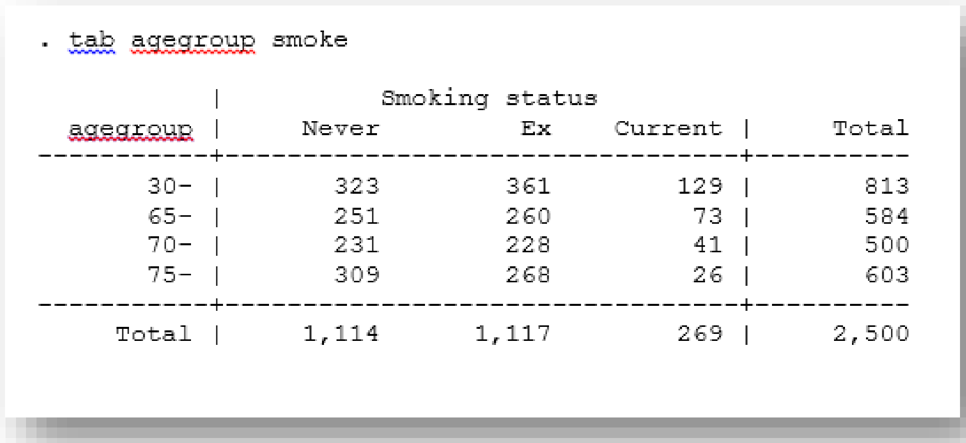

```
. tab agegroup smoke

          |        Smoking status
 agegroup |    Never        Ex   Current |    Total
-----------+--------------------------------+----------
      30- |      323       361       129 |      813
      65- |      251       260        73 |      584
      70- |      231       228        41 |      500
      75- |      309       268        26 |      603
-----------+--------------------------------+----------
    Total |    1,114     1,117       269 |    2,500
```

Table will be misaligned if a proportional font such as Calibri or Times New Roman is used



```
. tab agegroup smoke

           |          Smoking status
  agegroup |     Never         Ex    Current |     Total
-----------+---------------------------------+----------
       30- |       323        361        129 |       813
       65- |       251        260         73 |       584
       70- |       231        228         41 |       500
       75- |       309        268         26 |       603
-----------+---------------------------------+----------
     Total |     1,114      1,117        269 |     2,500
```

Same table using Courier New font

In version 14 Stata introduced a new command `putexcel` for exporting results, matrices and images directly to Excel, and in version 15 added `putdocx` and `putpdf` for writing paragraphs, images and tables to Office Open XML (Word) and PDF documents, respectively. This suite of commands allows the user to automate, for example, the generation (and formatting) of regular reports. We will not look in detail at these commands during this introductory module, but would encourage you to explore them yourself as you become more familiar with Stata. See `help putexcel` (or `putdocx` or `putpdf`) for details.

Chapter 8: Advanced Data Management

Aims & Objectives of Chapter 8

By the end of this chapter you should:

- Know how to work efficiently using loops
- Be able to process string variables
- Understand Stata's elapsed date format
- Be able to manage and create date variables
- Be able to manage repeated measures data
- Know how to create summary datasets
- Be able to reshape datasets into wide and long formats

The following commands and functions are used in this chapter: `forvalues`, `foreach`, `generate`, `lower()`, `upper()`, `substr()`, `strpos()`, `egen`, `bysort`, `mdy()`, `date()`, `contract`, `collapse`, `reshape wide` and `reshape long`. We also deal with subscripts and Stata's system variables _n and _N.

8.1 Creating Loops

A loop in Stata is simply a sequence of commands that are continually repeated until a condition is fulfilled e.g. a pre-specified sequence is completed or the end of a list is reached. Being able to create and used loops can save time and lines of Stata code in a do-file i.e. it is about working efficiently and making best use of time and space.

There are a number of ways of creating loops in Stata. Here we will look at how to create loops using the `forvalues` and `foreach` commands.

1. Looping with forvalues

The `forvalues` command is the simplest way to create a loop which runs over consecutive numeric values (e.g. 1, 2, 3, 4, 5) or a sequence of values (e.g. 0, 5, 10, 15, 20).

The syntax is:

```
forvalues macname = range       {
      command(s) to be executed repeatedly
}
```

Taking the syntax in order:

- `forvalues` is the Stata command

- *macname* is a local macro name provided by the user and which will act as the counter

- *range* specifies the values over which the local macro will be incremented e.g.
    - 1/5 meaning 1 to 5 in steps of 1
    - 0(2)10 meaning 0 to 10 in steps of 2
    - 3 6 to 36 meaning 3 to 36 in steps of 3 i.e. the step is determined by the interval between the first two numbers in the sequence

- { the open brace (a curly bracket) must appear at the end of the `forvalues` line; nothing may follow the open brace (except comments)

- Commands to be executed repeatedly until the end of the loop. These commands will usually include a reference to the local macro defined by the `forvalues` command. When referring to a local macro (called dereferencing) the *macroname* must be punctuated correctly i.e. enclosed by a left single quote and a right single quote (see Figure 8.1 and text below).

- } the close brace appears on a line on its own following all the commands that are to be repeated.

Local macros
Macros are the variables of Stata programs. They consist of a macro name and macro content. The content of a macro is accessed by dereferencing using precise punctuation. When dereferencing a *local macro* we must put a left single quote (generally located on the key immediately left of the 1 key) immediately before the macro name and a right-single quote, usually on the @ key, immediately following the macro name.

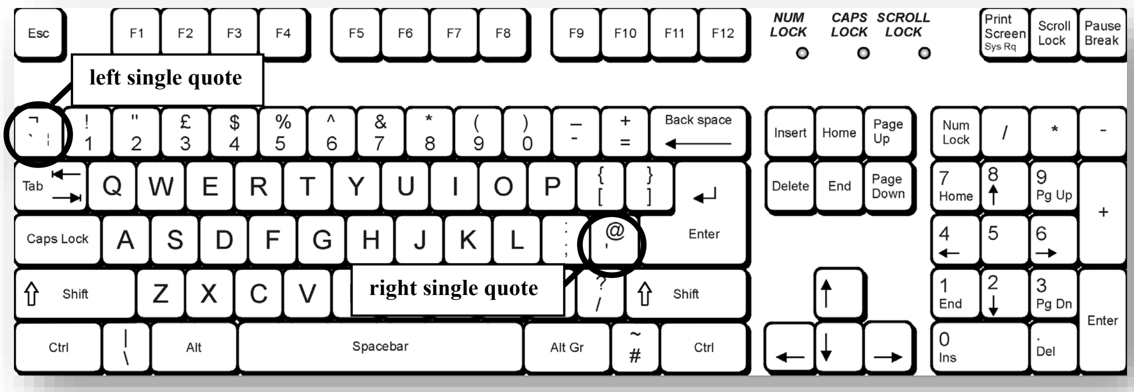


Figure 8.1: Location of left and right single quotes

Forvalues: Example 1

Type the following commands in a do-file and then run (you will need to run all three lines at the same time).

```
forvalues i = 1/5 {
disp "This is loop number: `i'"
}
```

The `forvalues` command creates a local macro called i, which takes the value 1 on the first loop, then 2 on the second loop, etc., through to 5 on the final loop. While i is less than or equal to 5 Stata continues to execute the commands in the loop. The command inside the loop asks Stata to display "This is loop number: `i'" where local macro i has been dereferenced appropriately. On the first pass through the loop Stata reads this as display "This is loop number 1".

The output in the *Results Window* should be as follows:

```
. forvalues i=1/5 {
  2. display "This is loop number: `i'"
  3. }

This is loop number: 1
This is loop number: 2
This is loop number: 3
This is loop number: 4
This is loop number: 5
```

Note that Stata first prints out the forvalues loop in full before executing the commands in sequence.

Forvalues: Example 2

Copy and run the following three lines in a do-file.

```
forvalues k = 10(-2)0 {
disp "Local macro k is now equal to `k'"
}
```

```
. forvalues k = 10(-2)0 {
  2.
. disp "Local macro k is now equal to `k'"
  3.
. }

Local macro k is now equal to 10
Local macro k is now equal to 8
Local macro k is now equal to 6
Local macro k is now equal to 4
Local macro k is now equal to 2
Local macro k is now equal to 0
```

2. Looping with foreach

The `foreach` command is similar to `forvalues` in that it enables looping through batches of commands, but is more flexible in that it allows looping through a list of numbers (not necessarily consecutive), a list of variable names, the elements of a local (or global) macro or indeed through any list. The syntax for `foreach` is:

```
foreach macname of list_type list {
      command(s) to be executed repeatedly
}
```

Taking the syntax in order:

- `foreach` is the Stata command
- *macname* is a local macro name provided by the user e.g. A, var, k, bob
- *of list_type* tells Stata what to expect e.g.
    - `of varlist` meaning *list* will consist of names of variables in memory
    - `of numlist` meaning *list* will consist of numbers
    - `of local` meaning list will consist of a local macro name
    - `of global` meaning that list will consist of a global macro name
- { the open brace (a curly bracket) must appear on its own at the end of the `foreach` line
- Commands to be executed – these are the commands which will be repeated until the end of the loop. These commands will usually include a reference to the local macro defined by the `foreach` command.
- } the close brace appears on a line on its own following all the commands that are to be repeated.

Foreach: Example 1

Open bl_combined2.dta. Type the following commands in a do-file and run.

```
foreach V of varlist diab angina cvd {
     tabulate bmicat `V' , row chi  nokey
}
```

The output from the *Results Window* appears below. There are three variables (diab, angina and cvd) in the variable list. On the first pass through the loop the local macro V contains the variable name *diab*, on the second pass *angina* and on the final pass *cvd*.

```
. foreach V of varlist diab angina cvd {
  2. tabulate bmicat `V' , row chi  nokey
  3. }

       BMI |       Diabetes
categories |         0          1 |     Total
-----------+----------------------+----------
       12- |       208         72 |       280
           |     74.29      25.71 |    100.00
-----------+----------------------+----------
       22- |       332        134 |       466
           |     71.24      28.76 |    100.00
-----------+----------------------+----------
       25- |       755        319 |     1,074
           |     70.30      29.70 |    100.00
-----------+----------------------+----------
       30- |       412        253 |       665
           |     61.95      38.05 |    100.00
-----------+----------------------+----------
     Total |     1,707        778 |     2,485
           |     68.69      31.31 |    100.00

          Pearson chi2(3) =  20.8086   Pr = 0.000


       BMI |        Angina
categories |        No        Yes |     Total
-----------+----------------------+----------
       12- |       190         90 |       280
           |     67.86      32.14 |    100.00
-----------+----------------------+----------
       22- |       287        179 |       466
           |     61.59      38.41 |    100.00
-----------+----------------------+----------
       25- |       573        501 |     1,074
           |     53.35      46.65 |    100.00
-----------+----------------------+----------
       30- |       344        321 |       665
           |     51.73      48.27 |    100.00
-----------+----------------------+----------
     Total |     1,394      1,091 |     2,485
           |     56.10      43.90 |    100.00

          Pearson chi2(3) =  29.8654   Pr = 0.000
       BMI |          cvd
categories |         0          1 |     Total
-----------+----------------------+----------
       12- |       105        175 |       280
           |     37.50      62.50 |    100.00
-----------+----------------------+----------
       22- |       154        312 |       466
           |     33.05      66.95 |    100.00
-----------+----------------------+----------
       25- |       298        776 |     1,074
           |     27.75      72.25 |    100.00
-----------+----------------------+----------
       30- |       205        460 |       665
           |     30.83      69.17 |    100.00
-----------+----------------------+----------
     Total |       762      1,723 |     2,485
           |     30.66      69.34 |    100.00

          Pearson chi2(3) =  11.7064   Pr = 0.008
```

Foreach: Example 2

Open fup_pot_long1.dta. What values does the variable visit take? In this example we loop through the categories of visit; within each loop we draw a box plot of potval (potassium value) over trt (treatment group) and then carry out a 2-sample t-test. Type and run the following commands.

```
foreach N of numlist 1 3/11 {
graph box potval if visit==`N', over(trt) name(visit`N',replace)
disp
disp "Two-sample t-test of potassium by treat at visit `N'"
disp
ttest potval if visit==`N', by(trt)
}
```

Here the values of visit do not follow a complete arithmetic sequence as potassium was not measured at visit 2. The Stata output for the first few loops is shown below.

```
. foreach N of numlist 1 3/11 {
  2. graph box potval if visit==`N', over(trt)
name(visit`N',replace)
  3. disp
  4. disp "Two-sample t-test of potassium by treat at visit `N'"
  5. disp
  6. ttest potval  if visit==`N' , by(trt)
  7. }
```

```
Two-sample t-test of potassium by treat at visit 1
Two-sample t test with equal variances
------------------------------------------------------------------------------
 Group    |   Obs       Mean    Std. Err.   Std. Dev.  [95% Conf. Interval]
---------+--------------------------------------------------------------------
      1  |  1232    4.32017    .0124056    .4354335   4.295832    4.344509
      2  |  1242    4.329605   .0126359    .445315    4.304815    4.354396
---------+--------------------------------------------------------------------
combined |  2474    4.324907   .0088533    .4403582   4.307546    4.342268
---------+--------------------------------------------------------------------
    diff |          -.009435   .0177094              -.0441617    .0252917
------------------------------------------------------------------------------
    diff = mean(1) - mean(2)                                    t =  -0.5328
Ho: diff = 0                                   degrees of freedom =     2472

    Ha: diff < 0                 Ha: diff != 0                 Ha: diff > 0
 Pr(T < t) = 0.2971         Pr(|T| > |t|) = 0.5942      Pr(T > t) = 0.7029
```

```
Two-sample t-test of potassium by treat at visit 3
Two-sample t test with equal variances
------------------------------------------------------------------------------
 Group    |   Obs       Mean    Std. Err.   Std. Dev.  [95% Conf. Interval]
---------+--------------------------------------------------------------------
      1  |  1206   4.444867    .0143625    .4987732   4.416689    4.473046
      2  |  1219   4.359688    .0135591    .4734041   4.333087     4.38629
---------+--------------------------------------------------------------------
combined |  2425   4.402049    .0099087    .4879487   4.382619     4.42148
---------+--------------------------------------------------------------------
    diff |         .0851791    .0197462               .0464579    .1239002
------------------------------------------------------------------------------
    diff = mean(1) - mean(2)                                    t =   4.3137
Ho: diff = 0                                   degrees of freedom =     2423

    Ha: diff < 0                 Ha: diff != 0                 Ha: diff > 0
Pr(T < t) = 1.0000          Pr(|T| > |t|) = 0.0000      Pr(T > t) = 0.0000
```

8.2 Processing String Variables

Datasets often contain complex string variables, such as identification codes, post-codes, drug names, adverse event narratives etc., which are often considerably more difficult to process than numeric variables. One of Stata's great strengths is its ability to manage string variables. It has many very useful string functions. We will consider just a few: `lower()`, `upper()`, `substr()`, `strpos()`, `length()`, `trim()`.

1. Changing Case

Stata is case sensitive. When it comes to searching for string characters "a" is not equal to "A" and therefore, as long as case is not important, a good starting point when dealing with string variables may be to make the case consistent i.e. change to all lower or all upper case characters. This makes it easier to search for a string of characters. To convert characters to lower case use the `lower()` function and for upper case the `upper()` function. For example:

```
    lower("Body Mass Index") = "body mass index"
    upper("Body Mass Index") = "BODY MASS INDEX"
    proper("BODY MASS INDEX") = "Body Mass Index"
```

2. Substrings

The function `substr()` can be used to decompose a string into parts. The function takes 3 arguments:

- argument 1 is the string or string variable which we are extracting from
- argument 2 is a number which indicates the starting position of the substring within the overall string; this is counted from the first character of the string unless the number is negative when this is taken as counting backwards from the last character.
- argument 3 is a number which indicates the length of the substring; a "." can be used to indicate all characters to the end of the string.

For example:

```
substr("abcde12345",1,3) = "abc"
substr("abcde12345",4,4) = "de12"
substr("abcde12345",-5,2) = "12"
substr("abcde12345",-4,.) = "2345"
```

In each example "abcde12345" is the string we are going to extract from. In the first example, the second and third arguments are 1 and 3 respectively meaning extract the substring starting from character position 1 and of length 3 characters. In the third example the second argument is negative, and therefore the starting position is interpreted as 5 characters from the end of the string. In the fourth example the third argument is given as missing indicating that all characters from the starting position (argument 2) to the end of string.

Substr: Example 1

Open bl_combined2.dta. The patient identifier (ptid) in our datasets is a string variable consisting of a centre id (first five characters) and a subject id (characters 6-9). See below.

```
. list ptid in 1/5
     +-----------+
     |      ptid |
     |-----------|
  1. | C11461001 |
  2. | C11461002 |
  3. | C13511001 |
  4. | C11421001 |
  5. | C11381001 |
     +-----------+
```

We now wish to create a separate variable for centre id (cid) so that we could merge in information at the country level if we wished. We will use the `generate` command and the `substr( )` function to do this.

```
. generate cid = substr(ptid, 1, 5)
```

We have generated a new variable called *cid*, which contains a substring of the variable *ptid*. The substring we have specified starts with the first character of ptid, and is 5 characters in length. It is a good idea to list a few of the old and new variable to check all has gone according to plan.

```
. list ptid cid in 1/5

     +-------------------+
     |      ptid     cid |
     |-------------------|
  1. | C11461001   C1146 |
  2. | C11461002   C1146 |
  3. | C13511001   C1351 |
  4. | C11421001   C1142 |
  5. | C11381001   C1138 |
     +-------------------+
```

3. Searching for a String

The `strpos()` function can be used to search for the occurrence of a string of characters within a longer string. The function takes two arguments:

- argument 1 is the string or string variable which we are searching
- argument 2 is the string of characters we are searching for.

The function returns the starting position of the string of characters if it is found and 0 otherwise. For example:

```
strpos("15mg Aspirin" , "15") = 1
strpos("15mg Aspirin" , "aspirin") = 0
strpos("15mg Aspirin" , "Asp") = 6
```

Note that in the second example Stata returns a 0 since we are searching for the entire string "aspirin" and this is not found. This is a reminder that Stata is case-sensitive and that "a" is not equal to "A". When searching for a string where case is not important it can be helpful to first change the case of all the characters to either lower or upper case using the `lower()` or `upper()` functions as explained above.

Strpos: Example 1

Open bl_meds.dta. In this dataset we have records of medications that were being taken by at baseline by patients in the study. The database allowed for up to 10 medications to be recorded (med1-10).

```
. list ptid med1 med2 med3 in 1/5

     +-----------------------------------------------+
     |      ptid         med1         med2         med3 |
     |-----------------------------------------------|
  1. | C10011001   Paroxetine      Aspirin    Enalapril |
  2. | C10011002     Losartan      Aspirin   Carvedilol |
  3. | C10011003     Losartan     Insuline      Aspirin |
  4. | C10011004    Enalapril   Amiodarone    Thiazides |
  5. | C10011005   Carvedilol    Enalapril   Amiodarone |
     +-----------------------------------------------+
```

Suppose we wish to identify all patients taking aspirin and create a variable called *asp* that is 1 if aspirin is being taken and 0 otherwise. We have a number of issues: the word aspirin may occur alone or with a dose; the string "aspirin" may vary in terms of case (e.g. aspirin or Aspirin); the word aspirin may appear in one of 10 different variables or not at all.

We can deal with the problem of case by use of the `lower()` string function. We can deal with problem of the string "aspirin" occurring at various points within a longer string using the `strpos( )` function. We can loop through each of the variables *med1* to *med10* using the `forvalues` command that we met above.

```
gen asp = 0
forvalues K = 1/10 {
replace med`K' = lower(med`K')
replace asp = strpos(med`K', "aspirin") if asp==0
}
```

The first command creates a variable called *asp* which takes the value 0 for all patients. We then use `forvalues` to create a local macro called K that will enable us to loop through the commands within the brackets. On the first pass through the loop K=1; so the first line within the loop changes med1 to lower case; the second line replaces *asp* with the position of "aspirin" within med1 if it is found or zero if not.

It is important to note the expression `if asp==0`. Suppose within the first loop that "aspirin" is found at string position 1 in *med1* for the patient in row number 1. The variable *asp* (which starts with all values=0) is then replaced with the value 1 in row 1. Now on loop two it is unlikely that the string "aspirin" will be found in med2 for the same patient and therefore `strpos(med2 , "aspirin" )` will evaluate to 0. Since we have already found the string "aspirin" in med1 and set asp=1 we do not want to replace 1 with 0. Hence we only replace *asp* if *asp* is currently equal to 0 i.e. if the string aspirin has not been found in any previous variable.

We should now tabulate the variable asp.

```
. tab asp

        asp |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      1,112       76.11       76.11
          1 |        323       22.11       98.22
          5 |          1        0.07       98.29
          7 |         17        1.16       99.45
          8 |          1        0.07       99.52
         10 |          2        0.14       99.66
         11 |          1        0.07       99.73
         15 |          1        0.07       99.79
         24 |          1        0.07       99.86
         25 |          1        0.07       99.93
         29 |          1        0.07      100.00
------------+-----------------------------------
      Total |      1,461      100.00
```

For 1112 observations no occurrences of the substring "aspirin" were found. Where it was found it was generally at the beginning of the string (asp=1). We should now recode all values greater than 1 to be equal to 1.

```
. recode asp 1/max=1
(asp: 26 changes made)

. tab asp

        asp |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |      1,112       76.11       76.11
          1 |        349       23.89      100.00
------------+-----------------------------------
      Total |      1,461      100.00
```

Save the data in memory as bl_meds1.dta.

4. Converting Numbers to Strings and Vice Versa

From numbers to strings

It can sometimes be useful to convert a numeric variable into a string variable. We can do this in Stata using the `string()` function. This takes a single argument which must be a numeric variable or a number and returns the variable or number as a string. A second optional argument can be used to specify the format.

For example:

```
    string(2014)                = "2014"
    string(23.57 , "%3.1f")     = "23.6"
```

From strings to numbers

The `real()` function performs the opposite task. This converts a string to a real numeric value, provided that the string consists of numeric characters only.

For example:

```
real("15")              = 15
real("20.4")            = 20.4
real("12.5kg")          = .
```

5. How long is a string?

The final string function we will consider now is the `length()` function. This takes a single argument – the string to be evaluated – and returns the length of the string. For examples,

```
length("abcde") = 5
length("WC1E 7HT") = 8  (includes the embedded space)
length(" abc ") = 5 (includes a leading and trailing space)
```

The problem of leading or trailing spaces can be dealt with using the trimming functions:

`ltrim(" s")`	returns "s" with leading blanks removed.
`rtrim("s ")`	returns "s" with trailing blanks removed.
`trim(" s ")`	returns "s" with leading and trailing blanks removed.

There are many other useful string functions. See `help functions` or `help strfun` for more information.

8.3 Dates in Stata

Dates are often recorded as strings e.g. "09/12/1973" or "Sept 7 1965". Dates in this format are not convenient for any data manipulation. For example, if we have recorded a person's date of birth and the current date it would be possible to calculate the number of days or years that person has been alive. Dates that are stored as strings do not allow us to easily carry out such calculations. Stata deals with this by storing dates as elapsed dates (see below) and has functions that enable the user to easily convert string dates into elapsed dates.

Stata stores dates as integers with 0 corresponding to 1 January 1960, 1 to 2 January 1960 and so on. Dates prior to 1 January 1960 are stored as negative integers with –1 corresponding to 31 December 1959 and so on. These are known as *elapsed dates*. Time between dates can therefore be calculated by subtraction. For example, given a person's date of birth and a screening date, the age at screening can be calculated as date of screening minus date of birth. This would give the "number of days old" which could, for example, be divided by 365.25 to convert to years.

Stata has two main functions that can be used along with generate for creating elapsed dates. They are `date()` and `mdy()`. Which to use depends on the format of the variable being translated.

If the variable to be translated takes the form “07/12/1997” or "July 11, 1948" i.e. a single string variable with a standard date structure then we can use the `date()` function. If the date to be translated consists of three different variables i.e. the day, month and year then we would need to use the `mdy()` function.

The Date Function

The syntax for the date function is:

```
generate newdate = date(stringdate, “pattern”)
```

Taking the syntax in order:

- `generate` is the Stata command for creating a new variable
- *newdate* is the name of the elapsed date being created i.e. decided by you
- `date( )` calls Stata’s date function which takes two arguments (in the simplest case).
- *stringdate*, specifies the name of the string variable that is to be translated
- *“pattern”* is the order in which month, day and year occur in the *stringdate*. E.g:
  - for "07-11-1956" pattern would be “DMY”
  - for "June 11, 1962" pattern would be “MDY”
  - for “1965/Sept/07” pattern would be “YMD”
  - for “24-10-78” pattern would be “MD19Y”

The mdy Function

The syntax for the `mdy` function is:

```
generate newdate = mdy(mvar, dvar, yvar)
```

Taking the syntax in order:

- `generate` is the Stata command for creating a new variable
- *newdate* is the name of the elapsed date being created i.e. decided by you
- `mdy( )` is the function which takes three arguments. The three arguments are:
  - *mvar* the variable containing the month
  - *dvar* the variable containing the day
  - *yvar* the variable containing the year

Formatting dates

Once the new date variable has been generated, it can then be formatted so that it appears (i.e. is viewed in the browser) as a date rather than just as a number. For example,

```
format newdate %td              1jul1948
format newdate %tdd_m_cy        1 Jul 1948
format newdate %tdM/D/CY        July/01/1948
```

Elapsed date: Example 1

Open *example_dates.dta* which contains dates in different formats.

```
. list date_1-year_in in 1/5   , noobs

+-------------------------------------------------------------------+
|      date_1      date_2    date_3    date_4  day_in  month_in year_in  |
|-------------------------------------------------------------------|
| 10/Aug/1929  2006/07/03  03-30-06  20060330       2         3    2006  |
|  6/Sep/1920  2009/12/13  04-24-06  20060424      21         3    2006  |
|  6/Dec/1925  2007/04/29  05-05-06  20060505      15         2    2006  |
| 20/Feb/1918  2009/01/16  05-10-06  20060510      16         3    2006  |
| 18/May/1940  2010/11/11  05-10-06  20060510       8         4    2006  |
+-------------------------------------------------------------------+
```

The variable *date_1* is a single string variable which contains the day, month and year separated by forward slashes. We therefore need to use the `date()` function to create an elapsed date. The syntax is:

```
. gen edate_1 = date(date_1, "DMY")
```

The next step is to apply a date format the elasped date, but before we do so we will look at *date_1* and *edate_1.*

```
. list id date_1 edate_1 in 1/5

     +-------------------------------+
     |    id        date_1   edate_1 |
     |-------------------------------|
  1. | P1001   10/Aug/1929    -11101 |
  2. | P1003    6/Sep/1920    -14361 |
  3. | P1004    6/Dec/1925    -12444 |
  4. | P1005   20/Feb/1918    -15290 |
  5. | P1006   18/May/1940     -7167 |
     +-------------------------------+
```

We see here that *edate_1* is just a number. The values here are all less than zero since these dates are all prior to January 1st 1960. We will now apply a date format and list again.

```
. format edate_1 %td

. list id date_1 edate_1 in 1/5

     +--------------------------------+
     |    id        date_1     edate_1 |
     |--------------------------------|
  1. | P1001   10/Aug/1929   10aug1929 |
  2. | P1003    6/Sep/1920   06sep1920 |
  3. | P1004    6/Dec/1925   06dec1925 |
  4. | P1005   20/Feb/1918   20feb1918 |
  5. | P1006   18/May/1940   18may1940 |
     +--------------------------------+
```

The variable edate_1 now displays as a recognisable date, although it is still just a real number. It is good practice to check a few of the dates to make sure the conversion has worked properly. If we use the command codebook on these two variables we will get very different output.

```
. codebook date_1 edate_1
--------------------------------------------------------------------------------
date_1                                                             Date of birth
--------------------------------------------------------------------------------

                  type:  string (str11)

         unique values:  50                       missing "":  0/50

              examples:  "15/Feb/1933"
                         "18/Nov/1934"
                         "24/May/1936"
                         "31/Mar/1933"
--------------------------------------------------------------------------------
edate_1                                                              (unlabeled)
--------------------------------------------------------------------------------

                  type:  numeric daily date (float)

                 range:  [-15290,-5653]               units:  1
       or equivalently:  [20feb1918,10jul1944]        units:  days
         unique values:  50                       missing .:  0/50

                  mean:  -9352.54 = 24may1934 (+ -13 hours)
              std. dev:   2502.23

           percentiles:        10%       25%       50%       75%       90%
                            -13335    -10835     -9023     -7433     -6254
                         29jun1923 03may1930 19apr1935 26aug1939 17nov1942
```

Elapsed date: Example 2

The variable date_3 is a single string variable which contains the month, day and year separated by hyphens. However, note that the year does not include the century i.e. it is in the format MM-DD-YY. As with example 1 we need to use the `date()` function but this time we will need to specify the century. The syntax is:

```
. gen edate_2 = date(date_3, "MD20Y")
. format edate_3 %td
```

As mentioned above it is always worth checking that the date conversion has worked by comparing a few of the original and new dates. Here everything appears to be in order.

```
. list date_3 edate_3 in 1/5

     +-----------------------------+
     |    id     date_3     edate_3 |
     |-----------------------------|
  1. | P1001   03-30-06   30mar2006 |
  2. | P1003   04-24-06   24apr2006 |
  3. | P1004   05-05-06   05may2006 |
  4. | P1005   05-10-06   10may2006 |
  5. | P1006   05-10-06   10may2006 |
     +-----------------------------+
```

Elapsed date: Example 3

The final three columns in example_dates.dta contain the day_in, month_in and year_in i.e. the date_in is recorded in three variables. To translate this into an elapsed date we will need to use the `mdy()` function.

```
. gen edate_in = mdy( month_in, day_in, year_in)
. format edate_in %td
```

Note that the order is important. The function is `mdy()` and the three arguments must appear in that order. If month and day are switched then all the dates will be incorrect (apart from dates where the month and day are the same!) and some missing values will be generated.

Check that the date creation has worked.

```
. list day_in month_in year_in edate_in  in 1/5

     +-----------------------------------------+
     | day_in   month_in   year_in    edate_in |
     |-----------------------------------------|
  1. |      2          3      2006   02mar2006 |
  2. |     21          3      2006   21mar2006 |
  3. |     15          2      2006   15feb2006 |
  4. |     16          3      2006   16mar2006 |
  5. |      8          4      2006   08apr2006 |
     +-----------------------------------------+
```

Elapsed date: Example 4

The variable *date_4* is a numeric variable but is not an elapsed date. For example, consider the first observation 20060330, which we understand means 30th March 2006. Stata understands this as 20,060,330 i.e. as twenty million sixty thousand three hundred and thirty – in elapsed date terms that would actually equate to sometime in the 75th century.

In this case we cannot directly use the `date()` function, because it expects a string variable as input, and we cannot directly use the mdy() function because it expects three separate numeric variables as input.

The easiest way to convert such a variable is to first convert it to a string variable using `tostring`, and then apply the `date()` function to create the elapsed date.

```
. tostring date_4, replace
date_4 was long now str8

. gen edate_4 = date(date_4, "YMD")

. format edate_4 %td
```

Now check that the process has worked correctly.

```
. list date_4 edate_4  in 1/5

     +----------------------+
     |   date_4     edate_4 |
     |----------------------|
  1. | 20060330   30mar2006 |
  2. | 20060424   24apr2006 |
  3. | 20060505   05may2006 |
  4. | 20060510   10may2006 |
  5. | 20060510   10may2006 |
     +----------------------+
```

Other date functions

Other date related functions allow the user to extract the month, day of month, day of year, day of week, quarter of year, etc from an elapsed date.

For example, to extract the day of the week we can use the `dow()` function

```
. gen dow_edate_1=dow(edate_1)

. tab dow_edate_1

dow_edate_1 |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |          6       12.00       12.00
          1 |          5       10.00       22.00
          2 |          4        8.00       30.00
          3 |         12       24.00       54.00
          4 |          6       12.00       66.00
          5 |          8       16.00       82.00
          6 |          9       18.00      100.00
------------+-----------------------------------
      Total |         50      100.00
```

Where 0= Sunday, 1=Monday, through to 6= Saturday.

To extract the month or year we can use the `month()` or `year()` functions.

8.4 Managing Repeated Measures Data

Stata has a number of useful functions and tools for handling repeated measure data. By repeated measures data we mean data where measurements of a variable (or variables) for an individual have been measured on a number of occasions during a study. For example, in a randomised controlled trial each patient's blood pressure may be measured and recorded at baseline and then at each subsequent visit. We may then be interested in looking at the change in blood pressure from baseline or from the previous visit, or the time until blood pressure dropped below a given level or increased by a given amount. In this next section we will explore some of Stata's commands, system variables and other tools for dealing with such data through a series of example tasks.

Repeated measures: Example 1

Open fup_egfr1.dta. In this dataset we have patient id, visit number and date and the egfr value at each visit. We also have the patient's age at entry to the study, their sex and the treatment group. Our aim in this example is to create a variable containing the change from screening in egfr level.

```
. use fup_egfr1 , clear
. sort ptid visit
. list in 1/10

     +----------------------------------------------------------------+
     |      ptid       visit     visdate    egfr   age    sex     trt |
     |----------------------------------------------------------------|
  1. | C10011001   Screening   20jul2006   52.82    63   Male  Active |
  2. | C10011001     Month 5   19dec2006   58.34    63   Male  Active |
  3. | C10011001    Month 13   05sep2007   56.37    63   Male  Active |
  4. | C10011001    Month 21   30apr2008   53.67    63   Male  Active |
  5. | C10011001    Month 29   08jan2009   56.92    63   Male  Active |
     |----------------------------------------------------------------|
  6. | C10011002   Screening   21jul2006   82.65    66   Male Placebo |
  7. | C10011002     Month 5   13dec2006   69.78    66   Male Placebo |
  8. | C10011002    Month 13   23aug2007   82.67    66   Male Placebo |
  9. | C10011002    Month 21   24apr2008   83.89    66   Male Placebo |
 10. | C10011004   Screening   21aug2006   71.49    62   Male  Active |
     |----------------------------------------------------------------|
```

To calculate the change in egfr we will need to use the `bysort` prefix command, understand the Stata *system variables* _n and _N and how to use *subscripts*.

(i) Repeating commands using bysort

We have already met the `bysort` prefix which allows a command to be repeated over subsets of the data. This prefix is particularly useful when it comes to managing repeated measures data where results for a patient are recorded over several rows of a dataset.

Recall that,

```
    bysort varname: command
```

will first sort the dataset in ascending order of *varname* and then repeat the command over each level of *varname.*

Now consider the following command:

```
    bysort varname1 (varname2): command
```

Here we have specified two "`bysort`" variables, but the second of these is enclosed in parentheses. What this will do is sort the whole dataset in ascending order by *varname1* and *varname2*, but only repeat the command over levels of *varname1*.

We will see how this can be useful shortly.

(ii) System Variables _n and _N

Stata has a number of very useful "built-in" variables called system variables. The names of all the system variables start with the underscore character (_). Among the most useful system variables are *_n* and *_N* which are generally referred to as "little-n" and "big-n".

_n	contains the row number in the dataset (as sorted)
_N	contains the total number of rows in the dataset

When combined with `bysort varname`: these then become the row number and total number of rows in the subset of observations within each level of *varname*.

For example, here we use `bysort` and `generate` to create two variables that are equal to _n and _N. We have sorted the dataset by *ptid* and *visdate*, but as *visdate* is in parentheses we only repeat the generate command by *ptid*.

```
. bysort ptid (visdate): generate n = _n
. bysort ptid (visdate): generate N = _N
. . list ptid visit visdate egfr n N in 1/14

     +-------------------------------------------------------+
     |      ptid       visit     visdate    egfr   n   N |
     |-------------------------------------------------------|
  1. | C10011001   Screening   20jul2006   52.82   1   5 |
  2. | C10011001     Month 5   19dec2006   58.34   2   5 |
  3. | C10011001    Month 13   05sep2007   56.37   3   5 |
  4. | C10011001    Month 21   30apr2008   53.67   4   5 |
  5. | C10011001    Month 29   08jan2009   56.92   5   5 |
     |-------------------------------------------------------|
  6. | C10011002   Screening   21jul2006   82.65   1   4 |
  7. | C10011002     Month 5   13dec2006   69.78   2   4 |
  8. | C10011002    Month 13   23aug2007   82.67   3   4 |
  9. | C10011002    Month 21   24apr2008   83.89   4   4 |
 10. | C10011004   Screening   21aug2006   71.49   1   5 |
     |-------------------------------------------------------|
 11. | C10011004     Month 5   01feb2007    85.2   2   5 |
 12. | C10011004    Month 13   27sep2007   92.89   3   5 |
 13. | C10011004    Month 21   05jun2008    64.5   4   5 |
 14. | C10011004    Month 29   21jan2009   70.26   5   5 |
     +-------------------------------------------------------+
```

We've listed the data for the first 3 patients.

Note that within each patient N is constant and indicates the number of rows for each patient; n varies within each patient and indicates the row number within each patient as currently sorted (i.e. by *visdate* within each patient).

Note that for each patient the first visit is where n=1 and the latest visit will be where n=N.

(iii) Subscripts

Specific rows of a dataset can be referred to using subscripts. Subscripts are specified by following a variable name with square brackets containing either a number or an expression. For example, using the display command to illustrate this:

```
. display ptid[1]
```

This is asking Stata to display *ptid* in row 1. Stata displays

```
C10011001
```

When combined with `bysort varname:` the subscript then refers to the row within each level of *varname*. For example, when combined with:

```
    bysort ptid (visdate):

    egfr[1]  =  52.82      for ptid C1011001
    egfr[1]  =  82.65      for ptid C1011002
    egfr[5]  = 70.26       for ptid C1011004
    egfr[5]  = .           for ptid C1011002
```

We now have all the tools we need to generate change in egfr from screening using a single line of Stata code.

```
. bysort ptid (visit): gen double egfr_chbl = egfr - egfr[1]
```

Running through the syntax in order:

- `bysort ptid (visit)` requests that the data is to be sorted by *ptid* and *visit* but that the command to be repeated by *ptid* only
- `gen` is the Stata command for creating a new variable
- `double` requests the new variable is calculated to double precision
- `egfr_chbl` is the user specified name of the new variable
- `egfr - egfr[1]` the new variable is equal to the value of *egfr* minus the value of *egfr* in the first row for each patient.

```
. list ptid visit visdate egfr egfr_chbl n in 1/9

     +------------------------------------------------------------+
     |      ptid       visit     visdate    egfr   egfr_c~l    n |
     |------------------------------------------------------------|
  1. | C10011001   Screening   20jul2006   52.82          0    1 |
  2. | C10011001     Month 5   19dec2006   58.34       5.52    2 |
  3. | C10011001    Month 13   05sep2007   56.37       3.55    3 |
  4. | C10011001    Month 21   30apr2008   53.67        .85    4 |
  5. | C10011001    Month 29   08jan2009   56.92        4.1    5 |
     |------------------------------------------------------------|
  6. | C10011002   Screening   21jul2006   82.65          0    1 |
  7. | C10011002     Month 5   13dec2006   69.78     -12.87    2 |
  8. | C10011002    Month 13   23aug2007   82.67        .02    3 |
  9. | C10011002    Month 21   24apr2008   83.89       1.24    4 |
     +------------------------------------------------------------+
```

Repeated measures: Example 2

We will now create a new variable to indicate the time (in days) since the previous visit. This time we do not want to keep referring to the first row but rather the previous row. We can do this by using the _n variable within the subscript.

```
. bysort ptid (visdate): gen double tsvis = visdate - visdate[_n-1]
(2500 missing values generated)
```

The first part of the syntax is exactly the same as for example 1. The key part here is the final section of code: `visdate - visdate[_n-1]`

Note that the [_n-1] will equate to 0 (1-1) in the first row, 1 (2-1) in the second row, 2 (3-1) in the third row etc. So for the first row for each patient this code equates to:

```
    visdate - visdate[0]
```

Since there is no row 0 for any patient this results in a missing value being generated (hence 2500 missing values generated above). This makes sense since there is no time since the previous visit for the first visit. In the second row for each patient this equates to:

```
    visdate - visdate[1]
```

So for ptid C10011001 in the output above this equates to 19Dec2006 – 20Jul2006.

```
. list ptid visit visdate tsvis n  in 1/9
     +-------------------------------------------------+
     |      ptid       visit     visdate   tsvis    n |
     |-------------------------------------------------|
  1. | C10011001   Screening   20jul2006       .    1 |
  2. | C10011001     Month 5   19dec2006     152    2 |
  3. | C10011001    Month 13   05sep2007     260    3 |
  4. | C10011001    Month 21   30apr2008     238    4 |
  5. | C10011001    Month 29   08jan2009     253    5 |
     |-------------------------------------------------|
  6. | C10011002   Screening   21jul2006       .    1 |
  7. | C10011002     Month 5   13dec2006     145    2 |
  8. | C10011002    Month 13   23aug2007     253    3 |
  9. | C10011002    Month 21   24apr2008     245    4 |
     +-------------------------------------------------+
```

8.5 Creating Summary Datasets

It can often be useful to create a summary dataset from a full dataset, particularly for producing table and figures. Here we will consider the two main Stata commands for creating summary datasets: `contract` and `collapse`.

Contracting Datasets

The command `contract` can be used to replace the dataset in memory with a summary dataset of frequencies.

Contract: Example

Open bl_combined2.dta. We will create a summary dataset containing the number of subjects within each combination of primary endpoint (*pep*), sex and treatment group (*trt*).

```
. use bl_combined2  , clear
. contract trt sex pep

. list, sep(4)

     +----------------------------+
     | pep   trt      sex   _freq |
     |----------------------------|
  1. |   0     1   Female     230 |
  2. |   1     1   Female      51 |
  3. |   0     1     Male     751 |
  4. |   1     1     Male     210 |
     |----------------------------|
  5. |   0     2   Female     201 |
  6. |   1     2   Female      76 |
  7. |   0     2     Male     703 |
  8. |   1     2     Male     278 |
     +----------------------------+
```

The contracted dataset contains the variables *sex*, *pep* and *trt* and a new variable called *_freq* containing the frequencies. Note that all other variables will be dropped.

As there are 2 levels of *sex*, *pep* and *trt* we have just 8 observations in the contracted dataset. It is possible to analyse datasets such as this using Stata's frequency weight facility. For example:

```
. tab trt pep [fw=_freq], chi row nokey

           |          pep
       trt |         0          1 |     Total
-----------+----------------------+----------
         1 |       981        261 |     1,242
           |     78.99      21.01 |    100.00
-----------+----------------------+----------
         2 |       904        354 |     1,258
           |     71.86      28.14 |    100.00
-----------+----------------------+----------
     Total |     1,885        615 |     2,500
           |     75.40      24.60 |    100.00

          Pearson chi2(1) =  17.1071   Pr = 0.000
```

Note that the command contract overwrites the dataset currently in memory including any changes that have been made since last save. It is very important therefore to think carefully before using this command and to make sure all changes have been saved. Of course if you have been making all your changes using commands which have been saved in a do-file this is not an issue. The contracted file should be saved with a new name.

Collapsing Dataset

The command `collapse` replaces the dataset in memory with a dataset of statistics e.g. means, SDs, counts, medians, sums, etc. For a complete list of the statistics available see `help collapse`. The syntax for this command is best explained by an example.

Collapse: Example

Open *fup_egfr1.dta*. Suppose we want to create a summary dataset containing mean egfr at each visit by treatment group. We also wish to save the number of measurements and the standard deviation. The syntax for this is:

```
collapse (mean) m_egfr = egfr (sd) sd_egfr = egfr (count) n_egfr =
egfr, by(visit trt)
```

Running through the syntax in order:

- `collapse` is the Stata command

There then follows three sets of statistics (we will run through first two):

▪ `(mean)`	indicates that the first statistic we want is the mean
▪ `m_egfr`	specifies the name of the new variable to be *m_egfr*
▪ `=egfr`	indicates that it is the mean of *egfr* that we want to calculate
▪ `(sd)`	indicates that second statistic we want is the standard deviation
▪ `sd_egfr`	specifies the new variable name for the second statistic
▪ `=egfr`	indicates that it is the SD of *egfr* that we want to calculate

- `by(visit  trt)` indicates that we want all the statistics calculated for each combination of *visit* and *trt*

On the next page we list the collapsed dataset.

Note there is no “uncollapse” command. So if you are working interactively make sure any prior changes have been saved before collapsing a datatset – or make sure you are working within a do-file. The collapsed dataset should be saved with an appropriate name.

```
. list

     +-------------------------------------------------------+
     |     visit       trt      m_egfr     sd_egfr   n_egfr |
     |-------------------------------------------------------|
  1. | Screening   Placebo   70.343079   21.883258     1251 |
  2. | Screening    Active   71.067814   21.908973     1240 |
     |-------------------------------------------------------|
  3. |   Month 5   Placebo   71.263944   22.619153     1040 |
  4. |   Month 5    Active   68.680884   22.055378     1052 |
     |-------------------------------------------------------|
  5. |  Month 13   Placebo   71.076831   21.192192      830 |
  6. |  Month 13    Active   69.179988   22.900335      838 |
     |-------------------------------------------------------|
  7. |  Month 21   Placebo   70.973732   23.704579      613 |
  8. |  Month 21    Active   68.300883     22.3774      634 |
     |-------------------------------------------------------|
  9. |  Month 29   Placebo   69.799057   20.891014      428 |
 10. |  Month 29    Active    70.07884   22.528182      457 |
     +-------------------------------------------------------+
```

8.6 Reshaping data

The **reshape** command converts data from wide to long format or from long to wide format. It is particularly used when dealing with datasets containing repeated measurements e.g. data from a clinical trial with measurements taken at 3-monthly follow-up visits. Below we give hypothetical examples of long format and wide format datasets.

Long format data is where the dataset contains multiple rows per subject, each row corresponding to a repeated measurement. The dataset will generally contain variables indicating patient id and the visit number as well as the variable(s) being measured each time e.g. blood pressure, total cholesterol etc. The table below gives an example of a long format dataset for just two patients each with measurements of systolic blood pressure and total cholesterol at four visits.

patid	*sex*	*visit*	*sbp*	*tchol*
1	Male	1	120	6.1
1	Male	2	124	6.2
1	Male	3	124	6.2
1	Male	4	125	6.0
2	Female	1	135	7.2
2	Female	2	130	7.4
2	Female	3	140	7.4
2	Female	4	145	7.6

Wide format data is where the dataset contains one row per subjects, with multiple variables containing the measurements made at repeated visits. Generally the dataset will contain variables with names made up of a stub indicating the parameter being measured (e.g. sbp, tchol) and a subscript indicating the visit number (e.g. sbp1, sbp2, sbp3). The table below gives an example of a wide format dataset for just two patients each with measurements of systolic blood pressure and total cholesterol at four visits.

patid	*sex*	*sbp1*	*sbp2*	*sbp3*	*sbp4*	*tchol1*	*tchol2*	*tchol3*	*tchol4*
1	Male	120	124	124	125	6.1	6.2	6.2	6.0
2	Female	135	130	140	145	7.2	7.4	7.4	7.6

Note that these two examples contain the same information. In the long format dataset we have a variable called visit indicating the visit number, whereas in the wide format dataset the information about visit is contained in the subscript to the variable names. In long format the variable *sex* is constant within *patid*.

Sometimes it is more convenient to have the data in wide format and at other times it is more convenient to have the data in long format. The `reshape` command allows you to move between the two formats. We demonstrate the command with some examples.

Reshape: Example 1 - Moving from long to wide format

We will use the dataset *fup_egfr1.dta* to demonstrate moving from long to wide format. The dataset contains repeated measurements of egfr levels by visits.

```
. use fup_egfr1, clear
. list in 1/5

     +---------------------------------------------------------------+
     |      ptid       visit     visdate    egfr   age    sex     trt |
     |---------------------------------------------------------------|
  1. | C10011001   Screening   20jul2006   52.82    63   Male  Active |
  2. | C10011001     Month 5   19dec2006   58.34    63   Male  Active |
  3. | C10011001    Month 13   05sep2007   56.37    63   Male  Active |
  4. | C10011001    Month 21   30apr2008   53.67    63   Male  Active |
  5. | C10011001    Month 29   08jan2009   56.92    63   Male  Active |
     +---------------------------------------------------------------+
```

Note that id number C10011001 occupies 5 rows of the dataset, corresponding to 5 separate visits. The variable *visit* contains the visit number and the variable *egfr* contains the egfr level measured at each visit. Note that the variables *age*, *sex* and *trt* are constant within *ptid* whereas the variables *visit, visdate and egfr* are not.

```
. reshape wide visdate egfr , i(ptid) j(visit)
(note: j = 1 5 7 9 11)

Data                               long   ->   wide
---------------------------------------------------------------------------
Number of obs.                     8383   ->    2497
Number of variables                   7   ->      14
j variable (5 values)             visit   ->   (dropped)
xij variables:
                                visdate   ->   visdate1 visdate5 ...
visdate11
                                   egfr   ->   egfr1 egfr5 ... egfr11
---------------------------------------------------------------------------
```

Running through the syntax in order:

- `reshape` is the Stata command
- `wide` indicates that we are moving from long to wide format
- `visdate egfr` are the variables that change over visit
- `i(ptid)` is the unique subject identifier indicating which observations go together
- `j(visit)` is the name of the variable that indicates the repeat visits; the values of this variable will form the suffixes on the new variables.

The Stata output indicates we have moved from 8383 to 2497 observations and from 7 to 14 variables. The j variable which had 5 values is dropped and the variable value is now suffixed with values 1, 5, 7, 9 and 11 to indicate which visit number they represent.

Below we list the first 2 rows of the new dataset.

```
. list ptid trt sex egfr1 visdate1 egfr5 visdate5   in 1/2

   +-------------------------------------------------------------------+
   |      ptid       trt    sex   egfr1    visdate1   egfr5    visdate5 |
   |-------------------------------------------------------------------|
1. | C10011001    Active   Male   52.82   20jul2006   58.34   19dec2006 |
2. | C10011002   Placebo   Male   82.65   21jul2006   69.78   13dec2006 |
   +-------------------------------------------------------------------+
```

Compare this listing of the data in wide format to that above in long format. We should now save this dataset; it is good practice to add the suffix *_wide* to the filename, and also indicate what it contains. So here *egfr_wide.dta* would be a good choice.

Reshape: Example 2 - moving from wide to long format

We will use the dataset *pot_wide.dta* to demonstrate moving from long to wide format.

```
. list ptid age trt sex potval1 pot_dt1 potval3 pot_dt3  in  1/5

   +-------------------------------------------------------------------------+
   |      ptid   age   trt    sex   potval1     pot_dt1   potval3     pot_dt3 |
   |-------------------------------------------------------------------------|
1. | C10011001    63     1   Male       4.9   20jul2006       4.9   28jul2006 |
2. | C10011002    66     2   Male       3.9   21jul2006       3.9   27jul2006 |
3. | C10011004    62     1   Male       4.7   21aug2006       4.6   29aug2006 |
4. | C10011005    66     1   Male       4.9   22aug2006       4.5   30aug2006 |
5. | C10011006    68     1   Male       4.7   29aug2006       4.6   05sep2006 |
   +-------------------------------------------------------------------------+
```

Note that the variables containing the potassium measurement at each visit all have the *same stub* (i.e. potval) and that the visit number is indicated by the suffix (i.e. 1, 3, 4, 5). The suffixes do not need to be consecutive. The variables sex, age and trt are only recorded once for each patient, but the visit date and potassium measurement are recorded for each visit.

To reshape from wide to long format.

```
reshape long  potval pot_dt  , i(ptid) j(visit)
(note: j = 1 3 4 5 6 7 8 9 10)

Data                               wide   ->   long
-----------------------------------------------------------------------------
Number of obs.                     2497   ->   22473
Number of variables                  22   ->       7
j variable (9 values)                     ->   visit
xij variables:
           potval1 potval3 ... potval10   ->   potval
           pot_dt1 pot_dt3 ... pot_dt10   ->   pot_dt
-----------------------------------------------------------------------------
```

Running through the syntax in order:

- `reshape` is the Stata command
- `long` indicates that we are moving from long to wide format
- `potval pot_dt` are the stubs for the variables that change over visit
- `i(ptid)` is the unique subject identifier indicating which observations go together
- `j(visit)` is the name of the new variable that will be created to indicate the repeat visits; the values of this variable will come from the suffixes of *potval* and *pot_dt*.

The Stata output indicates we have moved from 2497 to 22473 observations and from 22 to 7 variables. The j variable is called visit and takes on 9 values. All the information contained in the variables *potval1-potval10* is now held in the *potval*.

```
. list in 1/9

     +-----------------------------------------------------------------+
     |      ptid       visit   potval      pot_dt   age   trt     sex |
     |-----------------------------------------------------------------|
  1. | C10011001   Screening      4.9   20jul2006    63     1    Male |
  2. | C10011001      Week 1      4.9   28jul2006    63     1    Male |
  3. | C10011001      Week 4      4.6   17aug2006    63     1    Male |
  4. | C10011001     Month 5      4.7   19dec2006    63     1    Male |
  5. | C10011001     Month 9      4.6   26apr2007    63     1    Male |
     |-----------------------------------------------------------------|
  6. | C10011001    Month 13      4.5   05sep2007    63     1    Male |
  7. | C10011001    Month 17      4.2   09jan2008    63     1    Male |
  8. | C10011001    Month 21      4.5   30apr2008    63     1    Male |
  9. | C10011001    Month 25      4.5   03sep2008    63     1    Male |
     +-----------------------------------------------------------------+
```

Compare this listing of the dataset in long format to that above in wide format and make sure you understand how they relate. Save this dataset as *pot_long.dta*.