Introduction to Stata Programming

Erasmus Thesis Project

Armin Hoendervangers
AEclipse ETP Committee

Current version: April 18, 2023[1]

Click here for latest version

[1] This is a work in progress, so the document will have incomplete sections and may contain mistakes.

Contents

—*—

Example Code

—*—

1 Introduction

Reader with advanced tips and tricks for Stata. I'll add some introductory text about the reader here. Incomplete sections will have a very short description of what will be described in them. Could include some information about the thesis project, AEclipse, and maybe myself. I assume some knowledge of and/or experience with both Stata and programming in general. Any questions, comments, or feedback on this reader or AEclipse's Erasmus Thesis Project in general can be sent to thesisproject@aeclipse.nl.

Example code This reader contains several pieces of example code. All code has been written so that it can be copy-pasted to a do-file and run using Stata as is, unless specified otherwise. All example code is also available as separate do-files on the reader's GitHub page here. Note that some of the code in the reader contains automatically generated linebreaks, which might be included if you copy and paste the code into a do-file yourself. If any of the code doesn't work, please check if any command is broken up into multiple lines where it should not be and fix this. If the code still doesn't work, I've likely made a mistake – please let me know if this happens!

Acknowledgements I plan to add some acknowledgements here later.

—*—

2 Information Management

Information is incredibly important in programming, whether it is what a command does, how its used, or what is contained in variables. I take information as a starting point, and this section is all about obtaining and storing various forms of information.

2.1 The `help` command

Perhaps the most important command in Stata, `help` allows quick access to information on *any* command Stata has. The help-files Stata provides are incredibly detailed, including information on how to use the command (its *syntax*), what the command does, its output, examples, and sometimes even the theory behind it. As useful as it is, the help-files might seem daunting at first. Understanding their structure is key to (quickly) obtain information without falling into despair. I'll highlight what I believe to be the most important parts of the help-files through an example.

If I type `help summarize`, Stata opens the window in Figure 1. In help-files, the typography on its own already gives us a lot of information.

Bold words indicate commands or options; if we want to use these, we type them exactly as they are written down. In our case, **`summarize`** is written in bold under the syntax heading. It is, as we know, indeed a command.

Italicised text indicates something that should be substituted. Here, *`varlist`* tells us that we should write down a list of variable names here – should we want to use this option.

Optional arguments and functions are indicated by being [in brackets]. This means that anything that is written within brackets in the syntax is something that does not have to be specified for a command to work.

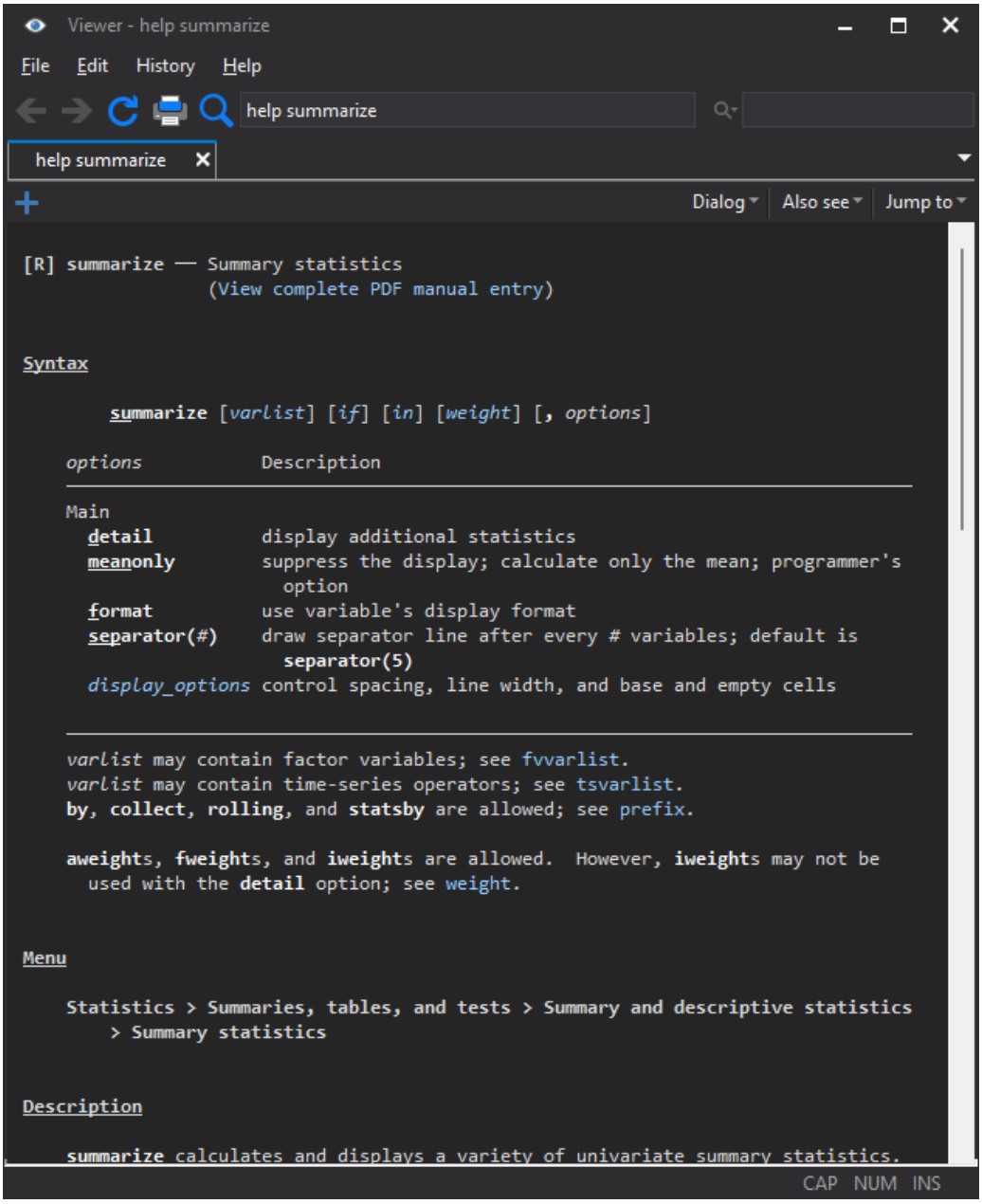
Underlined text indicates the *minimum* abbreviation of a command or option. In the case of `summarize`, I could simply write `su`. Additional letters are also allowed and how to use this is mostly personal preference. Personally, I always write `sum` as it's clearer to me what that means than `su`, but it still saves me the time and space from writing the command out in its entirety. When abbreviating commands, make sure you are familiar enough with them to remember what an abbreviation means if you open your do-files one week later.[2] Having to look it up every time you see an abbreviation can be quite a pain.

Finally, any text in blue is a hyperlink, generally leading to more information on whatever is written down.[3]

[2] To get you used to abbreviated commands, I'll abbreviate most commands as I would do when using them in my own code after I have introduced them. The other reason is that I'm lazy and don't like typing.

[3] Note that the exact colour depends on Stata's colour scheme, but the default and dark schemes do use blue.

Figure 1: Help-file for `summarize`



2.2 Storing Information

2.2.1 Scalars and matrices

Rather than storing information in variables, Stata offers us a couple of different ways of storing information independently of a datset. Scalars and matrices are perhaps the most basic of these options. Other than the names suggest, scalars are not limited to containing only numbers: we are free to store other types of information such as strings in them as well. Matrices can only contain numerical values, on the other hand. Another difference between the two is the amount of information stored: a scalar contains a single piece of information, whereas a matrix can contain multiple pieces of information.

Scalars can be created using the `scalar define` command, although the `define` can be left out. For example, if I want to create a scalar containing the number 7, I could type:

```
scalar define number = 7
```

but also:

```
scalar number = 7
```

After running this line of code – either through a do-file or the Stata console – Stata has now created a scalar with the name "number" that contains the numerical value 7. Naturally, we cannot always remember every piece of information we have stored. If we want to know what scalars currently exist in Stata's memory, we use another command:

```
sca def number = 7
sca second = "two"
sca list
```

After running the second command, Stata returns us a list of all scalars with both their name and value:

```
. sca list
    second = two
    number =           7
```

Note that we can also type `dir` instead of `list` and obtain the same result.

Compared to scalars, matrices are both more versatile and more complicated to work with. As they store multiple pieces of information, every piece of information also needs a position. Creating a matrix is slightly different compared to creating a scalar:

```
matrix input numbers = ( 7 , 2 \ 1 , 2)
```

This creates a two by two matrix (i.e. two rows and two columns), containing the values 7 and 2 in the first row and the values 1 and 2 in the second row. In this command, commas seperate row values while backslashes start a new row. Again, `input` can be omitted when creating a matrix. There is also a `matrix define` command, but this is used when we do computations with already existing matrices. `input` is used when inputting matrices by hand.

To see existing matrices, we use two commands:

```
matrix input numbers = ( 7 , 2 \ 1 , 2)
mat dir
mat list numbers
```

The first of these shows us a list of all matrices in Stata's memory and their size, while the second command shows us the values of the matrix called "numbers":

```
.   mat dir
      numbers[2,2]

.   mat list numbers

numbers[2,2]
    c1  c2
r1   7   2
r2   1   2
```

Finally, we can remove scalars and matrices from Stata's memory using their respective command followed by `drop`:

```
sca drop _all
mat drop numbers
```

We can remove a specific scalar or matrix by specifying its name, or we can remove all existing scalars or matrices by typing `_all` instead of a name. Scalars and matrices are also removed from memory when you close Stata itself or when you issue the `clear all` command.

The `help` file for both commands provide a lot more information on both scalars and matrices, especially so for the latter. Also note that, especially for matrices, a lot is possible using Stata's underlying programming language Mata. Unless you plan on writing

elaborate and complex estimation commands, you will likely never need or encounter Mata; it is thus beyond the scope of this reader – for now.[4]

2.2.2 Macros

Stata recognises two types of macros: `global`s and `local`s. If you are not familiar with the term, a macro is basically a shorthand or abbreviation: instead of repeatedly typing out some very long string of characters, we can define and use a macro instead, saving space, time, and keeping our code much more organised.

Locals Of the two macro types, the local is most common. The major (dis)advantage of a local is that Stata "forgets" it after running the code it is defined in. This means we cannot use locals interactively: if we define a local in Stata's command line, it will be gone by the time we execute a second command. At the same time, this also means we can repeatedly redefine the contents of a local without having to drop it after every time we run a block of code, and, more importantly, locals in our programs won't interfere with locals of other programs.

The basic syntax for defining a local is relatively simple:

```
1 local name contents
```

All we need is to indicate that we are defining a `local`, give it a name, and provide its content. To use the local, we need to tell Stata to expand it in our following command(s). We do this by surrounding the local's name by a single opening and closing quote, i.e. `` `name' ``. Note that you *have* to use the separate opening and closing quote characters, using a single closing quote twice – as you would in a regular text processor such as Word[5] – will generally cause Stata to return an error, as it does not recognise our local as such. In short: write `` `name' ``, not `'name'`.

To make the concept a little less abstract, I'll provide an example. Suppose you have a large amount of regressions or other estimation commands you want to run, all with the same control variables. Instead of typing out all the variable names every time, we can define a local with the variable names and use that instead. Example code 1 does just this, and you can copy the code into a do-file and try for yourself.

[4] At the moment, I do not have a lot experience with Mata yet, either. Although writing a guide on it would likely be a quick way for me to learn it, I do not think it would add much value for this reader. I might change my mind about this later, though.

[5] Most modern text processors automatically change the straight quotes of our keyboard into opening or closing quotes, depending on the surrounding characters. Most programming environments don't: you generally aren't writing text in a programming environment, but code.

Example code 1: local.do

```
// Import example dataset (this is part of Stata!)
sysuse auto, clear

// Define the local
local controls mpg headroom trunk weight length

// Show summary statistics of control variables
sum `controls'

// Run a regression
reg price `controls', r
```

Globals Where a local is a *private* macro available only to the code it is defined in, a global is a *public* macro and is available to other programs or code as well. While this can be very helpful for anything used often in more than just a single do-file or program, this comes with a caveat. Every global name is available only once, i.e. if one program defines a global named "myglobal", any other program attempting to define a global with this name will overwrite the previous contents of "myglobal". This can therefore interfere when running code written by others: if their code uses or defines a global with the same name as one of your globals, either their code or yours will likely not work as intended. It is thus best practice to avoid using globals wherever possible, e.g. by using locals instead.

Nevertheless, globals do have their uses. An example would be to define the path to the folder in your current project, so you can use the global instead of typing out the entire path every time you would refer to some file.[6]

The syntax for globals is slightly different compared to locals, but they are otherwise handled in the same manner. To define a global, we type `global name contents` and we can expand a global by affixing a dollar sign before its name, like so: `$name`.

General remarks and usage tips Before moving on, I'd like to highlight several other ways to manipulate or define macros. Instead of simply storing information we spell out, a macro can also be defined through an *expression* or through a *macro function*. To put it simply, using an expression tells Stata to evaluate the expression and store the result, while a macro function tells Stata to obtain the information defined through the function. The syntax for using an expression is `local name =expression`, and the syntax for using a macro function is `local name : function`.

To illustrate the use of an expression, consider the following code:

[6] While this helps in making sure your code always uses the correct files, you can also open Stata from the project folder and refer directly to file names. The downside of the latter method is that you always need to make sure Stata has the correct working directory when executing your code. You can check and change the current working directory with the commands `pwd` and `cd`, respectively.

```
loc sum 1 + 1
loc sum2 = 1 + 1
```

When we evaluate these, the first (`` `sum' ``) will expand to `1 + 1`, while the second (`` `sum2' ``) will expand to `2`. By using an expression, we basically told Stata to calculate 1 + 1 and store the result of that. The reason Stata does not evaluate the contents in the first local is that unless specified otherwise, Stata implicitly places double quotes surrounding the contents. In other words, writing `local name contents` is exactly the same as writing `local name "contents"`. Of course, there are cases where we would want to manually add quotes, but I'll get into the details of using double quotes in Stata programming later. For a list of available macro functions, see `help macro##macro_fcn`.

After defining a macro, we can manipulate it in the same manner as defining it. To do this, we simply define the macro again: `local name contents`. This will simply throw away the previous contents and store whatever new contents you define. If we want to keep the previous contents, we can expand the current local inside its new contents: `local name `name' newcontents` or `local name newcontents `name'`. This would add new content to the local after or before the current contents, respectively.

We can also use locals as a sort of counter. If we define a local as an integer, we can use a shorthand for incrementing the local. For example:

```
1 loc counter 1
2 loc counter = `counter' + 1
3 loc ++counter
4 loc counter++
```

After defining the local in the first line, lines 2–4 all increment the local by one, so that the local expands to 4 if it is used afterward. This is mainly useful when using a local inside loops, which we will get into later in this reader.

In general, macros are extremely flexible, especially if we combine the several ways of manipulating them. I wholeheartedly recommend practicing a bit in using them, as they can save an enormous amount of time spent coding – both by reducing the amount you need to type out and by making your code much more readable.

2.3 Inspecting and Obtaining Information

Now that we now how to store information, we also need to consider how to obtain information or inspect what information has been stored. I've already covered how to find information on commands (Section 2.1), and you've likely inspected a dataset before using the `describe` or `browse` commands. These are all very helpful, but it is now time to delve into handling nitty-gritty specific pieces of information.

2.3.1 The `display` command

The `display` command is one of – if not the – most used commands for me while programming. It's function is quite simple: it displays whatever you tell it to in Stata's output window. Furthermore, it evaluates whatever you tell it to display (unless specified otherwise), so you can also use it as a calculator if you so desire. The basic syntax is as follows: `display contents`. If we insert any sort of calculation, display gives us the results: `di 1 + 1` returns `2`. If we do not want the contents to be evaluated, we enclose them with double quotes. As an example, let's define a local containing an expression and display it with and without double quotes. Code for this is in Example code 2.

Example code 2: display.do

```
1 // define local
2 loc expression (1 + 1) * 3
3
4 // display without double quotes
5 di `expression'
6
7 // display with double quotes
8 di "`expression'"
```

If you execute this code, you'll see that the command in line 5 returns `6`, while the command in line 8 returns the expression as we wrote it: `(1 + 1) * 3`. The latter is especially useful if we automate the manipulation of locals and want to see if the contents are as expected. Note also that if we want to display a string of text we *need* to enclose it with double quotes as well, otherwise Stata will interpret it as the name of a stored object, such as a variable or scalar.

Quotes The more we start automating things, the likelier it will be that one of the strings we store will itself contain quotes. As both the starting and the ending symbol of quotes are similar, this can quickly produce unintended results. Suppose we create a local with such a string and tell Stata to display it:

```
1 // define local
2 loc quote Using quotes without thinking is a "wonderful" idea.
3
4 // display the local
5 di "`quote'"
```

If we run this code, Stata returns an error. To see where this goes wrong, we can manually "expand" the local and see what we *actually* told Stata to do:

```
di "Using quotes without thinking is a "wonderful" idea."
```

The problem here is that we did not provide a single string of text, but two, with some word, `wonderful`, in between. Stata does not recognise what "wonderful" means and thus doesn't know what to do. The issues is that Stata does not know the order or hierarchy of the double quotes we've written: they're all the same character. Luckily, there's a fix for this: *compound* double quotes. Instead of just writing "normal" double quotes, we indicate whether it is a starting or an ending double quote. To do this, we add a single opening or closing quote – the same characters used for local expansion. Let's build on the previous example and add the "correct" command in Example code 3 to illustrate. To make sure the code runs, I've commented out the incorrect command.

Example code 3: quote.do

```
// define local
loc quote Using quotes without thinking is a "wonderful" idea.

// display the local
* di "`quote'"


// correctly display the local
di `"`quote'"'
```

As your code (and locals) become more complicated, all these different kinds of quotes can quickly make it difficult to see where everything starts and ends. It took me ages to completely understand how and when to use these compound double quotes, so don't worry if looks like abracadabra – it often still does to me. Practice makes perfect.

Display options The `display` command also has options to format the output. While they don't make much of a difference for things like troubleshooting, they can be nice if you're writing an extensive program and want to differentiate between specific types of output. The options and an example of their use are provided in the official help file: `help display`.

2.3.2 Results of other commands

While working with Stata, you probably don't want to write every bit of code from scratch: there are already tons of useful commands available – shipped with Stata or written by other users, so why not build on those? Luckily, Stata stores lots of information obtained and produced by any well-written command. In general, (almost) all Stata commands are either *r-class* or *e-class*, corresponding to general commands and estimation commands,

respectively. General commands store their results in `r()` objects, while estimation commands store their results in `e()` objects. To access these results, we need to know the name of the object they are stored in. Of course, there is a command to find out the names of the available objects. For general commands we use **`return list`**, while we use **`ereturn list`** for estimation commands. There is also a *c-class* with `c()` objects, but they are not related to commands and always available. We can use **`creturn list`** for these. Example code 4 provides an example how to use this.

Example code 4: return.do

```
// load example dataset
sysuse auto, clear

// summary statistics of weight
sum weight

// list available results
return list

// report one of these results
di as text "Average weight is: " as result r(mean) as text " lbs."

// simple regression
reg price mpg headroom trunk weight length, r

// list available results
ereturn list

// report one of these results
di as text "On average, one lbs. extra weight leads to a price increase of " as
     result e(b)[1,4] as text ", ceteris paribus."
```

Note that we do not have to use the various return commands every time we want to access one their corresponding objects; they simply list what is available. In a similar manner to how I display the contents of the objects in Example code 4, we can also store them in locals or use them in pretty much any other way you can think of.

—*—

3 Automation

In this section we'll go over several commands that can be very useful for automating certain bits of code.

3.1 Grouped Command Execution

One of the easiest ways we can repeat a certain command for different groups of observations is with the `bysort` prefix. This prefix lets us run the command we use it with for every group defined by a variable separately. In my experience it's mostly useful for generating variables in programming, but it can also be used as a quick and dirty way to compare variables across groups. We can use the prefix like so: `bysort varlist: command`, where `varlist` is a list of the variables – or a single variable –identifying the different groups, and `command` is the command we would like to run. Note that `bysort varlist:` is equivalent to using `by varlist, sort:`. The `by` prefix does not work without sorting the data, so it is generally easier to just use `bysort`. Example code 5 provides an example.

Example code 5: bysort.do

```
// load example dataset
sysuse citytemp4, clear

// Summary statistics of temparature in January for each division
bysort division: sum tempjan
```

3.2 Conditionals

Conditionals, or if-statements, are where the real fun stuff begins. To put it simply, they allow us to differentiate our code based on anything we can turn into an expression that evaluates to true or false. Stata recognises two types of if-statements: one as a command suffix, and one for programming. In this section, we'll first go over expressions before we move on to the two types of if-statements.

3.2.1 Expressions

When we use conditionals, we set requirements that must hold for code to be executed. An expression can then be seen as a check whether these requirements are fulfilled. An expression always evaluates to true or false: the requirements are met, or they are not. In programming, we refer to a data type that has either the value "true" or "false" as a *boolean*. An evaluated expression is precisely that. In programming, true and false are

often represented by 1 and 0, respectively. This is also the case in Stata: if we look at dummy variables, for example, they function in much the same way. A dummy variable for gender is often coded in such a way that it represents either male or female, such as a variable `female` with value 1 for females and value 0 for males.

Expressions are much like mathematical equations, in that they have a left-hand side, a relational operator, and a right-hand side. Based on the relational operator, the left-hand side is compared to the right-hand side, and the expression is evaluated to be true or false. Let's take a look at an example. Suppose we have two scalars, a and b, and want to check whether these are equal to one another. First, we need to have these scalars defined. Let's say that $a = 3$ and $b = \pi$:

```
// define scalars
sca a = 3
sca b = c(pi)

// inspect the values of the scalars
di "Scalar a holds value " a
di "Scalar b holds value " b
```

We then want Stata to tell us whether $a = b$:

```
// show whether a and b are equal
di a == b
```

Of course, clever as we are, we know this to be false. This expression should therefore evaluate to false, i.e., 0. When we run this code, we see that, indeed, the last command returns 0 (see Figure 2).

Of course, we don't always want to know whether things are equal to one another. Luckily, there are more relational operators than just `==` (equals). To see the full list of available operators, type `help operator` in the command window. Furthermore, we may want to set more than one requirement. Luckily, we can combine multiple requirements using logic operators – also listed in `help operator`.

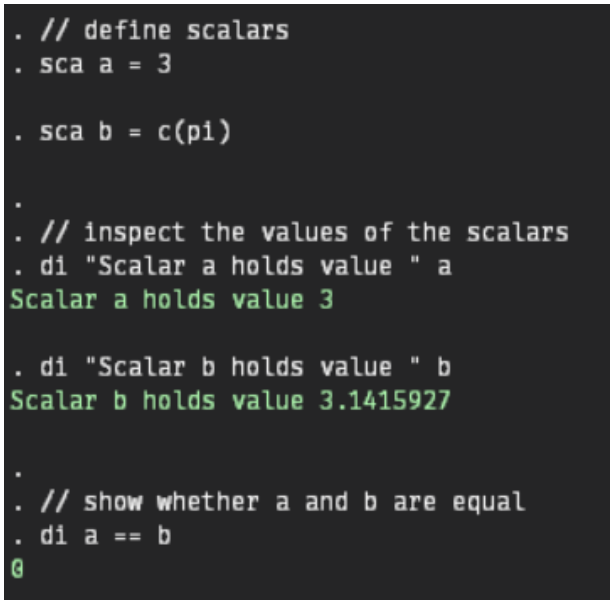
Suppose we now want to know whether $a \leqslant b$. We could then tell Stata to tell us whether $a = b$ *or* $b > a$ is true:[7]

```
// show whether a and b are equal, or b is larger than a
di a == b | b > a
```

As $\pi > 3$, this will evaluate to true. Try for yourself using the code in Example code 6!

[7] Of course, this can also be achieved using a single $\leqslant$ operator, but for the sake of the example I don't do that here.

Figure 2: Stata output for expression code



Example code 6: expression.do

```
// define scalars
sca a = 3
sca b = c(pi)

// inspect the values of the scalars
di "Scalar a holds value " a
di "Scalar b holds value " b

// show whether a and b are equal
di a == b

// show whether a and b are equal, or b is larger than a
di a == b | b > a
```

3.2.2 Command suffix if

The command suffix if statement is the simpler of the two. By adding an if-statement to the end of a command (but before the options!) we tell Stata to only use the specified subset of our data. Suppose we have some individual-level data with a dummy variable `female` indicating an individual's gender, i.e. it is 1 for females and 0 for males. We can then tell a command, such as `summarize`, to only use the observations for female individuals by typing `sum varlist if female == 1`. As we're using a dummy variable, we could even shorten this to `sum varlist if female`! This is because our expression, `female == 1`, evaluates either to true or false for each observation. A variable containing either true or false is also known as a *boolean* and is generally stored as either a value of 1 (true) or 0 (false). Dummy variables are structured in the same way: they have a value of 1 if something is true, such as an individual's gender being female, or 0 if false. We can thus exploit this to make much more compact if-statements.

We can also use non-binary variables for our if-statements, which allows us to use the full breadth of Stata's relational operators. Furthermore, we can combine several if-statements using logic operators, such as and, not, and or. For the specific characters of each operator, see `help operator`. The key takeway here is that any expression following `if` will evaluate to either true (1) or false (0). If the statement evaluates to true for an observation it is used, and if it evaluates to false it is left out.

On this note, a quick tip for generating dummy variables. Often people (including past me) do this in a roundabout way:

```
generate dummy = 1 if expression
replace dummy = 0 if dummy == . // missing
```

As any if-statement already evaluates to either 1 or 0, it is much simpler (and cleaner) to write:

```
generate dummy if expression
```

If you work with a lot of dummy variables, this will save a lot of time!

3.2.3 Programming if

Stata's programming if-statements have a multitude of uses. They allow us to execute bits of code only if a specified expression is true. For a single command, the syntax is as follows:

```
if expression command
```

where the expression works in the same way as in the suffix if-statement. In programming if-statements, you probably won't be using variable names, but rather locals or scalars that you defined previously. Of course, you are free to do as you like: locals and scalars can also be used in suffix if-statements, and variables can be used in programming if-statements, although the latter will generally include an observation number to identify a specific value.[8]

We can also include multiple commands in a single if-statement. The syntax for this is slightly different:

```
if expression {
    command1
    command2
    ...
}
```

Should the expression evaluate to true, Stata will execute all commands enclosed by the curly brackets. Note that no command may follow the opening curly bracket (comments are fine) in the same line, and the closing curly bracket must have a line just for itself. While not necessary, I recommend indenting the commands inside a code block like this to keep your code organised.

After a programming if-statement, we can follow up with `else`. The code following an else-statement executes when the if-statement is evaluated to be false. An else-statement is written in the same way as an if-statement, except no expression is given. Using both looks like this:

```
if expression {
    commands // these are executed if true
}
else {
    othercommands // these are executed if false
}
```

3.3 loops

different types of loops

—*—

[8] I.e. by writing `varname[number]` instead of simply `varname`

4 Custom Commands

Section on how to write a custom command.

4.1 program

defining a program

4.2 arguments

adding user input to a program

4.3 syntax

adding standard Stata syntax to a program

4.4 temporary variables

using temporary variables

4.5 output

defining program output

—*—

5 General Tips

Things I found helpful in using Stata.

5.1 Coding stuff

different types of comments
- linebreaking commands
- viewing multiple do-files side to side
- do-file editor settings

5.2 Customising Stata

changing the font
- changing Stata's colour scheme

5.3 Project management

folder structure
- best practice for naming
- multiple do-files
- profile.do

5.4 Making nice output

graphs
- tables