Lesson 1: Getting Started

This lesson is a gentle introduction to Stata, including why it's used, its interface, do files (the script files Stata uses), how to load in and save data, how to make use of the help system, setting the working directory, using the data browser and editor, and importing, exporting, appending and merging data.

I use Stata version 14.2 in the videos, but the overall look and the basic programs in Stata haven't changed markedly since Stata 12.

1.1 Why Use Stata?

Stata is a general-purpose statistical package, with the following advantages:

- It is quick
- It handles data very well
- The graphics facilities are (reasonably) good
- Users can write new commands, so Stata can be and is constantly updated
- Writing code for analyses is intuitive and (relatively) easy to understand
    - Compared to R and Python, I prefer writing code in Stata, and the Stata help files are usually much better than the equivalent R and Python help/man pages

Stata is best used by typing commands in the command window or, even better, in Stata's text editor, which is called a do file (see **Section 1.3**). In do files, you can write and execute several commands individually or together. Data manipulation and analysis is carried out through these commands, so it's simple to repeat an analysis if the commands are saved in a do file. Stata can also be used in a point-and-click way through using the drop-down menus at the top of the screen, but I won't be using these in the lessons and don't recommend using them in practice: writing out script and recording it in a do file is replicable, and I think probably the fastest way of learning how to use Stata, even if it's more difficult to start with.

Throughout this workbook, commands are written in **`bold courier font`**, the same font used in Stata. For example, most Stata commands follow the same format:

```
command {variable(s)} if … in …, [options]
```

where:

- **`command`** is the name of the command to be used, e.g. **`summarize`**
- **`{variable(s)}`** is a list of one or more variables, e.g. age
- **`if`** … restricts the command to a selection of observations that fulfil a criterion, e.g. participants that are over 40 years of age
- **`in`** … restricts the command to a selection of observations based on their order, e.g. the first 50 observations
- **`, [options]`** - any options a command can use, always typed after a comma

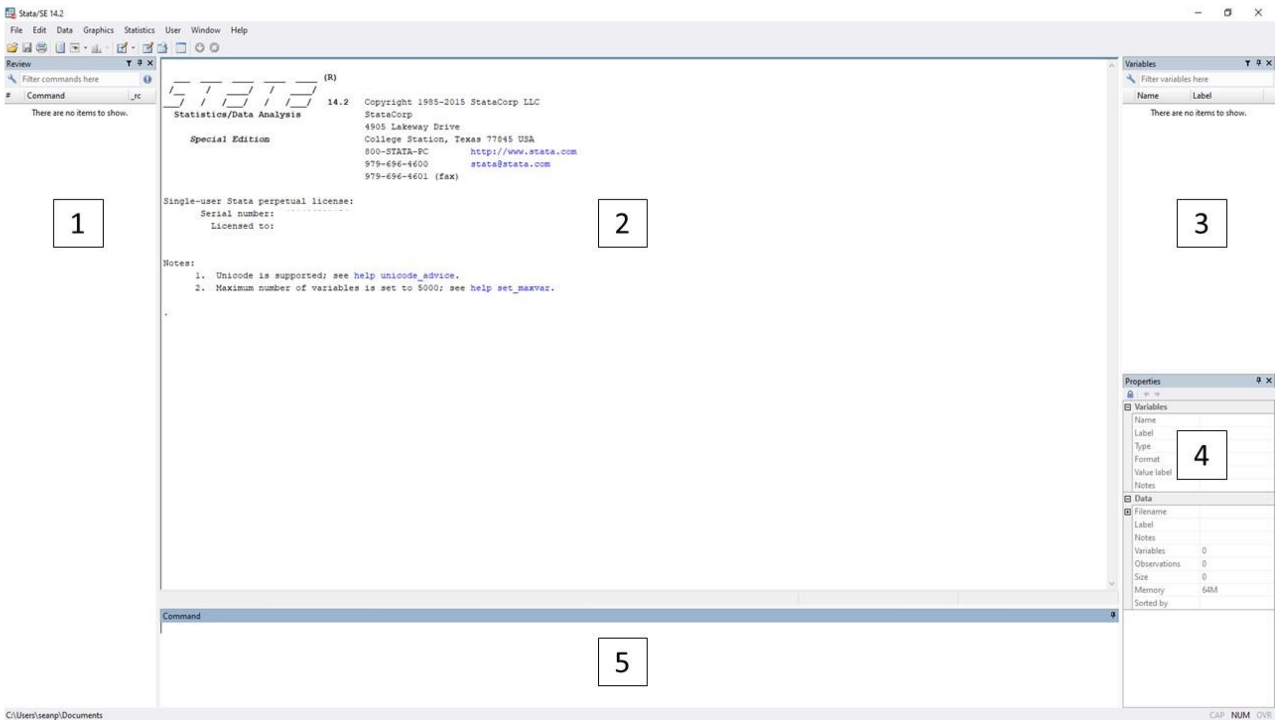
Most commands require just the command name and a list of one or more variables, though some commands either require or allow you to specify some options too. Commands can usually be shortened to save time, for example the **`tabulate`** command (wee use this later) can be shortened to **`tab`**. After each command is introduced, I'll put the short form (if any) I typically use as well, and after the first example, I will use the shortened form.

During this course, we'll go through plenty of examples of useful commands.

1.2 Stata's Interface

Stata opens with five main windows, and each window gives us different information:

1. Review: shows all commands that have been run in this Stata session
2. Results: displays the output of commands
3. Variables: displays a list of all variables in a dataset, as well as their labels
4. Properties: shows summary information about the current dataset
5. Command: you enter commands here



```
  ___  ____  ____  ____  ____ (R)
 /__    /   ____/   /   ____/
___/   /   /___/   /   /___/   14.2   Copyright 1985-2015 StataCorp LLC
  Statistics/Data Analysis            StataCorp
                                      4905 Lakeway Drive
     Special Edition                  College Station, Texas 77845 USA
                                      800-STATA-PC        http://www.stata.com
                                      979-696-4600        stata@stata.com
                                      979-696-4601 (fax)

Single-user Stata perpetual license:
       Serial number:
         Licensed to:

Notes:
      1.  Unicode is supported; see help unicode_advice.
      2.  Maximum number of variables is set to 5000; see help set_maxvar.

.
```

1.3 Do Files

Stata's text editor, a do file, is opened by clicking this button (below the **Graphics** and **Statistics** menus on my screen):

Do files allow you to write out and save all the commands you want to run. This is extremely useful for keeping track of commands, and for editing and re-running analyses.

To execute the commands in do files, highlight the line or block or lines that you want to run and press **ctrl+d**, or click this button in the do file editor:

If you are following along with the videos, feel free to either type commands into the command window (number 5 above – this is what I'll be doing to save switching between windows), or type the commands into the do file and run them individually (this is how I work in practice). I recommend using a do file so you can keep a copy of all the commands we'll use, though I've provided do files with all the commands we'll use in lessons.

In a do file, every command needs to be on a separate line. If you want to extend the command across multiple lines, you need to type `///` at the end of each line where you want to continue to the next line.

Additionally, if you want to insert a comment into a do file (a line of text that isn't run as code, which appears green in newer versions of Stata), then start a new line with an asterisk (`*`). If you want a whole section to be comments (or just not run), then start a new line with `/*` above where you want to comment to start, and `*/` on a new line just below where you want the comment to end. Finally, if you want to add a comment at the end of a command, type in `//` and any following text will be a comment.

1.4 The Dataset

I've created a simulated dataset that we'll be looking at in this lesson, **Lesson_01.dta**, which contains information for the following variables:

1. ID number
2. Age (years)
3. Sex (male or female)
4. Height (cm)
5. Weight (pounds)
6. Hair colour (black, blonde, brown, grey, red)
7. Systolic blood pressure (mmHg)
8. Diastolic blood pressure (mmHg)
9. Accommodation (own, mortgage, rent)
10. Exercise per week (numeric categories)
11. Calories consumed per day
12. Income (£s)
13. Run a marathon in the past year (yes or no)
14. Current smoker (yes or no)
15. Favourite colours
16. Date of blood pressure measurement
17. Date of recruitment

I've made it so that the dataset needs some editing before it can be analysed, which we'll cover in the next couple of lessons.

1.5 Loading and Saving Data

There are many ways to load a Stata data file (a **.dta** file); one of the simplest ways is to click the "open" icon as for any other program. This loads the dataset into Stata memory, and displays the command Stata used to do this:

```
use "{folder}\Lesson_01.dta", clear
```

You could have typed in the command directly, either into the command window or a do file. You can also load in a dataset by dragging and dropping a Stata data file directly into the results window. You probably won't see the option I've included above, **`clear`**, as this only appears automatically when you are loading in data on top of existing data that isn't saved. We'll go through more about options and Stata trying to stop you losing data in a bit. For now, bear in mind that *Stata has no undo button*: changes are permanent, which is presumably why Stata is quite keen for you to not accidentally overwrite or delete data. Also bear in mind that when typing folders or file names, you'll likely have to enclose the locations in quotation marks.

Once loaded into memory, you will see all the variables in the dataset in the upper right-hand window. The lower right-hand window displays some information about the dataset, for example that it has **16 variables** and **1000 observations**, i.e. 16 columns of data and 1000 rows, excluding the names of the variables.

We are now going to save a copy of the dataset to work with: in general, it is safest to always keep a master, unchanged copy of a dataset, and work on a copy. Because you can save the commands you use to clean and analyse a dataset in a do file, you don't need to change the master copy at all. This means that if you make a mistake, then you can go back and just fix the do file, then re-run all the code.

As with opening a dataset, there are a couple of options for saving a dataset. You can either use the "save" icon and choose a folder to save in, or use the **`save`** command:

```
save "{folder}\Lesson_01_{initials}.dta", replace
```

Anything in curly brackets in this workbook needs to be replaced by something else, in this case your initials. The option **`replace`** allows Stata to overwrite any saved Stata files with the same name in the same folder. It is optional, but Stata will give an error message if any file exists with the same name. If you don't specify the folder location, Stata will save to the folder acting as the *working directory* (see **Section 1.7**).

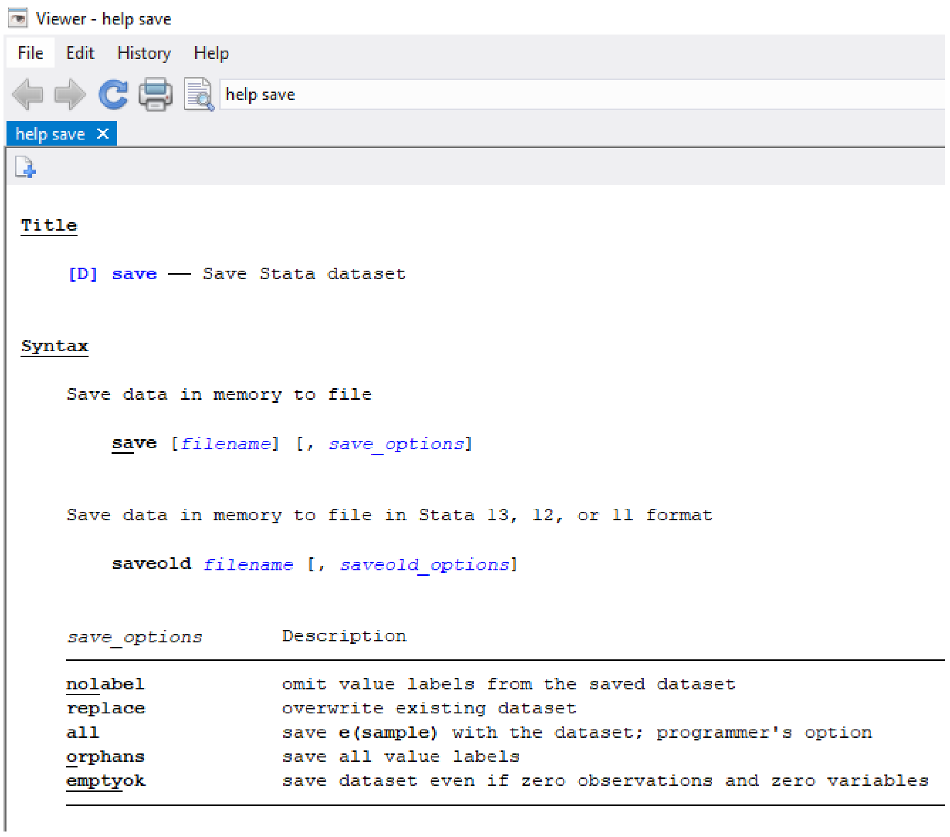
1.6 Help

Stata has an extensive help system, where every command has its own help page. General help pages for different topics are also available. We've now seen the commands **`use`** and **`save`**, so could look up the help files for each. For any command, just type:

```
help {command}
```

where **`{command}`** is the name of the command you want to look up. Let's try:

```
help save
```



```
Title

    [D] save — Save Stata dataset


Syntax

    Save data in memory to file

        save [filename] [, save_options]


    Save data in memory to file in Stata 13, 12, or 11 format

        saveold filename [, saveold_options]


    save_options          Description
    -------------------------------------------------------------------------
    nolabel               omit value labels from the saved dataset
    replace               overwrite existing dataset
    all                   save e(sample) with the dataset; programmer's option
    orphans               save all value labels
    emptyok               save dataset even if zero observations and zero variables
    -------------------------------------------------------------------------
```

This brings up the help file for the **`save`** command. The **`Syntax`** section details how the command needs to be written to be used. Stata syntax (the grammar of Stata) differentiates between bits of the command that are necessary and those that are optional: parts of the command in square brackets are optional. Here, you can see the **`[filename]`** and **`[, save_options]`** are optional because they are in square brackets, so you could just type **`save`** as a valid command.

The help files will always list the syntax of the command, detail any options, and usually give more information and some examples. However, not all commands are written by Stata, and for user-written commands, help files can be of variable quality. In my experience, the help files for almost all programs are pretty good though.

1.7 The Working Directory

The *working directory* of Stata is the folder in which Stata operates – you can see in the bottom left of the screen where Stata is currently operating. This is the folder Stata looks to load and save datasets if you don't provide a full file path, meaning you can save some time and effort setting your working directory at the top of a do file or start of a Stata session, and only specifying file names when loading and saving data.

The necessary command for setting the working directory is:

```
cd "{folder}"
```

The folder location can either be typed out, or by selecting the *copy address as text* option when right clicking on the address bar of the folder you want Stata to work in, or by copying the folder location from the **`use`** command we used earlier.

The command **`cd`** stands for *change directory*.

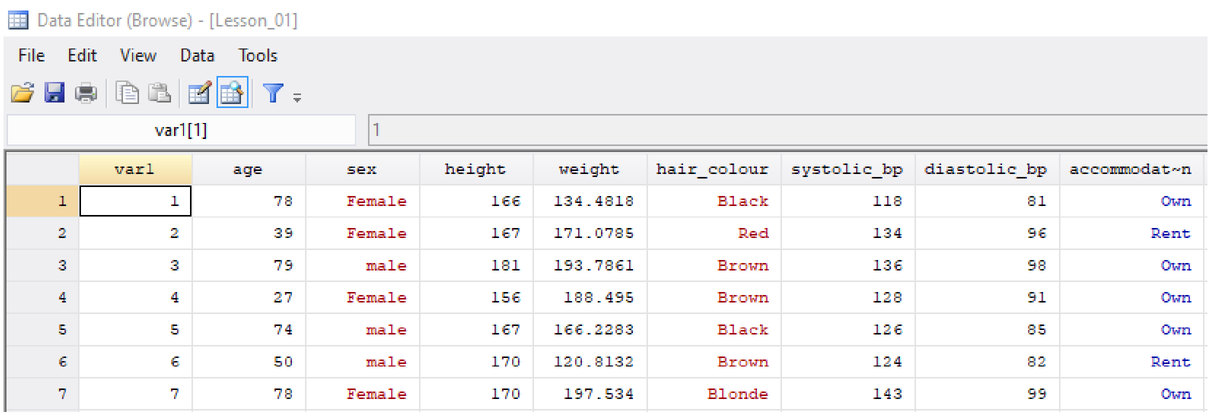
1.8 Looking at Your Data

A Stata dataset can be thought of as a spreadsheet, containing some number of columns (variables) and rows (observations). Each variable has a unique name which can be used to specify the variable in commands. There are rules about variable names:

- Variables have to start with a letter or an underscore
- Variables can't have more than 32 characters
- Variable names can't be used more than once
- Variable names can't have spaces or symbols in them (except an underscore)

Stata is also *case-sensitive*. This means that Stata will regard, for example, **Age** and **age** as two unique and completely separate variables. For this reason, it is usually recommended to keep all variable names in lower case (it's also easier to write in code if it doesn't need capitals).

The data can be directly viewed using the **Data Editor** buttons (below the **Statistics** and **User** menus on my screen): The left-hand button is the **`edit`** button, and the right-hand button is the **`browse`** button. Both buttons bring up a spreadsheet view of the dataset, but you can directly edit data using the **`edit`** button. This isn't usually recommended, as it's easy to make mistakes directly editing data and it isn't replicable, but there are situations where it's useful. Although the commands don't come up when you click these buttons, you can type **`edit`** and **`browse`** into the command window to bring up the same spreadsheet views.



	var1	age	sex	height	weight	hair_colour	systolic_bp	diastolic_bp	accommodat~n
1	1	78	Female	166	134.4818	Black	118	81	Own
2	2	39	Female	167	171.0785	Red	134	96	Rent
3	3	79	male	181	193.7861	Brown	136	98	Own
4	4	27	Female	156	188.495	Brown	128	91	Own
5	5	74	male	167	166.2283	Black	126	85	Own
6	6	50	male	170	120.8132	Brown	124	82	Rent
7	7	78	Female	170	197.534	Blonde	143	99	Own

You'll notice that the values in most variables are black, while some values are red and some blue. These correspond to different data types: black values are numeric (i.e. composed entirely of numbers and missing values), red values are strings (i.e. text or characters, e.g. "male", "brown", "Blue; Yellow; Red"), and blue values are labelled numbers (e.g. 1 = "Own", 2 = "Mortgage", 3 = "Rent" etc.). This distinction is made for each variable and is set once the data is created or loaded into Stata. You can't add strings to numeric variables, or numbers to string variables (unless they are put in quotation marks to become strings). This means it's really important to know what kind of variable you're working with at any time, since numbers and strings are dealt with entirely differently: the general rule is that strings need to be enclosed with quotation marks, otherwise Stata gets confused. Labelled numeric variables act as numbers, but are labelled, so whenever you look at them (or use a command to summarise information about the variable), you see something more informative than simple numbers. This is especially useful for binary and categorical variables, i.e. variables with 2 or more possible options.

You'll notice full stops (periods) in the numeric variables – these are missing values. Missing values are coded in Stata as incredibly large numbers, but this doesn't come up much apart from in the context of *if statements*, which we deal with in **Lesson 3**. All you need to remember for now is that if you see a full stop in a numeric variable, it's a missing numeric value. Missing string values, on the other hand, are literally blank.

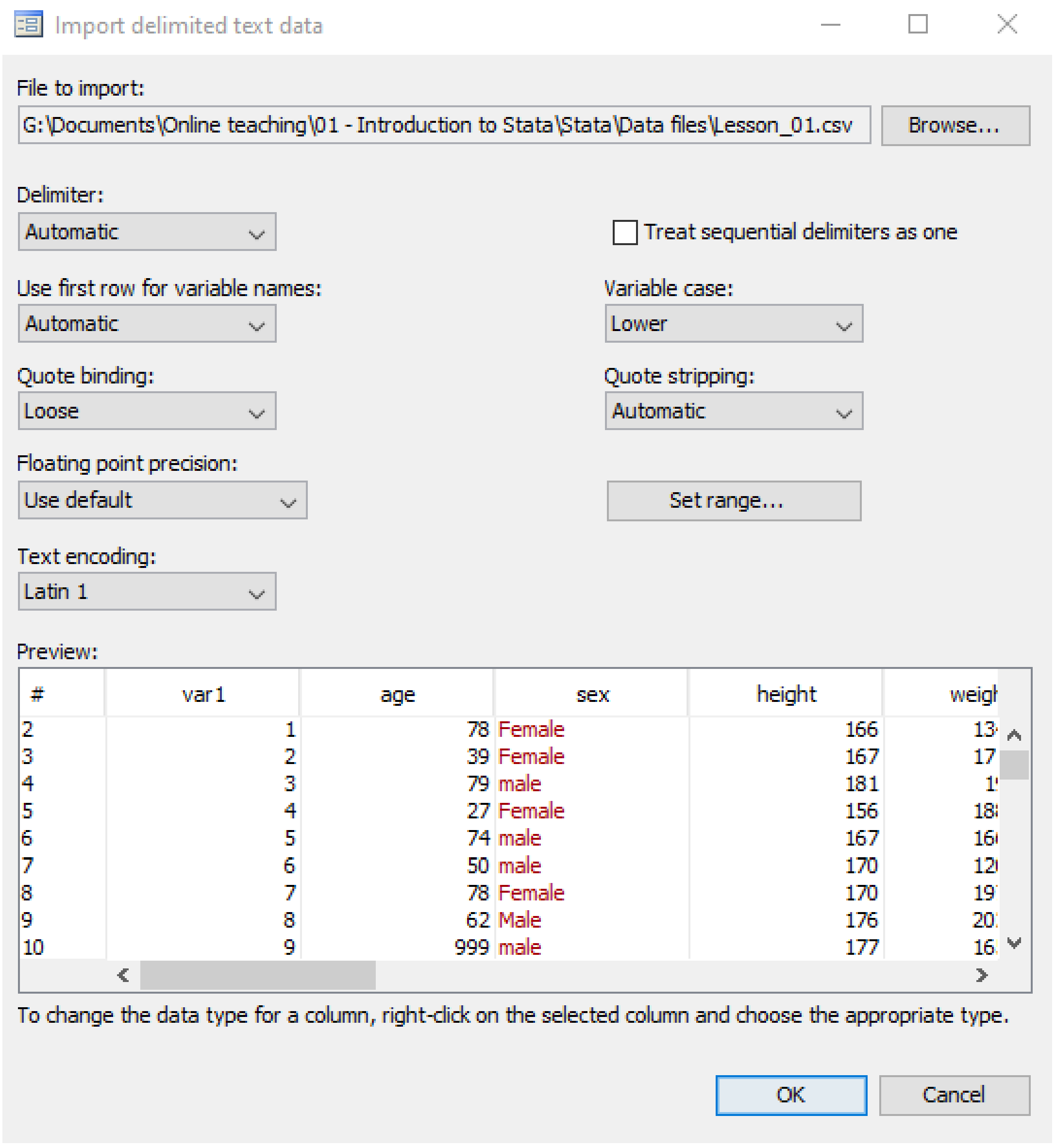
1.9 Importing, Exporting, Appending and Merging Data

Importing data

In addition to opening .dta files, Stata can import other types of data files too, for example Excel worksheets and .csv (comma-separated values) files.

The **`import`** command can be used to import files, though the first time you import a new file, it is often simpler to use the ***Import*** option under **File**. From there, you'll need to select the correct import option, usually either an ***Excel spreadsheet*** or a ***Text data*** file This brings up an import window that has a preview of what the data will look like once imported. Stata will attempt to import the data correctly automatically, but you may need to change some of the options to get the data to look right. Once you've correctly imported the file you want, you can copy the command into a do file for later use, or save the data as a .dta file and use that instead.

For example, if we wanted to import the **Lesson_01.csv** file, we could use the ***Import*** option under **File:**



File to import: G:\Documents\Online teaching\01 - Introduction to Stata\Stata\Data files\Lesson_01.csv

Delimiter: Automatic

Treat sequential delimiters as one

Use first row for variable names: Automatic

Variable case: Lower

Quote binding: Loose

Quote stripping: Automatic

Floating point precision: Use default

Text encoding: Latin 1

Preview:

#	var1	age	sex	height	weigl
2	1	78	Female	166	13
3	2	39	Female	167	17
4	3	79	male	181	1
5	4	27	Female	156	18
6	5	74	male	167	16
7	6	50	male	170	12
8	7	78	Female	170	19
9	8	62	Male	176	20
10	9	999	male	177	16

To change the data type for a column, right-click on the selected column and choose the appropriate type.

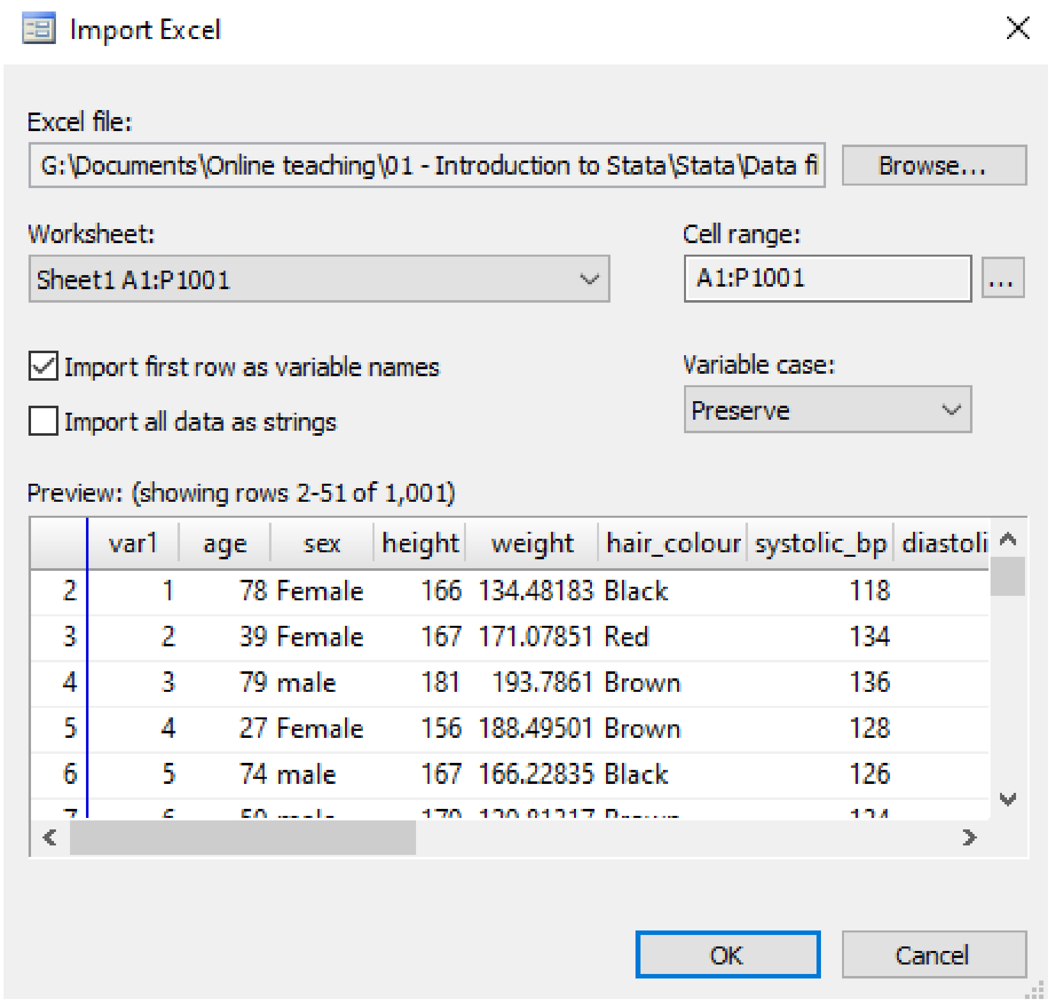
This gives us the following command:

```
import delimited "G:\Documents\Online teaching\01 -
Introduction to Stata\Stata\Lesson_01.csv", clear
(16 vars, 1,000 obs)
```

If we've set the working directory, we could remove the folder location and just write:

```
import delimited "Lesson_01.csv", clear
(16 vars, 1,000 obs)
```

Importing from Excel is similar, though we also need to specify that we want a particular Excel sheet (note the quotation marks in the command) and that we want the first row of observations to be treated as the variable names:



Import Excel

Excel file: G:\Documents\Online teaching\01 - Introduction to Stata\Stata\Data fi

Worksheet: Sheet1 A1:P1001

Cell range: A1:P1001

☑ Import first row as variable names

☐ Import all data as strings

Variable case: Preserve

Preview: (showing rows 2-51 of 1,001)

	var1	age	sex	height	weight	hair_colour	systolic_bp	diastoli
2	1	78	Female	166	134.48183	Black	118	
3	2	39	Female	167	171.07851	Red	134	
4	3	79	male	181	193.7861	Brown	136	
5	4	27	Female	156	188.49501	Brown	128	
6	5	74	male	167	166.22835	Black	126	

This gives us the following command:

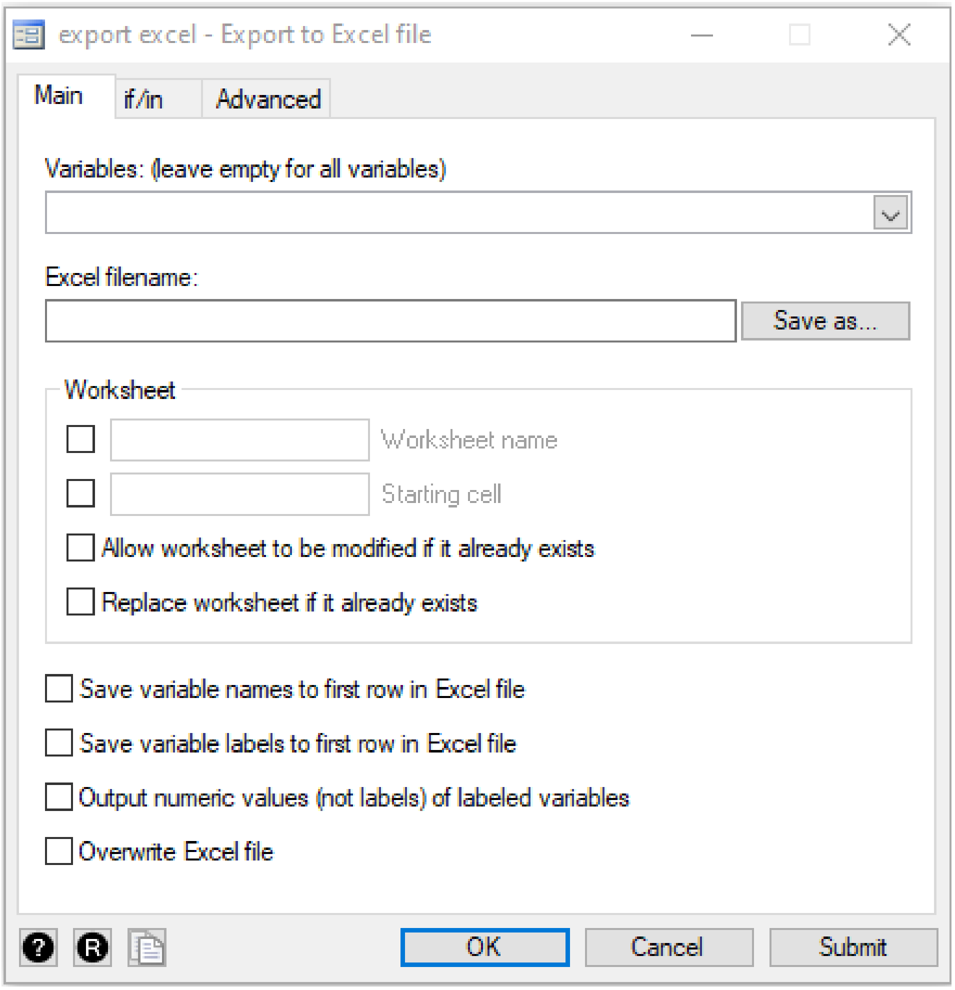
```
import excel "Lesson_01.xlsx", sheet("Sheet1") firstrow clear
```

We could also specify that we wanted a particular cell range, or that we wanted all variables to be imported as strings – this would convert all the numbers to strings.

Exporting data

The **`export`** command can be used to export data to Excel, tab delimited, comma delimited, and other file types. It works similarly to **`import`**, but in reverse. As with **`import`**, the first time you export a file, it makes sense to use the ***Export*** option under **File**, tinkering until you output the data in a format that you want.

In particular, you can select which variables to export (leave blank to export all variables), whether you want variable names or variable labels in the first row and whether you want the data *labels* or the underlying *numbers* (for numeric labelled variables). There are also more options to play around with if you're looking for something specific.



Variables: (leave empty for all variables)

Excel filename:

Worksheet

☐ Worksheet name

☐ Starting cell

☐ Allow worksheet to be modified if it already exists

☐ Replace worksheet if it already exists

☐ Save variable names to first row in Excel file

☐ Save variable labels to first row in Excel file

☐ Output numeric values (not labels) of labeled variables

☐ Overwrite Excel file

Appending data

If you have multiple datasets that have the same variable names and variable types (string or numeric), you can attach them one on top of the other by *appending* them. The syntax is:

```
append using "{new dataset}.dta"
```

`append` doesn't require the number of variables in the two datasets to be the same: any variables missing in the original or appended datasets will have missing values created for them. For example, if the *age* variable is in the original but not the appended dataset, every observation in the appended dataset will have missing values of *age*.

Merging data

While **`append`** adds observations to a dataset in memory, **`merge`** adds new variables to the same observations to a dataset in memory (although sometimes it adds rows too). This is useful if you have multiple datasets that contain *different* variables for the *same* observations, for example, a dataset containing demographic information about people in a study (age, sex, weight etc.), and another containing health outcomes (blood pressure, disease status etc.).

The **`merge`** command usually requires that there be at least one variable that is common to both datasets, for example an ID variable. You can have more than one matching variable, and you can also match on row number instead of a variable if you know the rows are sorted the same way in both datasets (though this is inherently riskier). The observations in the second dataset are matched to the first by the identification variable(s) or row number.

Merges can be *one-to-one*, where unique IDs in each dataset are tied together, *one-to-many*, where unique IDs in the dataset in memory are tied to potentially multiple instances of each ID in the second dataset, *many-to-one*, where potentially multiple instances of each ID in the dataset in memory are tied to unique IDs in the second dataset, and *many-to-many*, where potentially many instances of each ID in both datasets are tied together. I have often used one-to-one matching, sometimes used one-to-many and many-to-one matching, and don't think I've ever used many-to-many matching.

The syntax is as follows for merging on specified variables (**`m`** means many):

```
merge {1:1, 1:m, m:1 or m:m} {variable(s) common to both datasets
to match on} using "{new dataset}.dta"
```

or as follows for merging on observation number:

```
merge 1:1 _n using "{new dataset}.dta"
```

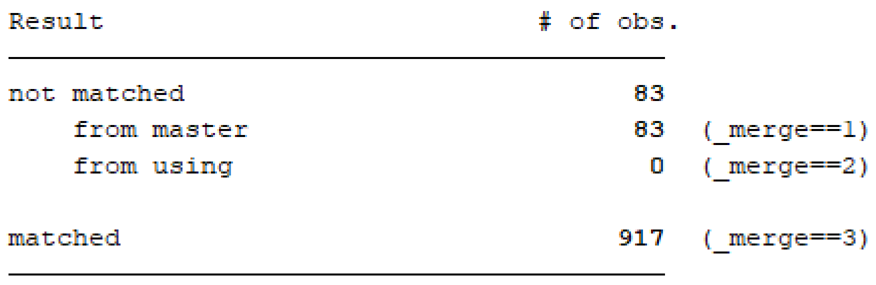
For example, we could add a new variable to the **Lesson_01.dta** file using the **merge_01.dta** file. The **merge_01.dta** file has the same ID variable, called *var1* (temporarily), and an additional variable called *date_of_recruitment*. We're merging using 1:1 matching as there is exactly 1 observation for each ID number in both datasets.

First, make sure your **Lesson_01_{initials}.dta** file is loaded into memory and the working directory is set to the folder containing your downloaded datasets (it should still be if you did this earlier):

```
use "Lesson_01_SH.dta", clear
```

Then to **`merge`**, type:

```
merge 1:1 var1 using "merge.dta"
```



```
Result                           # of obs.
-----------------------------------------
not matched                            83
    from master                        83  (_merge==1)
    from using                          0  (_merge==2)

matched                               917  (_merge==3)
-----------------------------------------
```

Stata displays an output for the results of the merge, showing the number of observations in the original dataset *not matched* in the new dataset (*not matched from master*, coded as a 1), the number of observations in the new dataset not matched with the original dataset (*not matched from using*, coded as a 2), and the number of observations that matched between the datasets (*matched*, coded as a 3). Stata also, by default, creates a variable called *_merge*, which lets you easily tell whether an observation was matched or not using the codes you can see above.

From this output, we can see that 917 observations matched between the **Lesson_01_SH.dta** and **merge.dta** datasets, but there were 83 observations in the **Lesson_01_SH.dta** dataset that weren't in the **merge.dta** dataset: these observations will therefore have missing values of *date_of_recruitment*.

The final command before moving onto the exercise is to save the dataset one more time, as we'll load this in at the start of the second lesson:

```
save "Lesson_01_SH.dta", replace
```

1.10 Exercise

For this exercise, see how you do with the following:

1. Open a fresh copy of Stata

2. Change the directory to the folder containing your downloaded copy of the **Exercise_01.dta** dataset
   a. This dataset is similar to the **Lesson_01.dta** dataset, but with some tweaks

3. Load in the **Exercise_01.dta** dataset

4. Save a copy of the dataset with your initials at the end

5. Have a look at the variables listed in the top right window, and the dataset information in the bottom right window

6. Have a look at the data itself using the spreadsheet view
   a. Take a note of any numeric, string and labelled numeric variables, and remind yourself of their colours and what they mean

7. Append the **Exercise_01_append.dta** dataset
   a. Check that the command added observations to the dataset

8. Merge the **Exercise_01_merge.dta** dataset
   a. You can load up another instance of Stata to take a quick look at the dataset (hint: you'll need to merge 1:1 on the *id* variable)
   b. Check which variable(s) were added to the dataset
   c. The variable will look strange, but that's ok, we'll go through how to **format** variables in a later lesson

9. Open up the help menu for **`merge`** and take a quick look at the available options
   a. See if you can figure out how to merge a dataset without creating the *_merge* variable

10. Save the appended and merged dataset as **Exercise_01_{initials}.dta**

11. Now export this dataset to an Excel spreadsheet
    a. Make sure the variable names are in the header row

1.11 Exercise – Answers

1. *Note: you can have multiple instances of Stata open at once – each new copy has its own settings, and each can have its own dataset loaded, although each copy of Stata can't interact with any others.*

2. To change the directory, use the **`cd`** command and specify the folder where you downloaded the datasets, making sure the folder name is in quotation marks:
    a. **`cd "G:\Documents\Online teaching\01 - Introduction to Stata\Stata"`**

3. To load a dataset, use the **`use`** command, followed by the name of the dataset you want to load. If you've changed the directory, you don't need the full file path:
    a. **`use "Exercise_01.dta", clear`**
    b. You don't need the **`clear`** option unless you are loading data over an unsaved dataset

4. To save a dataset, use the **`save`** command, followed by the name of the file you want to save the dataset as. If you've changed the directory, you don't need the full file path:
    a. **`save "Exercise_01_SH.dta", replace`**
    b. You don't need the **`replace`** option unless you are saving data over an existing dataset

5. You can see the variables and their labels in the top right window, and some summary dataset information in the bottom right window
    a. There should be 14 variables, with 1,000 observations, taking up something like 57 KB

6. Use either of the *Data Editor* buttons to load up the spreadsheet view, or use the **`browse`** or **`edit`** commands
    a. Most variables are black, so are numeric
    b. One variable is red, *var3*, which means it's a string variable
    c. Two variables are blue, *hair_colour* and *accommodation*, which means they are labelled numeric variables (numbers that have a label applied to them to be more informative)

7. To append a dataset, use the **`append`** command:
    a. **`append using "Exercise_01_append.dta"`**

8. To merge a dataset, use the **merge** command:
    a. `merge 1:1 id using "Exercise_01_merge.dta"`
    b. The merged variable, *date_of_recruitment*, is just a list of numbers in the 21,000 range – this is how Stata encodes dates (number of days since 01/01/1960), and we'll need to format it to show an interpretable date

9. To open up the help menu, use the help command:
    a. `help merge`
    b. One of the options, **nogenerate**, looks like it will not create the *_merge* variable
    c. You'll have to remove the *_merge* variable before trying to **merge** again – we'll cover this in the next lesson

10. Save again, making sure to include the replace option:
    a. `save "Exercise_01_SH.dta", replace`

11. You can export using the *Export* interface under *File*
    a. The code that Stata generates will hopefully look something like this:
    b. `export excel using "Exercise_01_SH", firstrow(variables) replace`

Lesson 2: Basic Commands

This lesson introduces some basic commands, including listing variables, displaying summary information about variables, generating, replacing, and renaming variables, labelling variables, applying labels to numeric data, and removing data.

First, let's load in the **Lesson_01_{initials}.dta** file (the one with your initials):

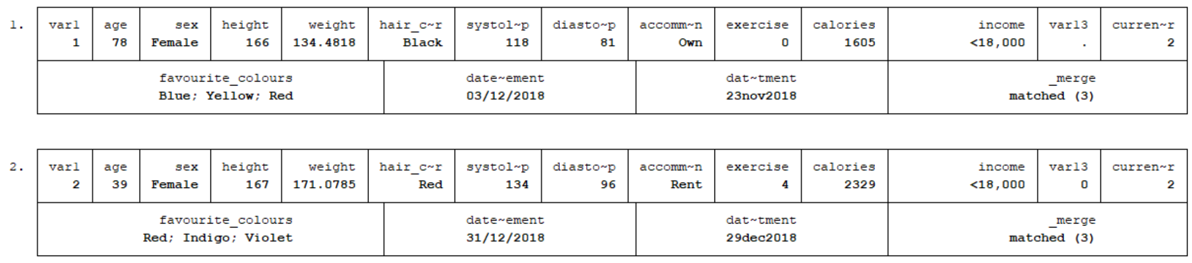
```
use "Lesson_01_SH.dta", clear
```

Alternatively, load in the **Lesson_02.dta** if you want a fresh dataset.

2.1 List

One of the simplest commands simply presents observations in a dataset. To do this, simply type:

```
list
```

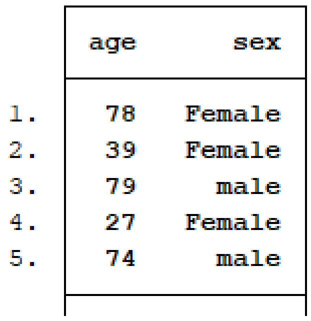


```
     +-------------------------------------------------------------------------------------------------------------------------------------------+
  1. | var1 | age |    sex | height |   weight | hair_c~r | systol~p | diasto~p | accomm~n | exercise | calories |         income | var13 | curren~r |
     |    1 |  78 | Female |    166 | 134.4818 |    Black |      118 |       81 |      Own |        0 |     1605 |        <18,000 |     . |        2 |
     |-------------------------------------------------------------------------------------------------------------------------------------------|
     |         favourite_colours          |        date~ement        |        dat~tment        |                  _merge                        |
     |         Blue; Yellow; Red          |        03/12/2018        |        23nov2018        |               matched (3)                      |
     +-------------------------------------------------------------------------------------------------------------------------------------------+

     +-------------------------------------------------------------------------------------------------------------------------------------------+
  2. | var1 | age |    sex | height |   weight | hair_c~r | systol~p | diasto~p | accomm~n | exercise | calories |         income | var13 | curren~r |
     |    2 |  39 | Female |    167 | 171.0785 |      Red |      134 |       96 |     Rent |        4 |     2329 |        <18,000 |     0 |        2 |
     |-------------------------------------------------------------------------------------------------------------------------------------------|
     |         favourite_colours          |        date~ement        |        dat~tment        |                  _merge                        |
     |        Red; Indigo; Violet         |        31/12/2018        |        29dec2018        |               matched (3)                      |
     +-------------------------------------------------------------------------------------------------------------------------------------------+
```

If the output goes on for a while, I'll only show a bit of it after each command to save space. Just typing **`list`** by itself will list all observations in all variables; if there isn't enough space on your screen to show all the variables on a single line, Stata will split each observation over multiple lines. It can help to restrict which variables are shown by typing them after **`list`**, for example if you only wanted to **`list`** *age* and *sex*:

```
list age sex
```



```
     +--------------+
     | age      sex |
     |--------------|
  1. |  78   Female |
  2. |  39   Female |
  3. |  79     male |
  4. |  27   Female |
  5. |  74     male |
     |--------------|
```

There' isn't much more to say about **`list`**, but depending on how your version of Stata is set up, Stata may wait for you to click –more– or press the spacebar once the results window is full. Alternatively, you can stop Stata from listing data by clicking the red cross ⊗ at the top of the screen. If you want to have Stata print out all the results without having to click –more–

, then type into the command bar:

```
set more off
```

This will force Stata to give you all the results in one go for the duration of the Stata session. If you want to permanently force Stata to do this for all future sessions, type:

```
set more off, permanently
```

Alternatively, if you want Stata to always wait for you to click through the results, type:

```
set more on [, permanently]
```

2.2 Summarize

summarize (note the American spelling, short: **sum**) gives a summary of one or more variables:

```
summarize
```

```
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
        var1 |      1,000       500.5    288.8194          1       1000
         age |      1,000     126.546     257.962         20        999
         sex |          0
      height |        957    168.6917    8.556731        149        192
      weight |        944    169.7401     26.3858   95.90097   246.9174
-------------+---------------------------------------------------------
```

This will display a summary of all variables in your dataset, telling you how many non-missing (numeric) observations there are for each variable, its mean value, standard deviation, and minimum and maximum values. String variables, like *sex*, will have 0 observations and no other information (you can't find the mean of a string, for example).

Adding the option **detail** gives you more detailed information, including the median and other percentiles, while adding variable names after **summarize** will restrict the summaries to those variables in the same way it did for **list**. For instance, if we wanted to know the median of age (the 50th percentile), we would write:

```
sum age, detail
```

```
                         Age (years)
-------------------------------------------------------------
      Percentiles      Smallest
 1%           20             20
 5%           23             20
10%           27             20       Obs               1,000
25%         36.5             20       Sum of Wgt.       1,000

50%           53                      Mean            126.546
                        Largest       Std. Dev.       257.962
75%           70            999
90%           79            999       Variance       66544.41
95%          999            999       Skewness       3.072292
99%          999            999       Kurtosis       10.49753
```

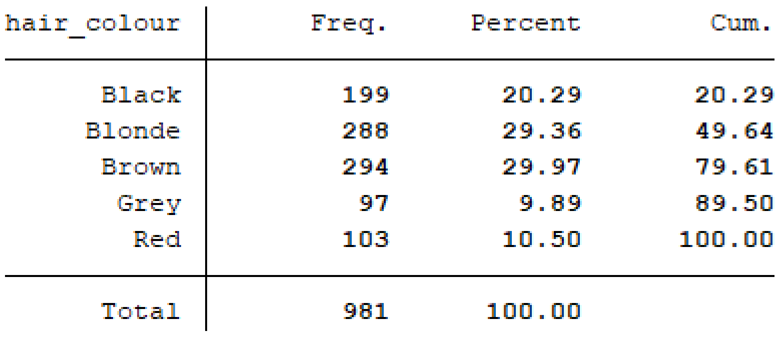
We can see that some ages are 999 (the largest four values at least, and the top 5%), which seems implausible when age is measured in years. We'll change this in just a bit to a missing value, as 999 is a common code for missing values.

Like most commands, we can shorten the **summarize** command to **sum**, and we often use it to have a quick look at the data before manipulation.

2.3 Tabulate

The command **tabulate** (short: **tab**), can produce one- and two-way tables of categorical variables (variables categorized into groups, like hair colour or sex), depending on whether one or two variables are specified when you run the command. If we wanted to see the number of observations for each hair colour in a one-way table, we would type:

```
tabulate hair_colour
```

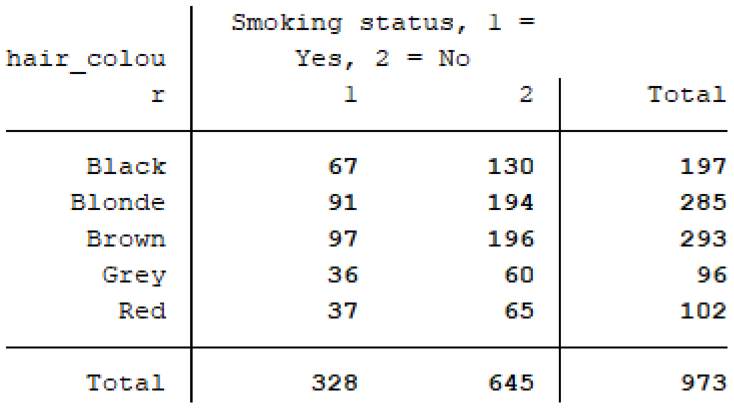


```
hair_colour |      Freq.     Percent        Cum.
------------+-----------------------------------
      Black |        199       20.29       20.29
     Blonde |        288       29.36       49.64
      Brown |        294       29.97       79.61
       Grey |         97        9.89       89.50
        Red |        103       10.50      100.00
------------+-----------------------------------
      Total |        981      100.00
```

In addition to the number of observations, we see percentages and the cumulative percentage. By default, missing values are left out of tabulations, so you can see only 981 out of 1,000 people have values of hair colour.

If we wanted to see the number of observations for each hair colour, but this time split by smoking status in a two-way table, we would type:

```
tab hair_colour current_smoker
```



```
           | Smoking status, 1 =
hair_colou |    Yes, 2 = No
         r |         1          2 |     Total
-----------+----------------------+----------
     Black |        67        130 |       197 
    Blonde |        91        194 |       285 
     Brown |        97        196 |       293 
      Grey |        36         60 |        96 
       Red |        37         65 |       102 
-----------+----------------------+----------
     Total |       328        645 |       973 
```

Here, we see only the number of observations within each hair colour for each smoking status. For example, there are 67 people who have black hair and smoke (smoke == 1), and 130 people who have black hair who do not smoke (smoke == 2), adding up to 197 people with black hair. The total number of observations has dropped to 973, as anyone with a missing hair colour *or* smoking status will be omitted from the table.

If we want to see the missing values, we can add the option **missing** to the command:

```
tab hair_colour current_smoker, missing
```

```
 hair_colou |    Smoking status, 1 = Yes, 2 = No
          r |         1          2          . |     Total
------------+---------------------------------+----------
            |         6         13          0 |        19 
      Black |        67        130          2 |       199 
     Blonde |        91        194          3 |       288 
      Brown |        97        196          1 |       294 
       Grey |        36         60          1 |        97 
        Red |        37         65          1 |       103 
------------+---------------------------------+----------
      Total |       334        658          8 |     1,000 
```

The missing hair colours are blanks (top row), and the missing smoking statuses are represented by a full stop (the missing value character for numeric variables in Stata, penultimate column).

Two-way tables don't show percentages by default: it wouldn't necessarily be clear whether the percentages were going across the rows, e.g. percentage of people with black hair who smoke, or down the columns, e.g. percentage of people who smoke who have black hair. However, we can add the percentages with the **`row`** and **`column`** (short: **`col`**) options, saying that we want to see the row and column percentages respectively:

```
tab hair_colour current_smoker, row
```

```
           |  Smoking status, 1 =
 hair_colou |     Yes, 2 = No
          r |         1          2 |     Total
-----------+----------------------+----------
     Black |        67        130 |       197 
           |     34.01      65.99 |    100.00 
-----------+----------------------+----------
    Blonde |        91        194 |       285 
           |     31.93      68.07 |    100.00 
-----------+----------------------+----------
     Brown |        97        196 |       293 
           |     33.11      66.89 |    100.00 
-----------+----------------------+----------
      Grey |        36         60 |        96 
           |     37.50      62.50 |    100.00 
-----------+----------------------+----------
       Red |        37         65 |       102 
           |     36.27      63.73 |    100.00 
-----------+----------------------+----------
     Total |       328        645 |       973 
           |     33.71      66.29 |    100.00 
```

```
tab hair_colour current_smoker, column
```

```
           |  Smoking status, 1 =
hair_colou |      Yes, 2 = No
         r |         1          2 |     Total
-----------+----------------------+----------
     Black |        67        130 |       197
           |     20.43      20.16 |     20.25
-----------+----------------------+----------
    Blonde |        91        194 |       285
           |     27.74      30.08 |     29.29
-----------+----------------------+----------
     Brown |        97        196 |       293
           |     29.57      30.39 |     30.11
-----------+----------------------+----------
      Grey |        36         60 |        96
           |     10.98       9.30 |      9.87
-----------+----------------------+----------
       Red |        37         65 |       102
           |     11.28      10.08 |     10.48
-----------+----------------------+----------
     Total |       328        645 |       973
           |    100.00     100.00 |    100.00
```

You can specify both row and column if you like, but it does get a little confusing.

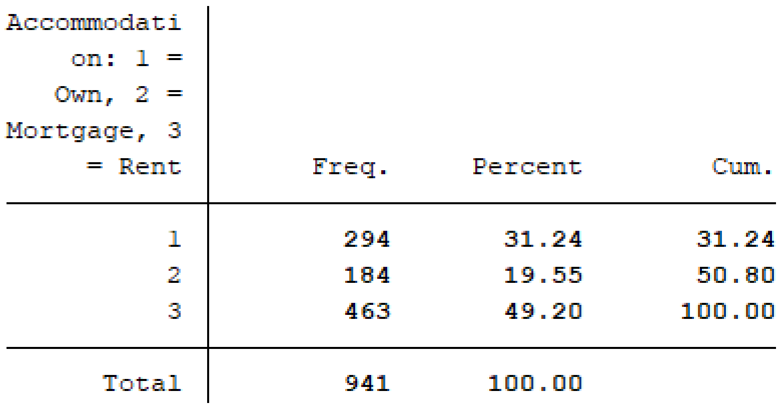
When you tabulate labelled numeric variables, like *accommodation*, then you see their labels rather than their underlying numbers:

`tab accommodation`

```
Accommodati |
  on: 1 =   |
  Own, 2 =  |
Mortgage, 3 |
     = Rent |      Freq.     Percent        Cum.
------------+-----------------------------------
        Own |        294       31.24       31.24
   Mortgage |        184       19.55       50.80
       Rent |        463       49.20      100.00
------------+-----------------------------------
      Total |        941      100.00
```

However, you can force Stata to show you the underlying numbers by adding the **`nolabel`** (short: **`nol`**) option:

`tab accommodation, nolabel`



```
Accommodati |
    on: 1 = |
   Own, 2 = |
Mortgage, 3 |
     = Rent |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |        294       31.24       31.24
          2 |        184       19.55       50.80
          3 |        463       49.20      100.00
------------+-----------------------------------
      Total |        941      100.00
```

This can be useful is you need to quickly check which label is assigned to which number, but it can also be useful to see them both at the same time: as this involves modifying **value labels**, which we do in **Section 2.8 Labelling Values**.

2.4 Generating Variables

You can create new variables in Stata using the **generate** command (short: **gen**), which has the syntax:

```
generate {new variable name} = {some expression}
```

Where **{some expression}** is an expression that Stata understands, which can be as simple as adding two variables together or multiplying a variable by 2, or could be as complex as you can imagine: my largest **generate** command stretched to 10 lines when I copied it into a word document. Remember that the **{new variable name}** must start with a letter, have no spaces and be unique, i.e. not named after any existing variables.

Let's suppose that we want to create a new variable that is the mean of systolic and diastolic blood pressure (I'm pretty sure this is clinically meaningless, but let's suppose anyway):

```
generate average_bp = (systolic_bp+diastolic_bp)/2
(73 missing values generated)
```

The only output from this command is to tell us if any observations in the new variable are missing. In this example, the new variable *average_bp* will be missing for any observations that are missing for either systolic or diastolic blood pressure, since missing values aren't used in most calculations in Stata. This is the usual cause of missing values, though Stata will also give missing values for calculations that don't make sense, like dividing a number by 0. Brackets follow BODMAS rules, i.e. brackets first, then orders (squares, cubes etc.), division, multiplication, addition and finally subtraction. As ever, it's usually a good idea to take a look at new variables to make sure they're doing what you want.

When writing out expressions, spaces don't matter all that much: they aren't necessary, but they also won't mess up a command if they're there. If things look clearer to you with spaces, put them in. If they don't, then don't.

Let's add another variable for practice. This time, let's generate another ID number equal to the observation number (there's already an ID variable that does this, *var1*, but let's pretend we didn't see that):

```
gen id2 = _n
```

The **_n** expression tells Stata to use the row number in each observation, so here the numbers 1 to 1,000 inclusive, and is a useful thing to know. There's no output for this command because there are no missing values in the expression.

2.5 Replacing Variables

Once a variable is created, we can't use **`generate`** to overwrite the data, as Stata requires a new variable name for each **`generate`** command. This is part of Stata's attempt to protect you from overwriting data: if you want to open a new dataset on top of a dataset already in memory, you either need to save your existing dataset or provide the option **`clear`**. Equally, if you want to save a dataset over an existing dataset, you need to provide the option **`replace`**.

If you want to overwrite a variable in a dataset, instead of using **`generate`**, you need to use the command **`replace`** (no short form, which may also be part of Stata's efforts to stop you accidentally deleting or overwriting data). The syntax, however, is the same as for **`generate`**:

```
replace {existing variable name} = {some expression}
```

Let's say we've noticed that height is in centimetres and we would really like it in metres:

```
replace height = height/100
(957 real changes made)
```

Stata will tell you how many changes it has made when replacing a variable, in this case 957 changes (the 43 missing observations for height aren't changed). We've also noticed that weight is in pounds (lbs), and would like it to be in kilograms (kg), and there are 2.20462 pounds per kilogram:

```
replace weight = weight/2.20462
(944 real changes made)
```

Now we have height in metres and weight in kilograms, we could also generate a new variable for body mass index (BMI), which is weight in kg divided by height in metres squared (the ^ symbol is used for powers/orders, so *x*^2 means *x* squared, *x*^3 means *x* cubed etc., where *x* can be a variable or a number):

```
gen bmi = weight/height^2
(95 missing values generated)
```

Let's do a quick check of BMI to see whether the command has generated something that looks reasonably correct:

```
sum bmi

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         bmi |        905    26.98933    3.204688   17.42509   37.25286
```

The summary of BMI tells us the average BMI is about 27 kg/m^2, which seems about right, and the minimum and maximum values are also about right for BMI. There are 95 missing values, but that's ok, there were quite a few missing values for height and weight too.

2.6 Renaming Variables

Renaming variables requires another new command, this time **`rename`**, which has the syntax:

```
rename {old variable name} {new variable name}
```

The limitations on naming variables still apply: it can’t be the same as an existing variable, can’t have special characters (apart from an underscore), and you have to start a variable name with a letter (or underscore).

In our dataset, the original ID number is named *var1*, so we will rename it *id*, and whether someone has run a marathon in the past year is named *var13*, so we’ll rename that *marathon*:

```
rename var1 id
rename var13 marathon
```

2.7 Labelling Variables

Labelling variables is usually a good idea as it gives more context and information to variable names. In the top right variables window, you can see the variable labels next to the variable names. Currently, *hair_colour* and each of the variables we've just created are unlabelled, so if we come back to the dataset in a year, we may forget what each variable actually is.

Labelling variables is straightforward, but note the quotation marks around the label:

```
label variable {variable name} "{Label of your choice}"
```

Let's add variable labels to every variable that doesn't have a label:

```
label variable hair_colour "Hair colour"
label variable average_bp "Average of systolic and diastolic
blood pressure (mmHg)"
label variable id2 "ID number (2)"
label variable bmi "Body Mass Index (kg/m2)"
```

Arguably, *hair_colour* and *id2* don't need labels, but who knows what we'll think these variables mean in a year. You can make labels up to 80 characters long, and because they're strings, there isn't a limit on the characters you can use. If you don't specify a label it will remove the variable label. For example, typing **`label variable bmi`** would remove the label for *bmi*.

Also, we've noticed that because we changed the units of height and weight, their labels are now wrong. Let's fix that:

```
label variable height "Height (m)"
label variable weight "Weight (kg)"
```

This will replace the height and weight labels: we're overwriting metadata (data about our data), not the data itself, and I guess Stata is less concerned about that than overwriting data.

2.8 Labelling Values

Labelled numeric variables receive their labels through the same **label** command, but it's used in a different way. Remember that the labelled numeric variables appear blue in the spreadsheet view, have labels that look like strings but act as numbers when using them in commands and expressions. These are particularly useful for categorical and binary variables, where the numbers mean something, for example 0 = "No" and 1 = "Yes".

This is a two-stage command, and so slightly trickier than previous commands. First, we need to define a label and give it both a name and a list, telling it what each number (which has to be an integer, i.e. whole number) in the label should mean. Labels are separate from variables, so you normally only see them through their effects on variables. The label names should be unique (they can be the same as variable names, just unique for label names) and start with a character, and you can have up to 1,000 (or 65,536 depending on your version of Stata) numbers with associated meanings per label. Once defined, we then apply the label to one or more variables.

The syntax is as follows:

```
label define {label name} {1st number} {"meaning of 1st number"}
{2nd number} {"meaning of 2nd number"} … {last number} {"meaning
of last number"}
```

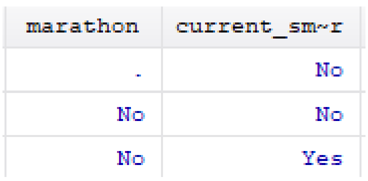
```
label values {list of variable names} {label name}
```

Let's do this for both *marathon*, where 0 = "No", 1 = "Yes"), and *current_smoker*, where 1 = "Yes" and 2 = "No":

```
label define marathon_label 0 "No" 1 "Yes"
label values marathon marathon_label

label define smoker_label 1 "Yes" 2 "No"
label values current_smoker smoker_label
```

There's no output from these commands, but you can see their effects when you switch to the spreadsheet view:



marathon	current_sm~r
.	No
No	No
No	Yes

The *marathon* and *current_smoker* variables are now blue, so are now labelled numeric variables rather than simply numeric variables. We can also see this if we **tabulate** the two variables:

```
tab marathon current_smoker
```

```
     Ran a
  marathon    Smoking status, 1 =
   in past       Yes, 2 = No
      year         Yes         No        Total
  ----------------------------------------------
        No         262        493          755
       Yes          64        142          206
  ----------------------------------------------
     Total         326        635          961
```

If you want to be able to see the underlying number *and* the label too, you can use the **numlabel** command, which adds or removes the underlying number as a prefix to the labels:

```
numlabel marathon_label smoker_label, add
tab marathon current_smoker
```

```
     Ran a
  marathon    Smoking status, 1 =
   in past       Yes, 2 = No
      year      1. Yes      2. No        Total
  ----------------------------------------------
     0. No         262        493          755
    1. Yes          64        142          206
  ----------------------------------------------
     Total         326        635          961
```

This can make it easier to see what the underlying numbers are for any numeric labelled variables. However, if you tire of this and want to remove the prefixes, you can, again using **`numlabel`**:

```
numlabel marathon_label smoker_label, remove
tab marathon current_smoker
```

```
     Ran a
  marathon    Smoking status, 1 =
   in past       Yes, 2 = No
      year         Yes         No        Total
  ----------------------------------------------
        No         262        493          755
       Yes          64        142          206
  ----------------------------------------------
     Total         326        635          961
```

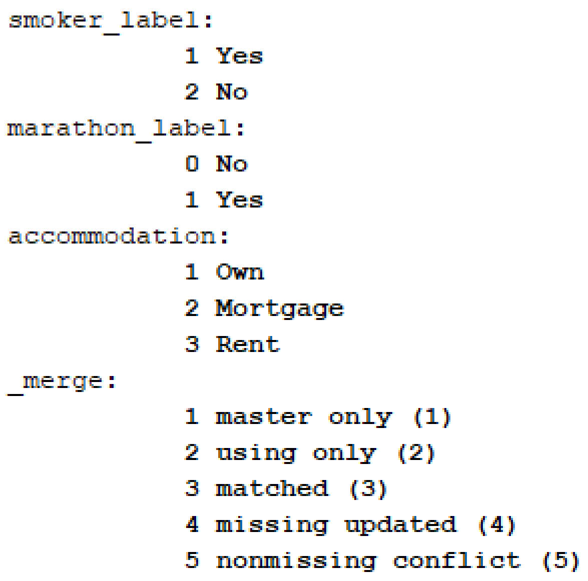
If you want to list all the labels in memory, type:

```
label dir
```

```
  smoker_label
  accommodation
  _merge
  marathon_label
```

This shows you all the labels you have in memory. The labels are saved with the dataset, so you don’t need to worry about losing them. If you want to list all the labels in memory along with their contents, type:

```
label list
```



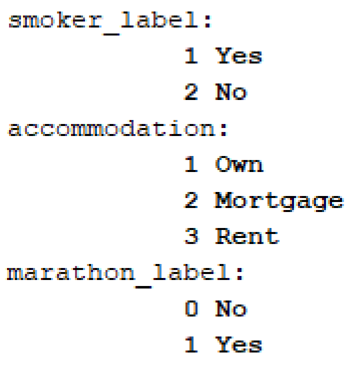
```
smoker_label:
           1 Yes
           2 No
marathon_label:
           0 No
           1 Yes
accommodation:
           1 Own
           2 Mortgage
           3 Rent
_merge:
           1 master only (1)
           2 using only (2)
           3 matched (3)
           4 missing updated (4)
           5 nonmissing conflict (5)
```

If you want to remove any labels, you can type:

```
label drop {label name}
```

For example, the *_merge* label, which was created when we merged datasets earlier, can be removed:

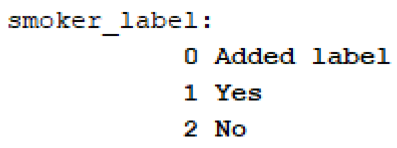
```
label drop _merge
label list
```



```
smoker_label:
           1 Yes
           2 No
accommodation:
           1 Own
           2 Mortgage
           3 Rent
marathon_label:
           0 No
           1 Yes
```

Finally, you can add to or modify existing labels using **`label define`** and the **`add`** and **`modify`** options. Adding a new number-meaning pair is easy enough with the **`add`** option:

```
label define smoker_label 0 "Added label", add
label list
```

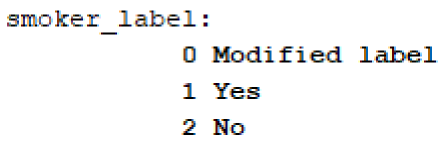


```
smoker_label:
           0 Added label
           1 Yes
           2 No
```

You can see this has added the **0 Added label** pair to the *smoker_label*.

You can't add a number-meaning pair that already exists – this is like generating a variable that already exists, Stata won't let you do it because you're overwriting data, instead of generating new data. Rather, if you want to change a number-meaning pair, you can with the **`modify`** option:

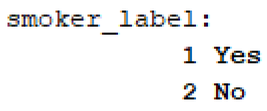
```
label define smoker_label 0 "Modified label", modify
label list
```



```
smoker_label:
           0 Modified label
           1 Yes
           2 No
```

Finally, if you want to remove a particular number-meaning pair, you can modify it but leave the meaning blank:

```
label define smoker_label 0 "", modify
label list
```



```
smoker_label:
           1 Yes
           2 No
```

2.9 Removing Data

Removing data in Stata is straightforward: use the **drop** command to remove any unwanted variable(s):

```
drop {variable(s) to be removed}
```

Dropping a variable is permanent, so it is important to be certain you are removing the correct variable. This is also why it makes sense to have a master copy of a dataset that you don't save over and use do files to make all the changes to a dataset, so you can always reopen the master dataset and redo all the commands if you make a mistake.

In our dataset, let's get rid of the *_merge* and *id2* variables, since we don't need them anymore:

```
drop _merge id2
```

Assuming the variables you've specified exist, Stata doesn't give you any notification that the variables have been removed, but if you check either the variables window in the top right, or the spreadsheet view, you'll find the variables are gone.

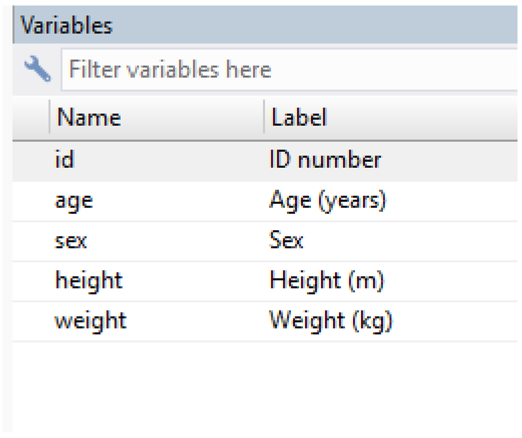
Now save your dataset with your initials and the lesson number, because we are about to start the exercises for this lesson, and we'll need this dataset in the next lesson:

```
save "BP_exercise_SH_02.dta", replace
```

Just before we go onto the exercises though, let's introduce the **keep** command: **keep** is the opposite of **drop**, in that **keep** keeps everything you specify and drops all other variables, and **drop** drops everything you specify and keeps all other variables. The syntax is the same.

For instance, if we decided we only needed the variables *id*, *age*, *sex*, *height* and *weight*, then we could use the command:

```
keep id-weight
```



Name	Label
id	ID number
age	Age (years)
sex	Sex
height	Height (m)
weight	Weight (kg)

Rather than writing out all the variables you want, you can use a quick trick: when listing variables, you can say you want two variables and all variables in between them by putting a dash between them. This trick is dependent on the variables being ordered correctly, so it's best to be sure you are specifying all the variables you want when doing this.

If you want to remove all variables and all observations, you can use another trick:

```
drop _all
```

In many commands, Stata understands **`_all`** to mean all variables. Dropping all observations and variables, however, doesn't drop labels, which persist. To get Stata back to a clean slate, as if you'd closed and reopened it, use:

```
clear
```

This will reset Stata entirely, ready for new data. This is the same as the option you specify when loading in a new dataset on top of existing, unsaved data. Stata is forcing you to explicitly say **`clear`** so it can be confident you're not overwriting data you actually need.

Finally, **`drop`** can be used to remove observations as well as variables, but this requires using *if* or *in* statements, which we cover in the next lesson.

2.10 Exercise

For this exercise, see how you do with the following:

1. Load in the **Exercise_01_{initials}.dta** dataset from the first exercise
    a. If you're not confident with your dataset, you can load in the **Exercise_02.dta** dataset instead

2. List the values of *age* and *height* together

3. Summarise the dataset, then summarise *height* in detail
    a. Check to see if anything looks wrong here

4. Tabulate *hair_colour* and find out how many people have red hair

5. Tabulate *hair_colour* and *accommodation* and find out what percentage of people with black hair rent

6. Tabulate *hair_colour* and *accommodation* and find out what percentage of people who own their home have grey hair

7. Add number prefixes to the *hair_colour* and *accommodation* labels, and tabulate *hair_colour* and *accommodation* again to check it worked
    a. You'll need to list the labels to find out to which labels you'll need to add the number prefixes

8. Modify the *hair_colour* label so "Blonde" is spelled correctly
    a. Check the label to make sure the label is fixed
    b. See if you can spot a new problem with it: if so, see if you can fix it

9. Convert *age* to years and *weight* to kilograms (1 stone = 6.35 kg)

10. Generate body mass index (BMI: weight in kilograms divided by height in metres squared), then give it an informative variable label
    a. Summarise the variable to check it looks about right
    b. If not, think about why they might be wrong

11. Create a label for *income* and apply it to the variable using these number-meaning pairs, then check it worked:
    a. 1 = <£18,000
    b. 2 = £18,000 to £30,000
    c. 3 = £30,000 to £50,000
    d. 4 = £50,000+

12. Rename *var3* and *var14* to something more informative using the variable labels as a guide

13. Remove the *_merge* variable (if present)

14. Save the modified dataset, calling it **Exercise_02_{initials}.dta**

2.11 Exercise – Answers

1. We've done this a few times now:
    a. **`use "Exercise_01_SH.dta", clear`**
    b. Or, if you're not confident in your save from exercise 1:
    c. **`use "Exercise_02.dta", clear`**

2. **`list`** command:
    a. **`list age height`**

3. **`summarize`** command:
    a. **`sum height, detail`**
    b. There are values of -1 for height, which definitely seems wrong – these are likely missing value codes again

4. A series of **`tabulate`** commands now:
    a. **`tab hair_colour`**
    b. 141 people have red hair

5. Add the row option to see the percentages across rows (if you specified *hair_colour* and *accommodation* the other way round, you'll need the column option)
    a. **`tab hair_colour accommodation, row`**
    b. 48.36% of people with black hair rent

6. Add the column option to see the percentages down columns (if you specified *hair_colou*r and *accommodation* the other way round, you'll need the row option)
    a. **`tab hair_colour accommodation, col`**
    b. 12.06% of people who own their home have grey hair

7. We'll need to check the label names, then use **`numlabel`** for this:
    a. **`label list`**
        i. The *hair_colour* and *accommodation* labels are called, simply, *hair_colour* and *accommodation*
    b. **`numlabel hair_colour accommodation, add`**
    c. **`tab hair_colour accommodation`**

8. This seems simple, but the number prefixes make it complicated:
    a. **`label define hair_colour 3 "Blonde", modify`**
    b. **`label list`**
        i. The "Blonde" label doesn't have a number prefix, while the others do.
    c. We can get around this in a couple of ways (either works fine):
    d. **`numlabel hair_colour, add force`**
        i. The **`force`** option gives number prefixes to those without them
    e. **`label define hair_colour 3 "3. Blonde", modify`**
        i. This also works, it just adds the prefix directly

9. **`replace`** commands:
    a. **`replace age = age/12`**
    b. **`replace weight = weight*6.35`**

10. You'll need a few commands here:
    a. **`gen bmi = weight/height^2`**
    b. **`label variable bmi "Body mass index (kg/m2)"`**
    c. **`sum bmi, detail`**
        i. Some values are very high (>100): this is likely due to the "-1" values of *height*, and we'll fix this in the next lesson

11. Create and apply a new label:
    a. **`label define income 1 "<£18,000" 2 "£18,000 to £30,000" 3 "£30,000 to £50,000" 4 "£50,000+"`**
    b. **`label values income income`**
    c. **`tab income`**

12. **`rename`** commands:
    a. **`rename var3 sex`**
    b. **`rename var14 smoking_status`**
        i. Or *current_smoker*, or anything really so long as it makes sense

13. **`drop`** command:
    a. **`drop _merge`**

14. And finally, a **`save`** command:
    a. **`save "Exercise_02_SH.dta", replace`**

Lesson 3: Restricting Commands

This lesson introduces restricting commands using *in*, *if* and *by*, as well as how to **`sort`**, **`order`** and **`preserve`** datasets.

First off, let's set the working directory and load in the dataset we saved at the end of the last lesson, or **Lesson_03.dta** if you want a fresh dataset.

So far, we've only looked at applying commands to the whole dataset. However, Stata has a few ways of restricting commands so they only apply to a subset of the data. This can be accomplished using either or both the *in* and *if* qualifiers. Both qualifiers go at the end of the command but before any options (so just before the comma), and both *in* and *if* can be used at once if you feel like it. We'll also go through sorting, ordering and preserving datasets, as these are useful things to know.

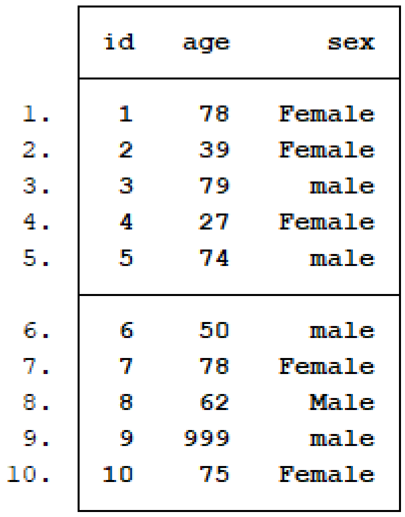
3.1 The *in* Qualifier

The *in* qualifier restricts the command to observations in a specified range of observations. For example, you could specify the first 10 observations, the last 100 observations, or any particular range of your choice. This is dependent on how the data is sorted, which we'll introduce here too. While *in* isn't a command, so doesn't have syntax in the same way, it looks like this:

```
{command} {variable(s)} in {range}
```

In Stata, you can specify a range of any kind as {number 1}/{number 2}, meaning all integers between the first and second number inclusive: think of the forward slash as Stata code for "to", so "number 1 to number 2". For example, you can list the first 10 observations for *id*, *age* and *sex* by typing:

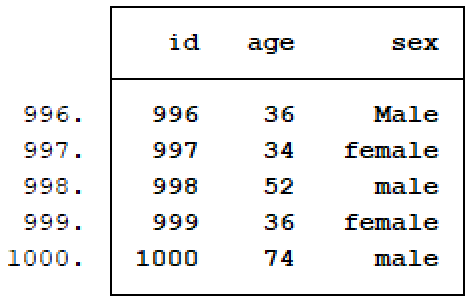
```
list id age sex in 1/10
```



```
     +----------------------+
     | id   age        sex  |
     |----------------------|
  1. |  1    78     Female  |
  2. |  2    39     Female  |
  3. |  3    79       male  |
  4. |  4    27     Female  |
  5. |  5    74       male  |
     |----------------------|
  6. |  6    50       male  |
  7. |  7    78     Female  |
  8. |  8    62       Male  |
  9. |  9   999       male  |
 10. | 10    75     Female  |
     +----------------------+
```

If you want to list the observations a set amount from the end of the dataset then you use minus numbers, so for example the last 5 observations is the range -5/-1. We can list these observations for *id*, *age* and *sex* by typing:

```
list id age sex in -5/-1
```



```
        +-------------------------+
        |   id   age        sex   |
        |-------------------------|
  996.  |  996    36       Male   |
  997.  |  997    34     female   |
  998.  |  998    52       male   |
  999.  |  999    36     female   |
 1000.  | 1000    74       male   |
        +-------------------------+
```

The negative sign effectively makes this statement "list all observations from the 5th observation from the end to the final observation".

The *in* qualifier also has a couple of special characters: you can use **`f`** and **`l`** to mean, respectively, the first and last observations. Personally, I think it's simpler and easier to read if you specify 1 and -1 to mean the first and last observations, but either will work.

There are two main limitations of the *in* qualifier. The first is that you can only specify a single range: if you want to specify multiple ranges, you're better off either using *if* (which is a lot more powerful in any case, see below), or using multiple commands specifying each range you want.

The second limitation is that *in* is entirely dependent on how the data is sorted. The first ten observations could be very different if the dataset is sorted by *age* rather than *accommodation,* for instance.

3.2 Sorting a Dataset

Sorting a dataset is, fortunately, simple to do. Currently, the dataset is currently sorted by *id*, but let's sort by *age* instead:

```
sort age
```

There's no output from Stata unless you misspell a variable, but you can see the effects using the spreadsheet view or **`list`**. You can specify any number of variables when sorting. If we wanted to sort by *hair_colour*, then *accommodation*, then *age*, you can by specifying the variables in that order:

```
sort hair_colour accommodation age
```

The sort command automatically sorts upwards, so larger negative numbers to zero to larger positive numbers, and alphabetically, where capital letters are sorted before lower case letters, i.e. A to Z then a to z. Missing string values (blanks), and are sorted at the top, while missing numeric values (full stops), so are sorted at the bottom.

If you want to sort in the reverse direction, so larger positive numbers to zero to larger negative numbers, and reverse alphabetically, i.e. z to a then Z to A, you have to use a slightly different command, **`gsort`**. **`gsort`** does the same job as **`sort`**, but allows you to put a negative sign in front of any variable you want to reverse sort. For example, if we wanted to **`sort`** by *age*, but oldest to youngest, then we could type:

```
gsort -age
```

If we wanted to sort by reverse *hair_colour*, then *accommodation*, then reverse *age*, we could type:

```
gsort -hair_colour accommodation -age
```

Let's sort the data by *id* again to get back to where we started:

```
sort id
```

3.3 Ordering Variables

While we're sorting observations, it is worth mentioning we can also **`order`** variables. This uses the **`order`** command, and is usually a bit more direct than **`sort`**, in that you often just specify the exact order of the variables that you want. There is little reason to **`order`** variables apart from keeping the dataset tidy, unless you regularly specify a group of variables by typing in the first and last variable of the group with a hyphen between them, e.g. **`list age-weight`**. However, there is much to be said for keeping a dataset tidy.

There are a few ways to use **`order`**. The first is to literally type out all variables in the order in which you want them to appear. This is time consuming, but gives the most control, and there may be no quicker way or ordering variables to be how you want them. However, you don't have to specify all variables at once: those that you specify will appear at the top of the variable list (and left of the spreadsheet view) in the order you give, and the rest of the variables will be stuck on the end in the same order they started in.

For example, let's say we want *bmi* to appear after *weight*. We could type:

```
order id age sex height weight bmi
```

There is no Stata output unless you misspell a variable. This does what we want though: it moves *bmi* and no other variable. However, it is more cumbersome than it needs to be. If we only want to move one variable at a time, we can move it to be before or after another variable using different options. For instance, if we want *average_bp* to come before *systolic_bp*, we can type:

```
order average_bp, before(systolic_bp)
```

Alternatively, if we want *average_bp* to come after *diatolic_bp*, we could type:

```
order average_bp, after(diastolic_bp)
```

Both **`after`** and **`before`** can be shortened, to **`a`** and **`b`** respectively.

Finally, we could order either all variables or a subset alphabetically. However, if you use this option, be prepared: there is no undo button, and the only way to get the variables back to how they started is to manually order them or reload the dataset. We'll cover ordering alphabetically in the next section.

3.4 Preserving Data

As frequently stated (possibly tediously so), there is no undo button in Stata. However, there is a command that allows you temporarily **`preserve`** the dataset within Stata memory, and **`restore`** it whenever you like. This is particularly useful when you are experimenting with new commands and don't want the hassle of redoing all the commands you've entered since the most recent save.

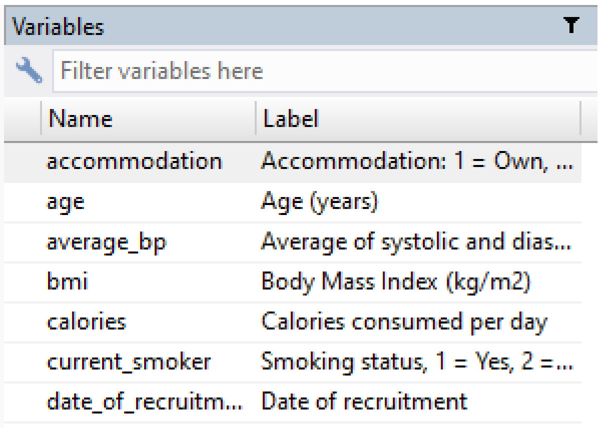
Let's give it a go: to **`preserve`** a dataset, simply type (in the command bar of Stata, not a do file, for reasons explained below):

```
preserve
```

This creates a temporary save within Stata memory that you can **`restore`** whenever you like. However, an important caveat: if you are using a do file to run these commands, **`preserve`** only works *while the do file is running code*. This means that as soon as the do file has finished executing the highlighted batch of code (or the whole do file), the preserved data will vanish entirely, and you can't restore it. We'll meet this quirk of Stata again when we look at **macros**, but for now, let's just work in the command bar of the main Stata window. It's also worth noting that you can't preserve a dataset more than once: if there's a preserved dataset in memory, Stata will give you an error saying there is data "already preserved".

We will now order all variable alphabetically, since we know if we don't like the result, we can reload the preserved data:

```
order _all, alpha
```



Name	Label
accommodation	Accommodation: 1 = Own, ...
age	Age (years)
average_bp	Average of systolic and dias...
bmi	Body Mass Index (kg/m2)
calories	Calories consumed per day
current_smoker	Smoking status, 1 = Yes, 2 =...
date_of_recruitm...	Date of recruitment

This orders all the variables alphabetically: note the use of *_all* to tell Stata to apply the command to all variables. However, the ordering of variables now makes no sense, so let's **`restore`** the data we preserved:

```
restore
```

On being restored, the preserved dataset vanishes and you can now use **`preserve`** again.

If you want to remove a preserved dataset from memory without loading it, you can type:

```
restore, not
```

If you want to **`restore`** a preserved dataset and keep the preserved dataset in memory as well, saving you from issuing another **`preserve`** command immediately after restoring, you can type:

```
restore, preserve
```

This doesn't really save any time as you still have to type out **`preserve`**, but it might be worth knowing.

A final note on preserving data: if you use **`preserve`** in a do file, and before restoring you encounter an error, Stata will **`restore`** the preserved dataset anyway after encountering the error. This can lead to odd situations when trying to work out what went wrong, so it's worth bearing in mind if you **`preserve`** data.

3.5 The *if* Qualifier

While the *in* qualifier restricts observations to a set range based on how the dataset is sorted, the *if* qualifier restricts observations to those where an expression is true. The expression, which I'll call an *if statement*, can be as short and simple or long and complicated as you like. For example, you can specify that a is be applied to observations where hair colour is "Black", where hair colour is not "Black", where age is above 60 years, or where height is 1.7 metres or over *AND* weight is below 100 kilograms *OR* weight is below 80 kilograms.

The general syntax for the *if* qualifier is:

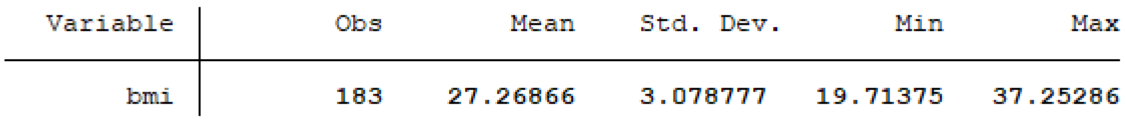
```
{command} {variable(s)} if {some expression}
```

There's also some particular characters you need to know about to represent different operators in Stata, such as *AND*, *OR* and *NOT*. We'll go through each of these as we use them, but for reference, here they all are:

Arithmetic	Logical	Relational
+ addition	~ not	> greater than
- subtraction	! not	< less than
* multiplication	| or	>= greater or equal to
/ division	& and	<= less or equal to
^ power/order	== equal to	
	~= not equal to	
	!= not equal to	

Right, let's go through some examples. Let's say we wanted to **`summarize`** *bmi* for all people with black hair:

```
sum bmi if hair_colour == "Black"
```



```
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         bmi |        183    27.26866    3.078777   19.71375   37.25286
```

You can see that only a relatively small number of observations were summarised: these are the observations with the value "Black" for the variable *hair_colour*. There's two points to be aware of here: the first is that when using *if* statements, you use two equals signs together to denote **equivalence**, i.e. observations where the variable *hair_colour* is equivalent to the value "Black". This is distinct from using a single equals sign when **assigning** a value to a variable, e.g. when we created the *bmi* variable. This is a really common error to make when using *if* statements – if you run into errors, then check the number of equals signs you have after the word *if*.

The second point to be aware of is, as ever, strings are put in quotation marks. This is to distinguish a string from a variable: without the quotation marks, then to Stata the *if* statement above would mean "**summarize** *bmi* for any observations where *hair_colour* is equivalent to [the variable] *black*", rather than "**summarize** *bmi* for any observations where *hair_colour* is equivalent to [the string] 'black'".

If we wanted to `summarize` *bmi* for all people who do not have black hair, we would type:

```
sum bmi if hair_colour != "Black"
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         bmi |        722    26.91853    3.234061   17.42509   36.28467
```

The exclamation mark and single equals sign means "not equivalent to", so Stata understands this as "**summarize** *bmi* for any observations where *hair_colour* is NOT equivalent to [the string] 'black'". Another important point here: missing values are interpreted as not equivalent to anything that isn't missing, so here, any person with missing hair colour will be counted in those 722 observations. As such, when creating *if* statements, we always want to think about whether we want missing values to be included or not. In this case, if we don't want missing values to be included, then we need to add *and* statement to the *if* statement:

```
sum bmi if hair_colour != "Black" & hair_colour != ""
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         bmi |        706    26.91199    3.236354   17.42509   36.28467
```

You can see this has reduced the number of observations, presumably by the number of people who have missing values for *hair_colour*. Three more important points here: the first is that missing values in strings are represented by blanks, which you specify by having two quotation marks next to each other.

The second point is that the ampersand (&) is used instead of the word *AND* would in regular speech (and some other programming languages), but it works in the same way: Stata understands this command as: "**summarize** *bmi* for any observations where *hair_colour* is NOT equivalent to [the string] 'black' *AND* where *hair_colour* is NOT missing".

The final point is that even though we're only talking about *hair_colour*, we always have to specify the variables in all parts of the *if* statement. For example, because not all parts of the *if* statement have a variable, Stata doesn't understand:

```
sum bmi if hair_colour != "Black" & != ""
="" invalid name
r(198);
```

3.6 Missing Values

A quick detour into missing values is called for now. Missing values in numeric variables are represented by a full stop rather than a blank string, so to say "not missing" for numeric variables, e.g. weight, you type:

```
sum bmi if weight != .
```

```
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         bmi |        905    26.98933    3.204688   17.42509   37.25286
```

We can now fix the values of *age* that are missing, but currently coded as 999. Let's build up the code a little. First, we know that we want to **`replace`** values of *age* as the variable already exists, so we know we're going to use the **`replace`** command. Next, we know we want some values to become missing, we know we'll want to have "= ." in the command too. Finally, we know we only want to set the values of *age* to missing for values of *age* that are currently 999, so we'll need an *if* statement to say exactly that, remembering that we'll need a double equals sign. Altogether, this means we'll want to type:

```
replace age = . if age == 999
(80 real changes made, 80 to missing)
```

Just to recap, we've replaced the values of age that were 999 with a full stop, which means "missing" to Stata, and it's told us that 80 values of age have been changed to missing. This is also a good representation of the difference between using a single equals sign to **assign** a value (**`age = .`**), and a double equals sign to denote **equivalence** (**`if age == 999`**).

Let's continue with a couple more examples. Let's say we want to **`count`** the number of people who are over 60 years of age. We could **`summarize`** *age* with an *if* statement, and that would tell us the number of people aged over 60 years, but we could also use the **`count`** command, which will do the same thing without the extra information:

```
count if age > 60
  387
```

Ok, so there are 387 people aged over 60. However, this is another case where we need to worry about missing values: missing numeric values are coded as a number close to infinity, which is higher than any number that you care to specify (definitely higher than 60). As such, when using *if* statements with greater than symbols, be wary of including missing values. For this example, we don't want to include missing values, so we'd type:

```
count if age > 60 & age != .
  307
```

That's better, the count now gives 307 people. We knew from turning the 999 values of age into missing values that there were 80 missing values of age, and as 387 minus 80 is 307, we can be pretty confident our *if* statement now excludes missing values.

One final important point, this time about missing values. For numeric values, there are actually multiple missing values, going from a single full stop, to **.a** to **.b** and all the way to **.z**, giving a total of 27 missing values. Each missing value is slightly higher than the one before it, so a single full stop is the smallest and **.z** is the biggest. The reasons there are so many different types of missing value is so that you can record different reasons for why values might be missing. For example, if someone says “I don’t know” when asked how old they are, that’s a different type of missing value to if someone wasn’t asked the question because the interviewer ran out of time, which is different again to a missing value because the person inputting data couldn’t read the person’s age on a form, which is different from age being missing because a page of a form was lost. By having multiple types of missing value for numeric variables, Stata allows you to record multiple reasons for having a missing value while still being counted as missing to Stata. This isn’t necessary for string variables because you could always write in the reason for why the value is missing.

This is a good thing, but it means that we need to be conscious of these different types of missing values when using *if* statements. If you *know* that you only have the standard missing values (full stops) in your dataset, then you can get away with using “not equivalent to missing”, as above (**`age != .`**). However, best practice is to always say “less than missing”, which means *all* types of missing are covered, since all types of missing values are larger than a single full stop. As such, whenever we encounter missing values again in this course, we’ll write “less than missing”. For example:

```
count if age > 60 & age < .
 307
```

3.7 The *if* Qualifier (Continued)

The final example in this section is summarizing *age* if either *height* is 1.7 metres or over *AND weight* is below 100 kilograms *OR weight* is below 80 kilograms. The *AND* and *OR* parts of *if* statements can be combined as many times as necessary to create the expression that you're looking for. Depending on your expression, brackets may be mandatory: this is much more likely if you are using a combination of *AND* and *OR* statements or multiple or nested *OR* statements, and it pays to be specific about what you want. In general, it's usually a good idea to put brackets around the different expressions in *OR* statements as this helps break the *if* statement into more digestible chunks.

Let's turn the example above into Stata code. First, we know we need to **`sum age`**, then have an *if* statement. The *if* statement should be true if either (or both):

- *height* is 1.7 metres or over *AND weight* is below 100 kilograms
- *weight* is below 80 kilograms

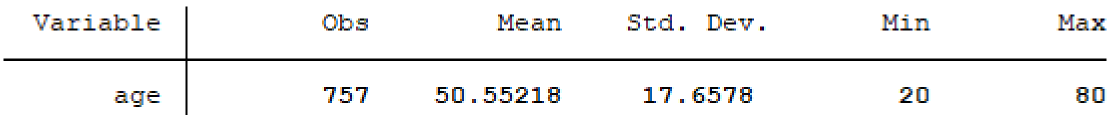
We can make this Stata code (remembering to account for missing values of height) by saying:

- **`height >= 1.7 & height < . & weight < 100`**
- **`weight < 80`**

"More than or equal to" in Stata is expressed as **`>=`**, and "less than or equal to" is expressed as **`<=`**. Although it isn't necessary in this case, we should probably still put the different parts of the *OR* statement in brackets to help distinguish them. The final thing to be aware of is that *OR* has a special character in Stata: the vertical line character **`|`** made on UK keyboards by pressing left shift and the key immediately to the right of it (left of z). This takes the place of *OR* the same way an ampersand (&) takes the place of *AND*.

Keeping all this in mind, the code we need to type is:

```
sum age if (height >= 1.7 & height < . & weight < 100) | (weight
< 80)
```



```
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         age |        757    50.55218     17.6578         20         80
```

To reassure yourself that any individual *if* statement you write correctly identifies the observations you want it to, there are several possible checks you could make. This includes summarizing or counting how many observations fit each individual bit of the *if* statement, and seeing whether when you put everything together the total makes sense.

However, one direct way to check is to create a new variable, but with the same *if* statement. This variable will only have values for observations where the *if* statement is true, so you can directly check to see whether a selection of observations where you know if they should be included or not, are in fact included or not.

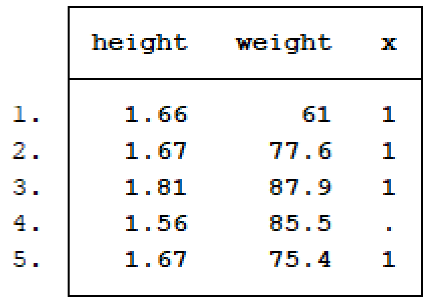
Let’s quickly check to see whether our code worked ok:

```
gen x = 1 if (height >= 1.7 & height < . & weight < 100) | (weight < 80)
(170 missing values generated)
```

Now, Stata says there are 170 missing values, meaning we have 830 observations that satisfy the *if* statement. You may have been expecting there to be 757 observations, since this is how many observations satisfy the *if* statement when we **`summarize`** *age*, but remember there are missing values of *age*, and those won’t have been summarized. I often use *x* as a temporary variable name, because it’s easy to remember that *x* isn’t a variable I want in my dataset and I can remove it later without worry.

To check the if the *if* statement worked as intended, we can either go into the spreadsheet view and skim over the results, or we list the first few observations and check this way:

```
list height weight x in 1/10
```



```
     +-------------------------+
     | height   weight     x   |
     |-------------------------|
  1. |   1.66       61     1   |
  2. |   1.67     77.6     1   |
  3. |   1.81     87.9     1   |
  4. |   1.56     85.5     .   |
  5. |   1.67     75.4     1   |
     +-------------------------+
```

Let’s go through each observation in turn:

1. Height is less than 1.7 so this doesn’t satisfy the first *OR* statement, but weight is less than 80 so this observation satisfies the second *OR* statement, so the *if* statement should be true, and it is because *x* is 1
2. As with the first observation, height is less than 1.7 but weight is less than 80 so it’s correct that x is 1
3. Height is more than 1.7 (and not missing) and weight is less than 100, so this observation satisfies the first *OR* statement, so it’s correct that x is 1
4. Height is less than 1.7 so this doesn’t satisfy the first *OR* statement, and weight is more than 80 so this observation doesn’t satisfy the second *OR* statement either, so it’s correct *x* is not 1
5. As with the first observation, height is less than 1.7 but weight is less than 80 so it’s correct that x is 1

While just looking through observations isn't guaranteed to catch all problems that could occur, it's a good start. Certainly, *if* statements can get very complicated very quickly, so doing a check like this, however brief, could show some problems.

We've finished with the *x* variable now, so can remove it:

```
drop x
```

Now that we can use *if* statements (or make a good go of it at least), let's fix the *sex* variable. If we **`tab`** *sex*, we can see that there's a mix of lower- and upper-case males and females:

```
tab sex
```

```
        Sex |      Freq.     Percent        Cum.
------------+-----------------------------------
     Female |        253       25.45       25.45
       Male |        267       26.86       52.31
     female |        226       22.74       75.05
       male |        248       24.95      100.00
------------+-----------------------------------
      Total |        994      100.00
```

This is a problem, because Stata is case-sensitive, so it is completely unaware that "male" and "Male" are likely to mean the same thing. But we can fix this with a couple of *if* statements:

```
replace sex = "Female" if sex == "female"
(226 real changes made)
replace sex = "Male" if sex == "male"
(248 real changes made)
```

You can see from the output that we've made the right number of changes: there were 226 observations with "female" instead of "Female", and 248 observations with "male" instead of "Male", and those are the numbers of changes we made using the **`replace`** commands. We can **`tab`** *sex* again just to make sure though:

```
tab sex
```

```
        Sex |      Freq.     Percent        Cum.
------------+-----------------------------------
     Female |        479       48.19       48.19
       Male |        515       51.81      100.00
------------+-----------------------------------
      Total |        994      100.00
```

That looks better, now we can easily perform commands split by sex.

3.8 The *by* Qualifier

The *by* qualifier works a little differently to the *in* and *if* qualifiers. Instead of restricting observations to a certain range or if an expression is true, *by* performs the same command on each level of a categorical variable in turn.

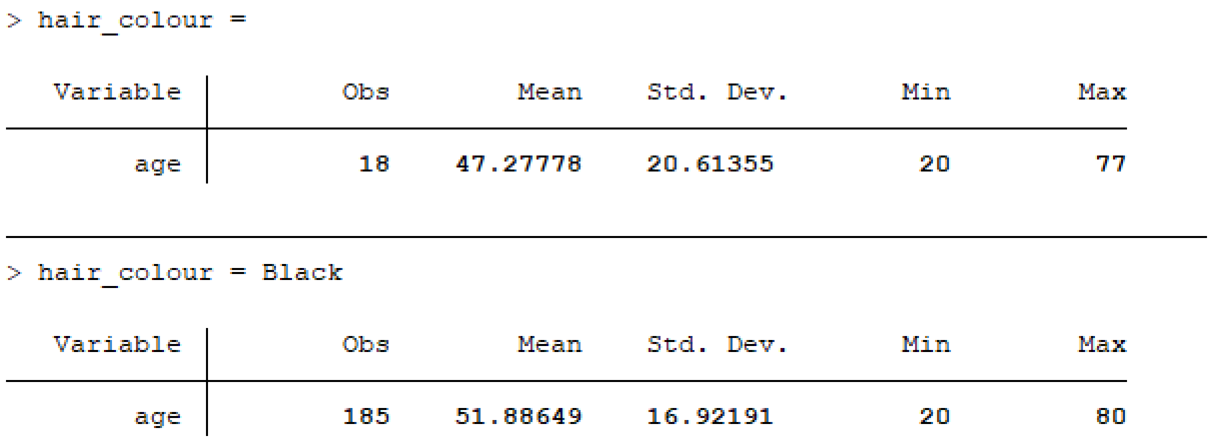
For instance, if you wanted to **summarize** *age* separately for each *hair_colour* separately, then this would take a while using different *if* statements, but with *by* you can do it in one command. The general syntax when using *by* is:

```
by{sort} {categorical variable(s)}: {command}
```

Note that unlike *in* and *if*, that come after the command, *by* comes before the command and is separated by a colon. Before you can use a *by* qualifier, you need to **sort** your data by the categorical variable(s) on which you want to separately run the command. However, Stata allows you to type *bysort* instead of *by*, and Stata will take care of the sorting for you. I find this much more helpful than having to remember to **sort** the data every time I want to use *by*, so I only ever use *bysort*. We'll go through a couple of examples of both, but I'd recommend using *bysort*.

Ok, let's **summarize** *age* separately for each *hair_colour*:

```
sort hair_colour
by hair_colour: sum age
```



```
> hair_colour =

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         age |         18    47.27778    20.61355         20         77

-----------------------------------------------------------------------
> hair_colour = Black

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         age |        185    51.88649    16.92191         20         80
```

I won't copy all the output here, but you can see that for every different hair colour (including for observations with a blank hair colour), Stata has summarised *age* just for people with that hair colour. Essentially, we've just run a series of **summarize** commands with *if* statements saying we only wanted people with each type of hair colour in turn.

We're now sorted by *hair_colour*, so let's sort again by *id* to get back to how the dataset was before:

```
sort id
```

Then do the same thing as above with *bysort*:

`bysort hair_colour: sum age`

```
> hair_colour =

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         age |         18    47.27778    20.61355         20         77

-----------------------------------------------------------------------------
> hair_colour = Black

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         age |        185    51.88649    16.92191         20         80
```

The output is exactly the same, and if you looked on spreadsheet view you'd see that we're now sorted by *hair_colour*, but we've saved a line of code.

You can use *by* with any number of categorical variables: you can also use *by* with continuous variables as Stata doesn't recognise the difference, but it's not advisable as you'll likely end up with as much output as you have observations.

In the next example, let's say that we want to **`tabulate`** *current_smoker* by different levels of *sex* and *income* combined:

`bysort sex income: tab current_smoker`

```
    Smoking |
status, 1 = |
 Yes, 2 = No |      Freq.     Percent        Cum.
------------+-----------------------------------
        Yes |          2       66.67       66.67
         No |          1       33.33      100.00
------------+-----------------------------------
      Total |          3      100.00

-----------------------------------------------------
-> sex = , income = 30,000 to 50,000

    Smoking |
status, 1 = |
 Yes, 2 = No |      Freq.     Percent        Cum.
------------+-----------------------------------
        Yes |          1      100.00      100.00
------------+-----------------------------------
      Total |          1      100.00
```

When more than one variable is specified in the *by* qualifier, Stata will cycle through all values of the last variable specified, then the next last variable, then the next and so on. Here, Stata cycled through all the levels of *income* (starting with the missing values) before cycling to the next level of *sex*, then it cycled through *income* again, then carried on until it finished. This would have required a lot of *if* statements if we had to do this without using *by*. Let's resort the dataset by *id*:

```
sort id
```

Finally, let's the dataset with our initials and the lesson number, and we'll move on to the exercise:

```
save "BP_exercise_03_SH.dta", replace
```

3.9 Exercise

For this exercise, see how you do with the following:

1. Change your directory (if necessary) and load in the **Exercise_02_{initials}.dta** dataset from the first exercise
    a. If you're not confident with your dataset, you can load in the **Exercise_03.dta** dataset instead

2. Sort by age, then list the values of *age* and *height* together in the first 20 observations
    a. Now reverse sort by *height* (i.e. highest to lowest), and list *age* and *height* together in the final 10 observations
    b. Note anything that looks odd
    c. Sort by *id* again

3. Summarise *height* in detail, then convert the values of height that are impossible into missing values
    a. Check that what you did worked by summarising height again and checking the minimum and maximum values
    b. Also set to missing any value of *bmi* created from the impossible heights (hint: all missing values of *height* should have a corresponding missing value of *bmi*)

4. Put *bmi* after *weight*, and *date_of_recruitment* before *age*

5. Preserve your dataset (hint: it's possible to use a do file for this question, but I wouldn't recommend it), then systematically remove data:
    a. Remove *id*, *accommodation*, *exercise*, *calories*, *income*, *marathon* and *smoking_status*
    b. Remove everyone with red hair (hint: remember blue variables are labelled numeric)
    c. Remove everyone who isn't female (lower or upper case)
    d. Keep the *age* and *diastolic_bp* variables only
    e. Remove everyone whose *age* is higher than half their *diastolic*_bp, as well as anyone with missing values for either variable
    f. Count how many observations you now have, then restore your dataset

6. Summarise *age* for people who have black hair or are shorter than 1.7 metres and female (lower or upper case)
    a. This may be tricky, so check whether you've put in the right code by creating a temporary variable including the *if* statement and checking some observations
    b. Then drop the temporary observation.

7. Fix the *sex* variable so all observations start with a capital letter

8. Tabulate *sex* and *marathon* if *bmi* is more than 25

9. Using if statements, create a variable called *bmi_categories*, which is equal to:
    a. 0 if *bmi* is less than 25
    b. 1 if *bmi* is more than or equal to 25 and less than 30
    c. 2 if *bmi* is more than or equal to 30
    d. Then define and apply a value label for *bmi_categories*, showing what 0, 1 and 2 mean, and give the variable a label too
    e. Now order *bmi_categories* after *bmi*
    f. Make sure you check to see if you have generated *bmi_categories* correctly

10. Use tabulate to find out what percentage of people earning £30,000 to £50,000 are female for each level of *bmi_categories*

11. Create a new variable, *sex2*, which is a labelled numeric version of *sex* ordered after *sex*, and give it a variable label too

12. Change the values of *smoking_status* so that 0 = "No" and 1 = "Yes" – check the variable label to see which is which currently
    a. Then update the variable label to reflect the new numbers and meanings
    b. Create a label where 0 = "No" and 1 = "Yes", then apply it to both *smoking_status* and *marathon*

13. We've made lots of changes to variables now, check all the variable labels and fix any that aren't correct

14. Save the modified dataset, calling it **Exercise_{initials}_03.dta**

3.10 Exercise – Answers

1. As before:
    a. `use "Exercise_02_SH.dta", clear`
    b. Or, if you're not confident in your save from exercise 2:
    c. `use "Exercise_03.dta", clear`

2. **`sort, gsort`** and **`list`** commands:
    a. `sort age`
    b. `list age height in 1/20`
    c. `gsort -height`
    d. `list age height in -10/-1`
    e. `sort id`

3. **`summarize`** and **`replace`** commands:
    a. `sum height, detail`
    b. `replace height = . if height == -1`
        i. Don't forget you need a double equals sign for *if* statements
    c. `sum height, detail`
    d. `replace bmi = . if height == .`

4. **`order`** commands:
    a. `order bmi, after(weight)`
    b. `order date_of_recruitment, before(age)`

5. **`preserve, restore, drop, keep`** and **`count`** commands:
    a. `preserve`
    b. `drop id accommodation - smoking_status`
    c. `drop if hair_colour == 5`
    d. `keep if sex == "Female" | sex == "female"`
        i. *OR*
    e. `drop if sex == "Male" | sex == "male" | sex == ""`
    f. `keep age diastolic_bp`
    g. `keep if age <= diastolic_bp/2 & diastolic_bp < .`
    h. `count`
        i. *OR*
    i. Look in the bottom right window or the spreadsheet view
    j. `restore`

6. **summarise**, **generate**, **list**, and **sort** commands:
    a. **sum age if (hair_colour == 1) | (height < 1.7 & (sex == "female" | sex == "Female"))**
        i. The key part of the *if* statement is that the second *OR* statement has a nested *OR* statement: have a careful look at the brackets, because we want it to say "if the person is shorter than 1.7 metres AND is ('female' OR 'Female')"
        ii. If you miss out those brackets, Stata will interpret the statement as "if the person is shorter than 1.7 metres AND 'female', OR if the person is 'Female'", which includes people who are 'Female' of any height
        iii. Liberally using brackets until you're certain you have the right *if* statement is usually the best way to go, and the more of these you do the more intuitive it will become
    b. **gen x = 1 if (hair_colour == 1) | (height < 1.7 & (sex == "female" | sex == "Female"))**
    c. **list age hair_colour sex height x in 1/20**
    d. **sort hair_colour**
    e. **4543**
    f. **browse**
    g. **sort id**
    h. **drop x**

7. **replace** commands:
    a. **replace sex = "Male" if sex == "male"**
    b. **replace sex = "Female" if sex == "female"**
    c. We haven't covered this yet, but there's also a command that capitalises the first letter of all words in a string, and this would also work (we'll cover commands to manipulate strings in another lesson):
    d. **replace sex = proper(sex)**

8. **tabulate** command:
    a. **tab sex marathon if bmi > 25 & bmi < .**
    b. Make sure you account for missing values!

9. **generate, replace, label** and **order** commands:
    a. **gen bmi_categories = 0 if bmi < 25**
    b. **replace bmi_categories = 1 if bmi >= 25 & bmi < 30**
    c. **replace bmi_categories = 2 if bmi >= 30 & bmi < .**
        i. Watch out for missing values
    d. **label define bmi_categories 0 "<25" 1 "25-30" 2 "30+"**
    e. **label values bmi_categories bmi_categories**
    f. **label variable bmi_categories "Categorical BMI"**

g. **`order bmi_categories, a(bmi)`**

10. **`tabulate`** command:
    a. **`bysort bmi_categories: tab sex income, col`**
    b. You could also do this with a series of **`tabulate`** commands with *if* statements, but using *by* is quicker
    c. The answers are as follows:
        i. BMI < 25 kg/m^2 = 54.24%
        ii. BMI 25 – 30 kg/m^2 = 40.74%
        iii. BMI 30+ kg/m^2 = 60.00%
        iv. BMI missing = 63.64%

11. **`generate, replace, label`** and **`order`** commands:
    a. **`gen sex2 = 0 if sex == "Male"`**
    b. **`replace sex2 = 1 if sex == "Female"`**
    c. **`label define sex 0 "Male" 1 "Female"`**
    d. **`label values sex2 sex`**
        i. There's a variable called **`encode`** that can do these steps for you, more on that later
    e. **`label variable sex2 "Sex (numeric)"`**
    f. **`order sex2, a(sex)`**

12. **`replace`** and **`label`** commands:
    a. **`replace smoking_status = 0 if smoking_status == 2`**
    b. **`label variable smoking_status "Smoking status 0 = No, 1 = Yes"`**
    c. **`label define yes_no 0 "No" 1 "Yes"`**
    d. **`label values smoking_status marathon yes_no`**

13. **`label`** commands:
    a. **`label variable age "Age (years)"`**
    b. **`label variable weight "Weight (kg)"`**

14. And finally, a **`save`** command:
    a. **`save "Exercise_03_SH.dta", replace`**