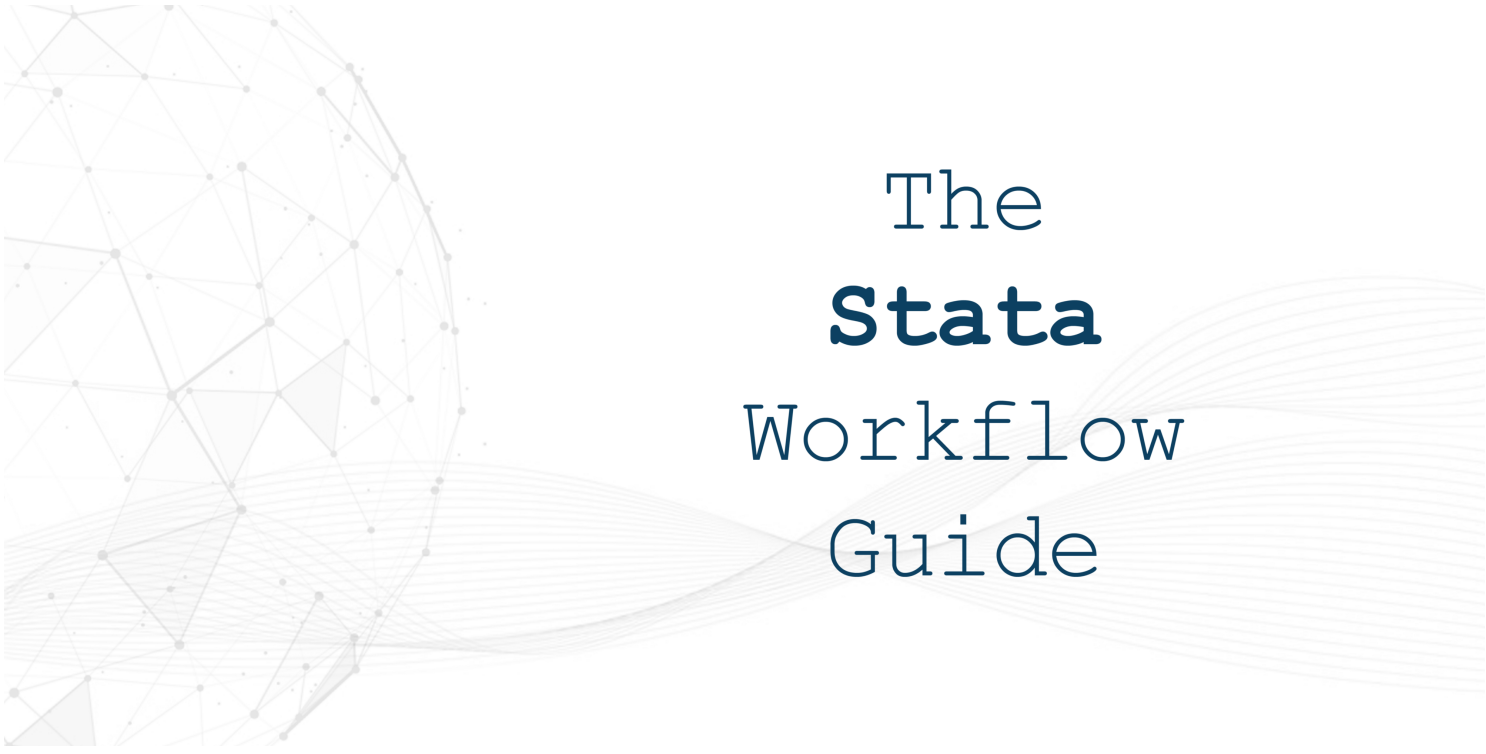
The Stata workflow guide

Asjad Naqvi Follow

Jun 7 · 24 min read ★

While programming in Stata or even in other languages, it is often easy to forget about structuring your files and folders. Most of the time this is left as a last step, usually when finalizing a thesis or some paper for submission. I have also heard researchers often say that once they get some revisions, then we will eventually organize the stuff in the next round. Workflow management is extremely important and organization should be done as a **first step** otherwise a lot of time is lost later figuring out what you actually did, where the files actually are, and where the latest code is saved. This also makes it easier not only for yourself but also for your co-authors and collaborators or even people trying to replicate your code. A good litmus test of a good workflow management system is that when you open an old project, you can immediately figure out where all the files are.



In this guide, I will cover workflow management of files and data scripts or dofiles. This article will not discuss coding but rather how to structure your code. The broad workflow tips discussed here can be extended to any project in any software. Additionally, these are not hard rules but guidelines to get your started with organizing your life around research work, data management, and analysis.

The guide is in six parts:

- **Part 1**: Folders
- **Part 2**: Naming conventions
- **Part 3**: Splitting tasks across dofiles
- **Part 4**: Making use of relative file paths
- **Part 5**: Styling your code
- **Part 6**: Linking different dofiles

So let's get started!

Part 1: Sort EVERYTHING into folders

Over the course of a project life-cycle one accumulates files. Just on the analysis side this includes getting raw data, scripts to process the data, putting together data for analysis, generating tables, graphs, and figures. And then there are versions of all of these.

Around the data part is also the writing part which includes project documents, literature folder, and your Word or LaTeX files and their versions. ALL of this should be sorted in folders. If you have followed my guides, I always suggest following this sort structure for your data projects:

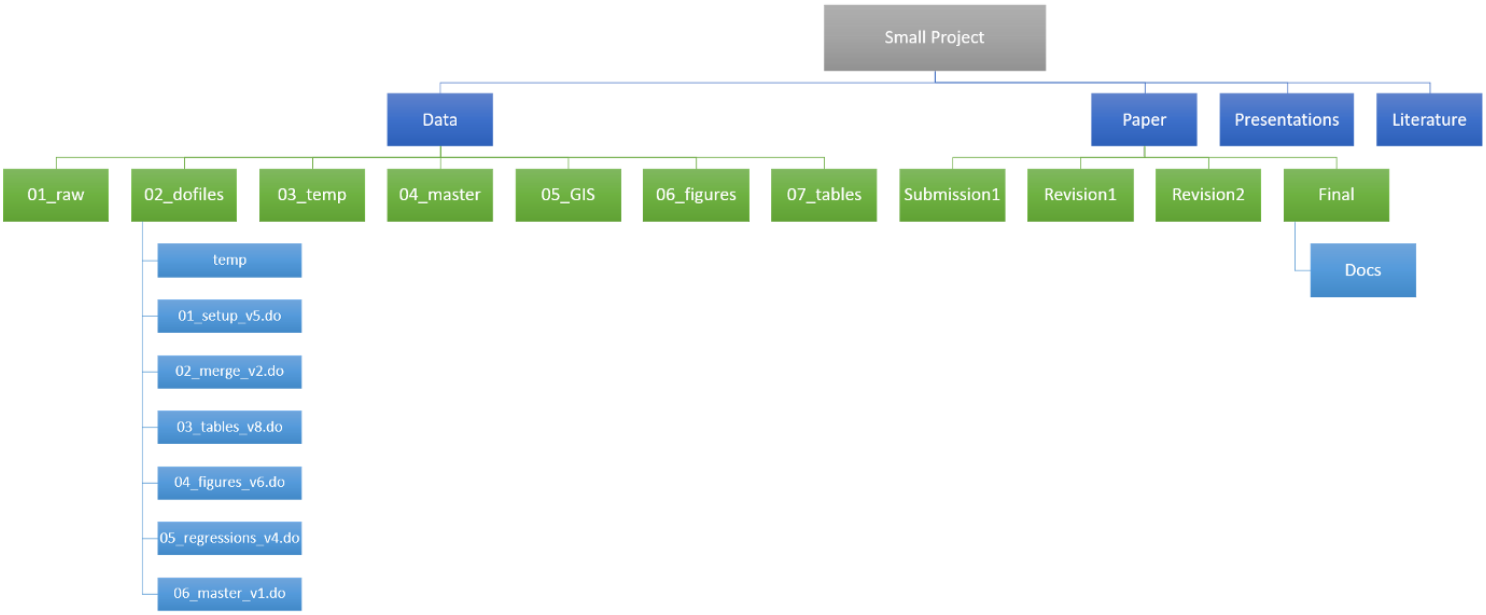
Here the names are obvious:

- **raw**: contains all the raw files
- **dofiles**: contains all the scripts to process, clean, and analyze the raw files
- **temp:** contains intermediate files that are generated from the raw data
- **master:** contains the final data that is ready for analysis
- **GIS:** contains the spatial layers
- **graphs:** contain the figures

This list is not exhaustive and it is good to get you started but here we need to go a step further to add more control. Here, I would like to differentiate folder organization between small projects (e.g. single papers) and large projects (multiple papers).

Organizing small projects

For small projects or standalone papers, I recommend a folder structure that looks something like this:



First of all, number the folder in the order they are used. For example, you will get raw data first (**01_raw**), then you will write the scripts (**02_dofiles**). While writing the scripts, you will create temporary files (**03_temp**), before that are saved in the master folder (**04_master**). You might also have spatial data that can do in its own folder (**05_GIS**). Once the data is clean, it can be used to generate tables (**06_tables**) and figures (**07_figures**).

Adding a number prefix to folders keeps the sequence in check. Plus if anyone uses this folder, they can also sort of follow the logical order of the workflow. Numbers also sort the files in numerical order, otherwise they appear alphabetically which really doesn't tell much about the sequence. This can be problematic even with your own old projects. Plus quite a few times, I have downloaded replication files, and folders are placed in an alphabetical order, which can be confusing since they don't correspond to the workflow.

TIP: *Never have spaces in folder or file names.*

While this is not so much of an issue now since operating systems can deal with them, older systems would throw errors. Sometimes you even see spaces replaced with %20 in URLs. Similarly other characters like `~#@!\%^&` and `-` (basically most of the symbol) should be avoided at all costs. Most of these are used for programing and usually require some special treatment to be included in scripts. In short, skip the special characters.

TIP: *Never use hyphens (-) in folder or file names. If you need to provide spaces use underscores (_).*

Technically Stata can read names with spaces if they are in double quotes `""`. But as a good practice use underscores whether it is for folder names or file names.

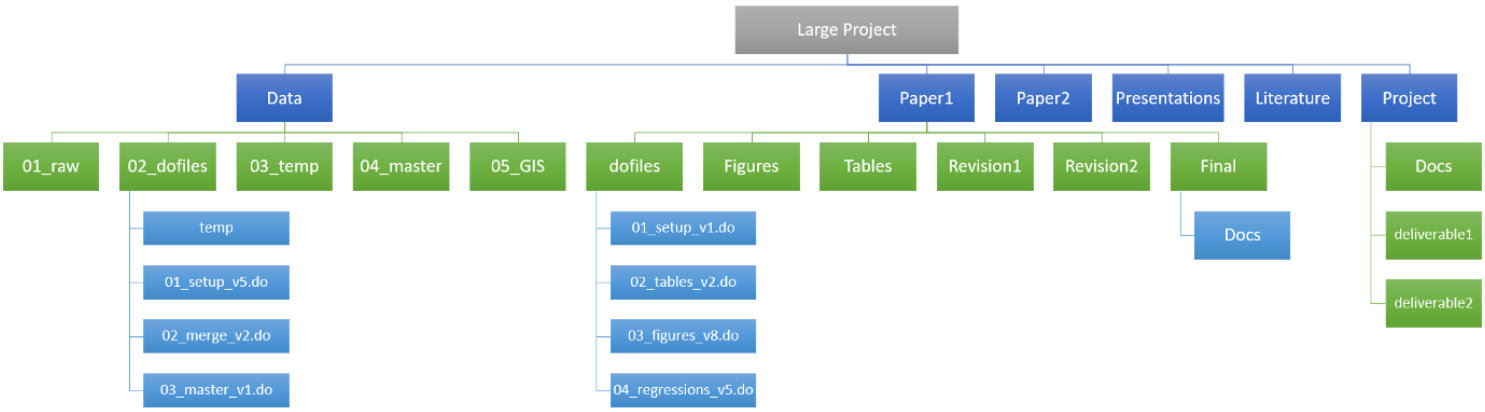
TIP: *Avoid capital letters in names.*

While the screenshots above have capital letters (thank you Powerpoint!), I would avoid having caps in names. Again capital letters work, but it is usually considered bad practice. The aim of scripting and programming is to develop a flow, and things like capital letters also affect the speed. Even if I see capital letters in variables names I just make them lowercase. Properly formatted text should be left for variable and value labels since they are used in graphs and tables.

Organizing large projects

If you are working on large projects, which involve multiple datasets, several papers, and project deliverables, then I would suggest splitting the data setup from the analysis part.

For large projects, I use this type of a folder structure:



Here the “data” folder is just raw data and scripts that clean up the raw files. The aim here is to generate the final data files in the master folder. This setup works well for projects that are massive in terms of data handling and processing. For example, one project I am working on has fairly large spatial files that are merged with micro census

data. The data size is several gigabytes and the processing can take several hours with scripts usually running overnight.

The aim of the scripts in the data folder is to put everything together after doing all sorts of spatial merges, cleaning, processing etc., which can have their own set of challenges. Therefore, using a separate folder for data can also help document the steps neatly and most of the times, once the final dataset is created, the large scripts that put the data together sort of become redundant unless some modifications are required. Once the master file is created, this portion of data management should basically remain untouched.

The master data files are picked up by paper-specific dofiles in the "paperX" folders. The paper folder follows the same structure the small project folder. It also contains a setup file since where one can merge, append, modify, the data files to create a paper-specific master datafile. Here essentially the master dataset in the "data" folder play the role of raw data for the paper folder.

TIP: *Throw out the variables you don't need for specific papers. Every unused data point is a burden on the memory.*

Useless variables that are loaded in the dataset will take up memory and if the file is large, your computer will feel it as well. Plus this also helps with the analysis part especially if one doesn't have to scroll through some 200 variables.

TIP: *Write up a readme.txt file.*

This might seem a pain, but leaving readme files help a lot! For example, they include information about folder and files and the sequence and order on how to use them. More importantly they can also contain information on where and how the data was accessed. Here I would even suggest to leave notes in your dofiles on URLs from where one can find the raw files. These could be links or even table numbers from large datasets like the World Bank, Eurostat, OECD etc. There have been times, when I have opened some old projects and could not figure out where to update the data from. This seems trivial during the data cleaning process but one forgets later on!

Part 2: Get those filenames RIGHT

You can see in the screenshots above that the dofiles are also prefixed with numbers and post-fixed with version numbers. Again the same logic of sorting everything in order applies here. Everyone uses they own file naming convention. Having numbers prefixed to dofiles helps sort them out and align them with the workflow. Similarly post-fixed files with versions also helps you keep track. Here you can see that I use _v1, _v2, etc. This can be replaced with dates as well. For example, in some projects, I will have 01_setup_210531.do, 01_setup_210602.do etc.

TIP: *When using dates follow yymmdd (year month date) format. This will automatically sort the files in the correct order with the latest one showing first.*

Here I would reiterate to avoid using the *ddmmyy* (date month year) format or the American *mmddyy* (month date year) format. The aim here is to have files in the correct order and not have hang-ups about the date formats.

TIP: *Hide the older file versions.*

Don't collect a gazillion versions in the same folder. At some point, older versions will be redundant or useless. Delete them. Or if you are a collector (like me), throw them in a temp folder. I always have a temp folder for old dofiles, tables, graphs. Keep your main folders and sub-folders as neat as possible. So if you are working on a project for months, then do some **housecleaning** once in a while. There is a good chance that those versions from a few weeks back won't be used.

The naming convention should not be limited to just dofiles. Do this for all the graphs, tables, paper versions etc. It might seem like a pain but in retrospect you won't regret it especially if you don't have to sift through hundreds of graphs to find the ones you really need.

Part 3: Different dofiles for DIFFERENT tasks!

The dofiles should have a sequence and order. I myself use variations of the following structure:

- 01_setup_v4.do
- 02_merge_v2.do

- 03_tables_v11.do
- 04_regressions_v15.do
- 05_figures_v2.do
- 06_master_v1.do

The names are self-explanatory. The first dofile picks the raw data, cleans it up and dumps it in the temp folder. Here one can process all the raw files or even have different setup files for different raw files. For example, I can have

- 01_setup_emissions_v1.do
- 01_setup_economic_v4.do

Since they are prefixed with 01_, they will be clustered together in the folder. The second file 02_merge takes all the datasets and puts them together to create the master file. Everything up till here is data cleaning. Unless you are very lucky and have a very clean set of files, this will be 60–70% of the time spent with the data. You can also do version control with dates but remember to stick to the *yymmdd* rule.

Some authors also prefer to use a different dofiles for different tables and figures etc. This is absolutely fine. What is important to remember is that the file name should reflect the figure or table number. For example:

- 03_table1_v1.do
- 03_table2_v4.do
- 04_regression1_v2.do
- 04_regression2_v6.do

and so on.

At the beginning of a project the sequence of tables figures are usually not clear. It is mostly at the end of the project when things start getting finalized and the sequence of outputs is sort of defined.

Some tips for starting new projects

The initial phase of projects is usually a messy one even if everything is structured in folders etc. Hence dofiles tend to be messy as well. For example, an early stage workflow might look like this:

import data > generate some variables > make some figures > generate more variables > some more figure/tables > modify some data> do some complex loop to generate regressions.

Here I would suggest two things:

TIP: *Partition the code within the same dofile*

If you are generating different figures/tables from the same dofile, write it up such that you can split it up later into different code blocks within a dofile. Or split it across different dofiles.

TIP: *Move the variable generation to the beginning of the dofile or ideally in the setup or merge dofiles.*

Ideally, the core variables needed across all the files should go in the setup file. Or if you are generating variables from different files, they can also go in the merge.do file after the cleaned files are put together, and before the final master datasets are saved. If you do need to generate variables in figure/table specific files (for example, if you are collapsing or reshaping data), then don't do it in the middle of the dofile, bunch them all up together at the beginning of the dofile and comment them as much as possible.

The master dofile

The last dofile *_master_*.do serves one purpose and that is to run all the previous dofiles to compile everything for the project. It has a structure which should look like this:

```
**** <some project info here> ****

clear
<some directory setting stuff here>

<some paths to folders. usually defined as globals>
```

```
<other macros for tables/regressions here>


// run all the dofiles

do ./dofiles/01_setup_v11.do
do ./dofiles/02_merge_v2.do
do ./dofiles/03_tables_v4.do
do ./dofiles/04_regressions_v5.do
do ./dofiles/05_figures_v2.do

*** END OF FILE ***
```

The aim of your project is to reach this file so you can run everything in one go without breaking the data. In order to do this, it is important to use **relative paths** which we discuss below.

Part 4: Use relative paths in dofiles

Relative paths are key to a smooth workflow. Here the idea is that you should set your main directory once. And everything should be relative to that directory. It is sometimes ok to move in and out of subdirectories but it is better to avoid it!

So what do we mean by all of this. Let's go back to the vanilla folder structure shown earlier:

Let's assume that the main folder is called "myproject" and it is on your D: drive (on Windows). We can point to this folder in the dofile as follows:

```
clear
cd "D:/myproject"
```

Note here that I use `clear`, which clears everything in the memory. Then I specify `cd <folder path>`. Also note here that I use forward slash `/`.

TIP: *For file paths ALWAYS use forward slash* `/`.

Backslash `\` is reserved for special operations that can sometimes causes errors under some circumstances. It can work but don't use it!

The folder name has no spaces or other characters so the double quotations are not necessary. But if the path is enclosed in double quotes, it should not matter.

Here I would like to point to another scenario. Assume you have your files synced on Dropbox and you are working across multiple computers (e.g. home and office). It is very likely that the paths will be different across the computers. While there are packages to find the Dropbox main folder path, a simple trick is to just use Stata's `capture` command:

```
clear
cap cd "C:/Program files/Dropbox/myproject"        // home
cap cd "D:/Program files (x86)/Dropbox/myproject"  // office
```

The `capture` or `cap` command skips Stata errors and moves on to the next line. Without capture Stata would just give a red error code on the screen and will stop the script. This makes `cap` an extremely risky command and it should be used with caution especially when doing analysis.

In the above script what we are telling Stata is to see if it finds the first path, and if it doesn't then it should move to the next line and try and set the directory. Either one of

the two have to be correct, if you have set them correctly that is, but Stata will let you know. If you are collaborating with others, for example on Dropbox or some network drive, then you can keep adding as many `cap cd` options as you want.

Also note the use of `//` at the end of the line for commenting on the code.

TIP: *use `//` to comment on codes where ever required!*

In short, comment as much as you can. Here `//` are particularly useful since they allow you to comment after a code is written. For example:

```
*** here is my first regression
reg y x1 x2 x3    // some comment here
```

whereas `*` are basically used for marking out code but can also be used to leave notes on new lines. But more in commenting in the next section.

Coming back to relative paths, if the above code runs correctly, we should be in the main "myproject" directory. For this point onward the subdirectories can be accessed via relative paths. For example, if you have a data file "rawdata.xls" in the raw folder. It can be read as follows:

```
import excel using ./raw/rawdata.xls, clear first
```

The more crude way of doing it would be:

```
import excel using "C:/Program
files/Dropbox/myproject/raw/rawdata.xls", clear first
```

and here you would have to correct the path if you are working on multiple computers. Plus since there are spaces in the folder names everything is enclosed in double quotes. Chances of mistakes are also high with this method.

Similarly once you are done with cleaning up the dofile, you can then use relative paths to save it in the temp folder:

```
import excel using ./raw/rawdata.xls, clear first

<some data cleaning stuff>

order var1 var2 x* y*
sort var1 var2
compress
save ./temp/rawdata.dta, replace
```

The middle part, where we generate variables, clean them, find outliers, label them, label the values etc., is sort of what most people learn first. So let's focus on the last part where we finish up after we are done with the cleaning.

First step, order the variables using `order`. This helps organize the columns. For example if you have a data which has variables that are something like *countryid, countryname, provinceid, provincename, districtid, districtname, year, month* etc, then order these in the dataset as well. You can sort variables by their alphabetical order as well:

```
order ctryid ctryname distid distname year q1* q2*
```

what is not specified in the order is moved after the order variables. One can also do different types of variable ordering. See `help order` for more details. For order, I would suggest following some logic, for example highest unit to lowest unit (usually the unique row identifier) or vice versa.

Next sort the data. This just makes it easier to view and eyeball the data in the browser window (`br`). In earlier Stata versions sorting was essential for merging but this is no longer the case. For example, if you have a panel dataset that has some unit id variable and some time variable then `sort id time`. If you have some unit order specified, then you can sort on that order as well going from highest to lowest. This step is not really

necessary but it helps keep the dataset neat. Maybe you want to sort units by time `sort id time` or time by units `sort time id` but this is case dependent.

The next command `compress` checks if the data can be stored more efficiently and makes it more efficient. For example text variables might by stored with a much larger character length then is required (a typical problem when importing data into Stata). Or numbers can be more efficiently stored as floats and doubles etc. Compress sees if it can "compress" the storage of the variables and does it automatically. This also helps save space on your hard driveIf you have a very large dataset.

In the last step, the files are saved using the relative path to the temp folder `./temp/`. Here I use the same name as the raw file. This also not necessary but if you are cleaning dozens of files, the last thing you want to do is to figure out which file was saved as which file. It is usually good if you can trace your intermediate files back to the raw data files especially if you are generating dozens of them using some loops. For one or two raw files, this is not really necessary but good to internalize these rules!

Moving in and out of sub-directories and relative paths

Let's now explore the scenario where you change the directory to a subfolder. For example, you might want to switch to the GIS directory to make life easy when dealing with shapefiles. But this applies to any subfolder. Changing to a subdirectory in Stata can be done in two ways:

```
clear
cd "C:/Program files/Dropbox/myproject/GIS"
```

or using relative paths:

```
clear
cd "C:/Program files/Dropbox/myproject/"

cd ./GIS
```

Once in the GIS folder, you can process the shapefiles etc. Let's say you want to generate a map and save it in the "graphs" folder, which is one directory up and then another directory down again. Again we can make use of relative paths here:

```
clear
cd "C:/Program files/Dropbox/myproject/"

cd ./GIS
<some shapefile processing here>

spmap <commands>
graph export ../graphs/map1.png, replace
```

Here note that to access the graphs folder we use double dots `..` which means move up one directory and then go down one directory in the graphs folder.

Similar logic can be applied to navigate the directory tree. Let's say you were in another directory inside GIS called for example "country1". The relative path to the graphs folder would be defined as:

```
clear
cd "C:/Program files/Dropbox/myproject/"

cd ./GIS/country1
<some shapefile processing here>

spmap <commands>
graph export ../../graphs/map1.png, replace
```

where we move twice up `../../` and down once `graphs/`.

Using this sort of coding is fairly confusing. Try and stick to the root directory and so all the directories are one level below. But if you do need to move around so much within folders then the next step helps:

File paths using macros (globals and locals)

Storing file paths in macros is probably the most used option when looking at replication files. Macros basically store the information either temporarily (`local`) or permanently (`global`). Locals disappear after a code instance ends while globals are permanently stored in memory until you close Stata. This makes globals a bit risky as well potentially resulting in unintended consequences. But this is not so relevant for this part. What authors usually do is that they define the key paths at the very beginning in globals:

```
clear

*** replace this with your main directory path
global projectdir "C:/Program files/Dropbox/myproject/"

global graphdir "$project/graphs"
global tabledir "$project/tables"

cd "$projectdir"
cd "$graphdir"
cd "$tabledir"
```

Notice how the main project path has the full directory path while the graph and table paths are relative paths. Therefore, here you just need to define the `projectdir` global and the rest should sort itself out. Globals are called with the dollar sign `$graphdir` .

This can be done with locals as well but this means that everything has to run in one go so stick to globals if you are swapping between files and working on different code chunks.

TIP: *Make sure global names don't clash with variable names.*

Try and keep global names as unique as possible in order to avoid scripts getting messed up.

Now one can go in which ever directory and use the global names to access the correct folders. The above example with globals would look like:

```
clear
```

```
global projectdir "C:/Program files/Dropbox/myproject/"
global graphdir "$project/graphs"
global tabledir "$project/tables"


cd "$projectdir/GIS"
<some shapefile processing here>


spmap <commands>
graph export "$graphdir/map1.png", replace
```

This can also be very useful with large projects where the master datasets are stored completely somewhere else. For example, I usually dump very large files on my hard drive and leave the scripts, data subsets, tables, and graphs on Dropbox. Globals therefore help me navigate completely separate parts of the project very efficiently. Just image if you replace the globals or the relative paths with the full paths. It will immediately become messy!

Just for the sake of completeness, if we replace globals with locals, they would look like this:

```
clear
local projectdir "C:/Program files/Dropbox/myproject/"
cd "`projectdir'"
```

but this sort of use of locals is highly unusual.

Part 5: Code styling

Here I would emphasize the use of three things: **commenting your code, splitting your code across lines** and **using tabs**. And please use all of these liberally!

Commenting

Stata allows commenting in three ways:

```
* comments on individual lines

// comments on individual lines and after some code
```

```
/*   // for marking out code blocks

*/
```

Use these as much as possible. The stars can even be used to format your dofiles to make them stand out more. For example, I sometimes use this to start a dofile with notes and comments:

```
**********************
***                  ***
*** The Stata Guide ***
***   Tutorials     ***
***                  ***
**********************
```

Or I use starts to partition the data:

```
************************
***  COVID 19 data   ***
************************

insheet using "https://covid.ourworldindata.org/data/owid-covid-
data.csv", clear

gen date2 = date(date, "YMD")
format date2 %tdDD-Mon-yy
drop date
ren date2 date

save "./data/OWID_data.dta", replace

*********************************
***  Country classifications   ***
*********************************

copy "https://github.com/asjadnaqvi/COVID19-Stata-
Tutorials/blob/master/master/country_codes.dta?raw=true"
"./data/country_codes.dta", replace
```

This just makes it look neater and easier to navigate but adding sort of visual bookmarks.

Stars can be used to mark out individual lines of code as well:

```
*spshape2dta "wien_building.shp", replace saving(wien_building)
*spshape2dta "wien_leisure.shp" , replace saving(wien_leisure)
*spshape2dta "wien_roads.shp"   , replace saving(wien_roads)
*spshape2dta "wien_water.shp"   , replace saving(wien_water)
*spshape2dta "wien_railway.shp" , replace saving(wien_railway)
```

Here we don't need to generate shapefiles every time since this is a one time process. But we still keep the code in the dofile and mark it out. Sometimes, I also specify packages that need to be installed in the dofiles and mark them out:

```
** this dofile needs these packages
*ssc install geo2xy, replace
*ssc install palettes, replace
```

The first code block can be marked out in one go as follows:

```
/*
spshape2dta "wien_building.shp", replace saving(wien_building)
spshape2dta "wien_leisure.shp" , replace saving(wien_leisure)
spshape2dta "wien_roads.shp"   , replace saving(wien_roads)
spshape2dta "wien_water.shp"   , replace saving(wien_water)
spshape2dta "wien_railway.shp" , replace saving(wien_railway)
*/
```

This is particularly useful if you have test code inside your dofiles that is then adapted to work with some loops or other large code.

The two forward slashes can be used to add notes to the dofiles:

```
* Raw file downloaded from
* https://datahelpdesk.worldbank.org/knowledgebase/articles/906519

gen region = .
```

```
replace region = 1 if group29==1 // North America
replace region = 2 if group20==1 // Latin America and Caribbean
replace region = 3 if group10==1 // EU
replace region = 4 if group26==1 // MENA
replace region = 5 if group37==1 // Sub-saharan Africa
replace region = 6 if group35==1 // South Asia
replace region = 7 if group6 ==1 // East Asia and Pacific
```

Here I used `*` to say where I got the raw file from and `//` to identify the country groups. Otherwise I would have no idea what these numbers mean!

Splitting the code across lines

Using the three forward slashes `///` one can split code across lines. This is really help with making graphs which can look fairly incomprehensible in a single line.

For example look at this single line code of a fairly basic graph:

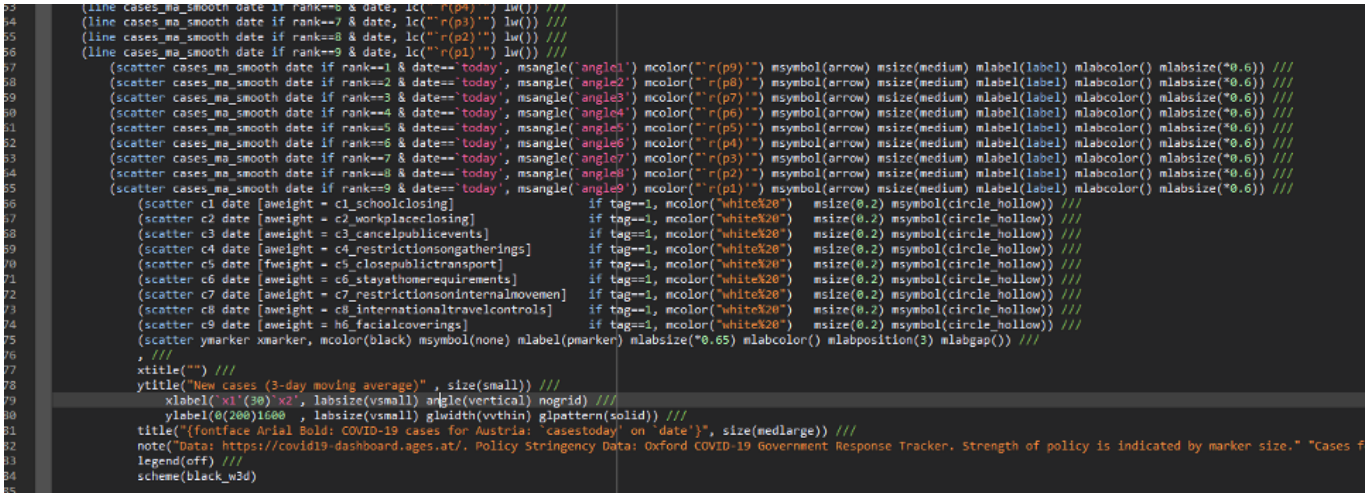
```
twoway (connected y1 x1, msize(vsmall))(connected y2 x2,
msize(vsmall)), aspect(1) xlabel(-1(0.5)1) ylabel(-1(0.5)1) xline(0)
yline(0)
```

and if we split it up across lines:

```
twoway ///
(connected y1 x1, msize(vsmall)) ///
(connected y2 x2, msize(vsmall)) ///
, xlabel(-1(0.5)1) ylabel(-1(0.5)1) ///
xline(0) yline(0) ///
aspect(1)
```

it looks fairly neater. Another way to split it up across lines is to use the delimit option (`help delimit`) but I am not a big fan of this since `///` is fairly fast and easy to use.

For example have a look at this graph code that I regularly post on Twitter:



```
(line cases_ma_smooth date if rank==6 & date, lc(`"`r(p4)'"') lw(0)) ///
(line cases_ma_smooth date if rank==7 & date, lc(`"`r(p3)'"') lw(0)) ///
(line cases_ma_smooth date if rank==8 & date, lc(`"`r(p2)'"') lw(0)) ///
(line cases_ma_smooth date if rank==9 & date, lc(`"`r(p1)'"') lw(0)) ///
   (scatter cases_ma_smooth date if rank==1 & date==`today', msangle(`angle1') mcolor(`"`r(p9)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==2 & date==`today', msangle(`angle2') mcolor(`"`r(p8)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==3 & date==`today', msangle(`angle3') mcolor(`"`r(p7)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==4 & date==`today', msangle(`angle4') mcolor(`"`r(p6)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==5 & date==`today', msangle(`angle5') mcolor(`"`r(p5)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==6 & date==`today', msangle(`angle6') mcolor(`"`r(p4)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==7 & date==`today', msangle(`angle7') mcolor(`"`r(p3)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==8 & date==`today', msangle(`angle8') mcolor(`"`r(p2)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
   (scatter cases_ma_smooth date if rank==9 & date==`today', msangle(`angle9') mcolor(`"`r(p1)'"') msymbol(arrow) msize(medium) mlabel(label) mlabcolor() mlabsize(*0.6)) ///
      (scatter c1 date [aweight = c1_schoolclosing]                if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c2 date [aweight = c2_workplaceclosing]             if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c3 date [aweight = c3_cancelpublicevents]           if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c4 date [aweight = c4_restrictionsongatherings]     if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c5 date [fweight = c5_closepublictransport]         if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c6 date [aweight = c6_stayathomerequirements]       if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c7 date [aweight = c7_restrictionsoninternalmovemen] if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c8 date [aweight = c8_internationaltravelcontrols]  if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter c9 date [aweight = h6_facialcoverings]              if tag==1, mcolor("white%20")   msize(0.2) msymbol(circle_hollow)) ///
      (scatter ymarker xmarker, mcolor(black) msymbol(none) mlabel(pmarker) mlabsize("0.65") mlabcolor() mlabposition(3) mlabgap()) ///
      , ///
      xtitle("") ///
      ytitle("New cases (3-day moving average)" , size(small)) ///
         xlabel(`x1'(30)`x2', labsize(vsmall) angle(vertical) nogrid) ///
         ylabel(0(200)1600  , labsize(vsmall) glwidth(vvthin) glpattern(solid)) ///
      title("{fontface Arial Bold: COVID-19 cases for Austria: `casestoday' on `date'}", size(medlarge)) ///
      note("Data: https://covid19-dashboard.ages.at/. Policy Stringency Data: Oxford COVID-19 Government Response Tracker. Strength of policy is indicated by marker size." "Cases f
      legend(off) ///
      scheme(black_w3d)
```

This makes use of spaces and `///` and is very easy to read. This whole code block in one line would drive anyone crazy. At least, here I can easily modify similar elements (like marker sizes or line widths) without having to search for them in one long code syntax.

Tabs for spacing

Tabs (usually the button next to Q on the keyboard) help space the code neatly. This is a highly underused formatting tool and I can't emphasize enough to use this more. The screenshot above also use tabs for automatic spacing of different code lines.

Let's space the code shown earlier using tabs:

```
twoway                                  ///
   (connected y1 x1, msize(vsmall))     ///
   (connected y2 x2, msize(vsmall))     ///
       ,                                ///
       xlabel(-1(0.5)1)                 ///
       ylabel(-1(0.5)1)                 ///
           xline(0)                     ///
           yline(0)                     ///
           aspect(1)
```

The last three codes are doing exactly the same thing but the last one way easier to read.

Commenting for sanity

Here is another example where we combine line splitting with spacing and commenting:

```
drop if ///
  _ID==14 |  ///   // Puerto Rico
  _ID==28 |  ///   // Alaska
  _ID==38 |  ///   // American Samoa
  _ID==39 |  ///   // United States Virgin Islands
  _ID==43 |  ///   // Hawaii
  _ID==45 |  ///   // Guam
  _ID==46           // North Mariana Islands
```

The IDs are also sequenced in numerical order. The same code in a single line would look like this:

```
drop if _ID==14 | _ID==28 | _ID==38 | _ID==39 | _ID==43 | _ID==45 |
_ID==46
```

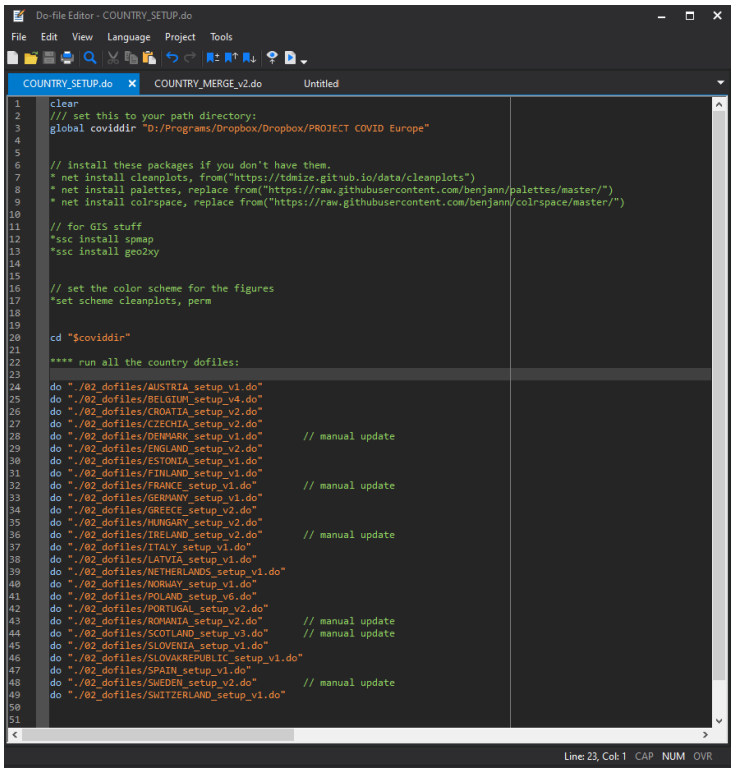
It does the job but if I look at this code months later, I will have to reverse engineer the IDs to the country names etc.

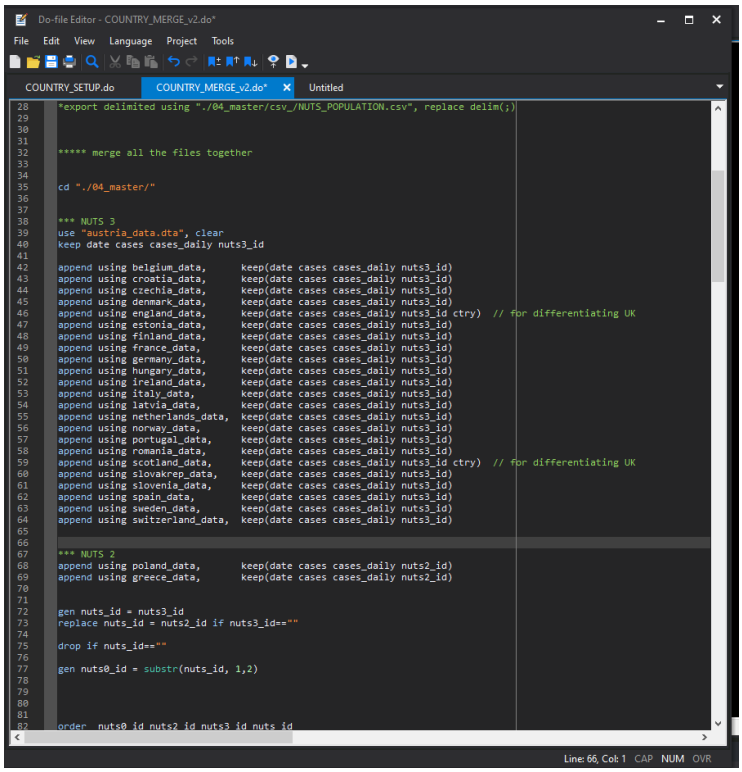
So use these three tools generously! They are more for styling code writing and they make life much easier both for yourself and for anyone reading your code.

Part 6: Linking the dofiles

As I mentioned earlier, one dofile for one task and the master dofile should run all the other dofiles and compile everything for your paper or project. The dofiles themselves are split into data management and data analysis. I will cover code syntax for these in later guides but here I want to talk a bit about how to put together individual dofiles in the master dofile.

The master dofile is essentially one very long script split into different dofiles. Here you can see a sample of my COVID-19 Tracker project done in Stata. The master file is actually two files just because the run time is extremely long and sometimes individual files give errors if there are changes in the raw files. The file on the left runs all the country files while the file on the right appends them and then goes on to prepare the master file and other files for graphs and maps. Here you can also see the use of file numbering, use of relative paths, commenting and spacing. All packages that are needed to run the files and compile the figures etc. are also there but commented out.





Here are three tips to make sure this file and individual country files run without hiccups.

TIP 1: *Each dofile should be able to run on its own*. At no point should you make it dependent on the master file or on defining some directory path etc. A lot of authors who publish their code do this. When sharing their codes, they will strip the individual dofiles of their standalone functionality. Whether this is intentional or not, it is extremely annoying to deal with and maybe even a bit off-putting especially for some researchers who are not comfortable unpacking very complex interlinked code. We will also deal with unpacking code written by other users in another guide. Even if everything depends on the master file, each individual file should have the syntax required for it to run inside the dofile, even if it is marked out.

TIP 2: Similar to the first point, sometimes authors define globals and locals inside the master file only and strip the individual dofiles of these macros making them useless on their own. For example let's assume that you have a fixed set of controls that are used in multiple regression files. Rather than copy pasting the variables, one can define a local or global and pass it on to the regressions:

```
global depvars "var1 var2 var3"
```

```
xtreg y $depvars
xtreg y $depvars, fe
```

This makes it compact and neat and one also avoids errors. The set of globals can also be used across multiple dofiles. Therefore if they are defined only once, it is the safest option since they only need to be checked once.

But if they are defined only once, for whatever reason, it is good to leave a note in the individual dofiles on where these variables are stored. Plus naming conventions matter here as well. Names like these will throw anyone off:

```
$depvars1
$depvars1_1
$depvars11
$depvars2_1
```

And if these are used in various loops which also do a bunch of various operations besides running regressions then it will be just confusion. One does not notice these things if one is in the flow and one with the code. But later one, this can become messy very fast. Again, when looking at replication files, where one has to “unpack” the code, poor naming conventions make a lot of different.

So try and name the globals in a way which makes it easy to follow for replicability and sure individual files have them duplicated even if they are commented out. Or at least leave some comments on where to find them. For example look at this code:

```
*** globals for regressions set in the master.do file

foreach x of global depvars {

xtreg `x' $indvars1, fe
xtreg `x' $indvars1 $controls1, fe
xtreg `x' $indvars1 $controls1 $controls2, fe

}
```

Fairly minimalistic for doing a large set of regressions and usually one finds this sort of a structure in dofiles but can also be very confusing if you just see this code without knowing what the globals are or have to hunt them down.

TIP 3: If you are running multiple dofiles then make sure that when an individual dofile ends executing, it always reset to the directory level that is needed to run the next dofile. In the left screenshot above, individual country dofiles sometimes navigate inside country-specific raw folders for checks and merges, but each country dofile is reset back to the root directory at the very end. This allows me to run the next dofile without errors. If at some point, some dofile gives an error the code will stop and sometimes the paths will be messed up.

In summary, it is OK to make the individual dofiles dependent on the parameters defined in the master dofile (directory, macros, packages etc.) but each dofile should be able to run on its own or it should at least contain the information that allows it to run on its.

And that is it for this guide! Hope you found this useful. If I missed something or you have suggestions what else should be covered then please reach out.

About the author

I am an economist by profession and I have been using Stata since 2003. I am currently based in Vienna, Austria where I work at the Vienna University of Economics and Business (WU) and at the International Institute for Applied Systems Analysis (IIASA). You can find my research work on ResearchGate and Google Scholar, and various repositories on GitHub. You can follow my COVID-19 related Stata visualizations on Twitter. I am also featured on the Stata COVID-19 webpage in the visualization and graphics section.

You can connect with me via Medium, Twitter, LinkedIn or simply via email: *asjadnaqvi@gmail.com*.

The Stata Guide, releases awesome new content regularly. Clap, and/or follow if you like these guides!

Stata Workflow Management