

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 of 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 *diastolic_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.

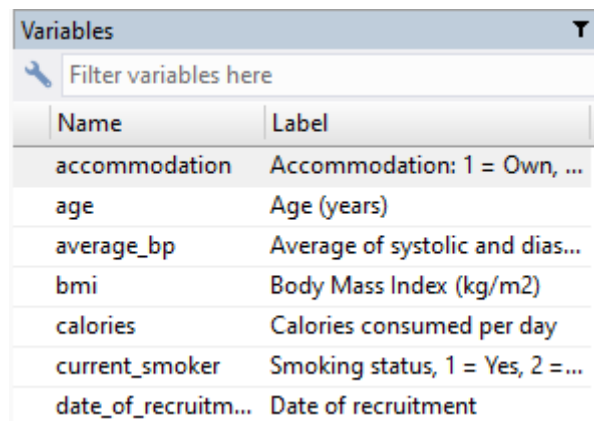
Let's give it a go: to **preserve** a dataset, simply type (in the command bar of Stata, not a do file, for reasons explained below):

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



Variables	
Filter variables here	
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 reason 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/m² = 54.24%
 - ii. BMI 25 – 30 kg/m² = 40.74%
 - iii. BMI 30+ kg/m² = 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**