Introduction to Stata Programming

Erasmus Thesis Project

Armin Hoendervangers
AEclipse ETP Committee

Current version: February 24, 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. Example do-files will be available on the reader's GitHub page. Comments and feedback are always welcome.

—*—

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.

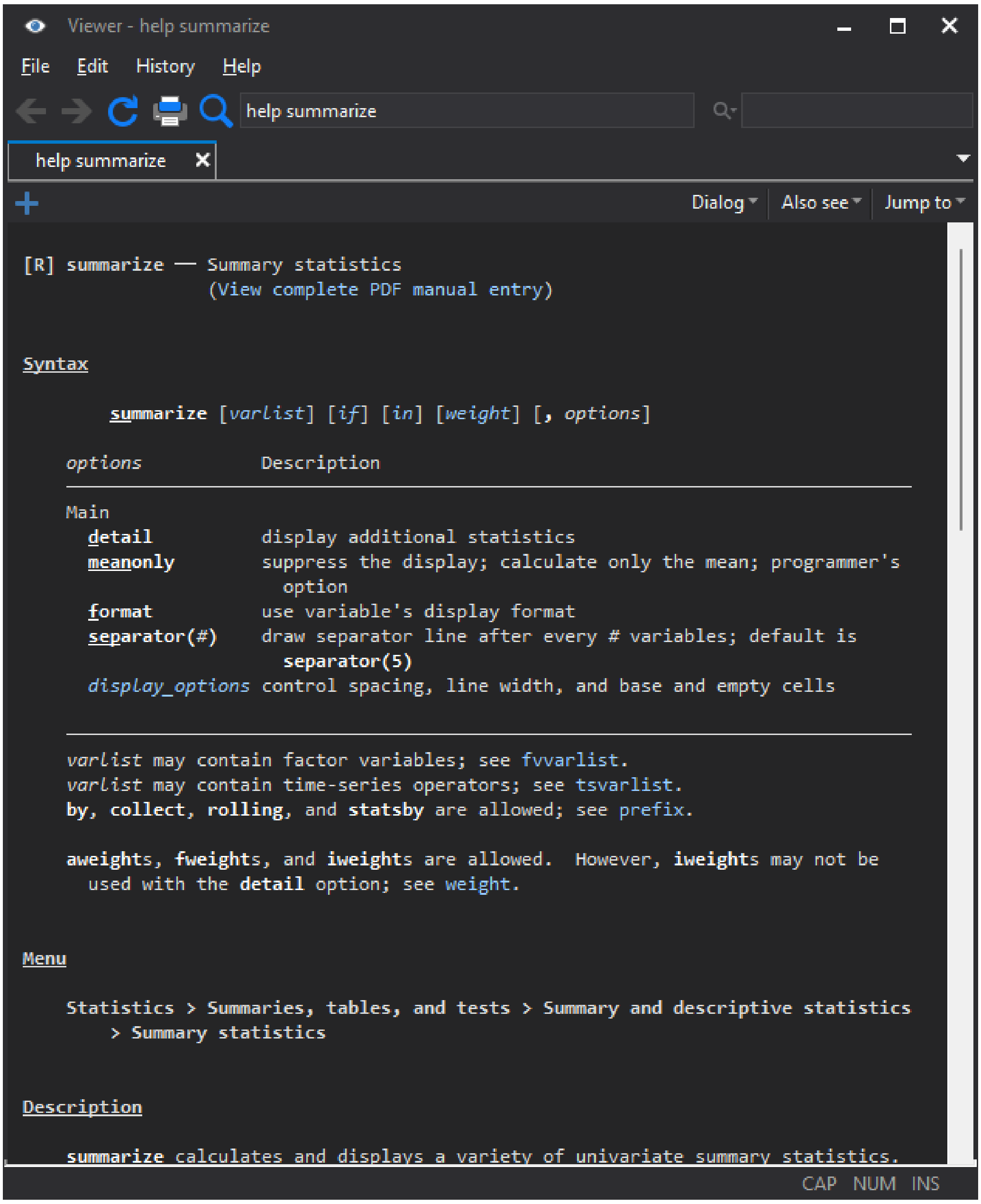
Optional arguments and functions are indicated by being [in brackets]. This means that anything that is written within brackets in the syntax is something that does not have to be specified for a command to work.

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 its clearer to me what that means than `su` would, 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. 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.[2]

[2] 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`



```
[R] summarize — Summary statistics
                  (View complete PDF manual entry)


Syntax

        summarize [varlist] [if] [in] [weight] [, options]

    options            Description
    ------------------------------------------------------------------------------
    Main
      detail           display additional statistics
      meanonly         suppress the display; calculate only the mean; programmer's
                         option
      format           use variable's display format
      separator(#)     draw separator line after every # variables; default is
                         separator(5)
      display_options  control spacing, line width, and base and empty cells
    ------------------------------------------------------------------------------
    varlist may contain factor variables; see fvvarlist.
    varlist may contain time-series operators; see tsvarlist.
    by, collect, rolling, and statsby are allowed; see prefix.

    aweights, fweights, and iweights are allowed.  However, iweights may not be
      used with the detail option; see weight.


Menu

    Statistics > Summaries, tables, and tests > Summary and descriptive statistics
        > Summary statistics


Description

    summarize calculates and displays a variety of univariate summary statistics.
```

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
```

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:

```
scalar define number = 7
scalar define second = "two"
scalar list
```

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

```
. scalar 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)
matrix dir
matrix 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":

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


.   matrix 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` :

```
scalar drop _all
matrix 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.[3]

[3] 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.

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:

```
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[4] – 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.[5]

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

[4] 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.

[5] The code is also available to download as a do-file on the reader's GitHub page here. Later example codes will all be available to download there as well.

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
```

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:

```
local sum 1 + 1
local 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

[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.

+ 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 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 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`.

—*—

3 Automation

This section is about differrent ways of automating things.

3.1 Conditionals

3.2 loops

different types of loops

—*—

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.