The Stata Bible 2.0

by Dawn L. Teele[1]

Welcome to what I hope is a very useful research resource for Stata programmers of all levels. The content of this tutorial has been developed while I was writing my thesis at Reed College, during two years spent doing research at Yale's Economic Growth Center, and currently while on duty as a consultant at Yale’s StatLab.

Table of Contents

[1] Doctoral Student, Department of Political Science, Yale University.
dawn.teele@yale.edu; www.dawnteele.com

Introduction: Terminology and Definitions

Stata is a statistical software package that is used by students and scholars in many fields, but is especially common in the Social Sciences, particularly Economics, Political Science, and Sociology. In many ways, Stata is a more advanced form of Excel -- it operates with data stored in rows and columns, and uses commands to transform these columns of numbers into statistics that are meaningful to the researcher. Stata can generate tables and graphs, and can be used to apply a statistical model to the data stored within.

The main difference between Stata and Excel is that the commands that the researcher writes to transform, manipulate and analyze data can be run from a command prompt and stored for future replication.2 Though some Stata users rely on the "pull-down" menus to carry out analyses, this tutorial will focus on using command prompt and "do file" programming to execute commands.

Command Prompt

Before delving deeply it should be said that a ***command*** is any action that is carried out on the dataset loaded into Stata. A command is given to Stata through the ***command prompt***, which is a wide (white) rectangle at the bottom of the Stata window (Ctrl+4 on Macs).

When someone uses Stata by typing into the command prompt, we say they are "programming on the fly". It is often useful to work on the fly in case errors come up (you miss a "." or an "=") and then transfer the good commands, the ones that worked, into a do-file.

Do-files will be discussed below, but a good rule of thumb is to "just do it" i.e. save all commands that you used to transform the data or make a calculation in a "do file", that way you will be able to reproduce your results very quickly. This is important even for seemingly simple homework assignments – it always sucks to think you have the correct answers, check it against someone else's' and have to do the whole thing over again.

Help, Shortcuts, Plug-ins

Help: Stata has good help files built in. From the **command prompt** simply type

2 True, Excel can do this via its Visual Basic language, but few people seem to use VBA in the social sciences.

help commandname

and the help files will come up. If you don't know the specific name of the command Stata can search its help files for a word or phrase.

Shortcuts: Two useful shortcuts: when you are working from the command prompt and you get an error message (a little red number and no output) you can use the "page up" button to get your last command (and more) back onto the command prompt. This means you don't have to re-type the command, and, that you don't have to take your hands off the keyboard to click on the command in the "review" pane. "Page up" can be approximated on most laptops or Macs by using "fn+up arrow"

Plug-ins: Though Stata is not "open-source", many programs are available that have been created by other researchers. Sometimes you will need a specific standard-error matrix, or want to generate fancy tables to be used with LaTeX, and will want to download these files.[3]

If you know the program name type:

ssc install *programname* /*this downloads from social science research council*/

or **findit** *programname* /*if you don't know the name or SSC doesn't have it*/

Make sure to download all the files associated with the program (one click should be all you need) so that Stata has the help files, etc. FYI, these programs that other people write, as well as the commands that are built-in to Stata are called "ado" files with an ".ado" extension.

Setting up Directories

A **directory** is just a folder somewhere on your computer. For example, "C:\" is the C directory, and typically contains other folders like Users (for a Mac) or Program Files (on the PC). Directories help to keep your projects organized. A **directory path** is the path, i.e. the series of folders, that you need to move to from C:\ to get to the appropriate folder. i.e. "C:\Users\dt6\My Documents" is the path to get to the folder My Documents.

[3] My favorite program for generating tables – in either .rtf, .txt, or .tex form – is "estout".

Typically Stata works from “C:\data” meaning that if you save a file (see below for saving) without specifying where it should go it will go there. You can find out what directory you are in either by looking at the bottom left hand side of the Stata window or by typing:

```
cd
```

“cd” is short for change directory, so typing just that will show you where you are, and if, say, you want to change directories from “C:\” to “C:\data” type:

```
cd data
```

If you want to go somewhere like “My Documents” use the quotation marks around the entire directory path; quotation marks allow the command cd not to be executed until it sees the end quote—a necessary thing if your folders have spaces in their titles.[4]

If you don’t know which folders are available to you in a particular directory, you can type:

```
dir
```

and Stata will print out a list of the folders in your directory. To move up one directory, e.g. to go from “data” back to “C” you can use the “..” option:

```
cd ..
```

All of this information about directories is useful when you keep data files in folders that are related to your project. Often you start a do-file by telling Stata to use a file that is somewhere (say, an original version of a dataset) and then by telling it to save the file in another directory. For this we would tell Stata to change the directory within the do-file.

Raw Data, Saving, Versions, Backwards Compatibility

The importance of always saving a raw, un-manipulated copy of your data cannot be over-emphasized. If you save a raw version you can always go back to the original if something looks funny later on down the line. Be sure to record all changes to the

[4] Some programs, like arcGIS, cannot load files from folders with spaces in their name. This means that if you are working in GIS you can store files in “C:\rawdata” but you cannot store files in “C:\raw data”.

original dataset in a do-file (more on these below) so that the process by which you make your raw data into your "using" data is replicable.

Once your raw data has been imported I recommend you do the following:

```
save violencedata_original
```

this will automatically create violencedata.dta

```
save violencedata_working
```

this will be the "working" dataset that you will use forevermore.

If you had already created a violencedata_working.dta, but you want to over-write this (for example, after you make changes you like), type:

```
save violencedata_working, replace
```

Stata will replace the original with the new dataset.

Stata puts out new versions of its software every few years or so. When you are working on a long term project it is important to note what version of Stata you were working with, as sometimes the command syntax changes over time, and sometimes old commands are no longer part of newer versions. Suppose you are using version 8.0, to prevent problems, at the top of your do file simply write:

```
version 8.0
```

This will tell Stata to act as-if it is in that version, and will allow it to read and execute (hopefully all) of your code.

Also note that whereas newer versions of Stata can read dataset files ".dta" produced on earlier versions of the software, this is not the case in the reverse. In other words, Stata is not "backwards compatible". Sometimes if you try to use, say, Stata 8 to read a Stata 11 data file you will receive an error message. This is a real inconvenience and there is no quick fix if you don't have access to the newer version of the software.

If you do have the newer version you can "saveold". This might be a good cautionary procedure if you are traveling, working with others, and so on.

```
saveold violencedata_working
```

Importing Data

If your data is not a spreadsheet (.xls, .csv, etc.) but comes from SPSS, SAS or some other program you need to convert it to Stata using StatTransfer.

If your data is an Excel file, save it as a .csv, or comma-separated-value file. To retain the column titles as variable names make sure that they contain no spaces and that there is a single row separating the titles from the data.

To import a .csv file you us the "insheet" command which brings in the spreadsheet file. It is best to specify the entire directory path from do-files, but if you are programming on the fly (see below) and know you are in the right folder, you can specify the name of the document only.

```
insheet using violenceproject, comma /*if it is a .csv*/

insheet using "C:\data\violenceproject.csv", comma
```

Memory

To even open a dataset Stata sometimes requires that you increase the memory "available" to the program. (You will see a message telling you not enough memory has been allotted if this is a problem.)

Based on the size of the file you can choose to set the memory that Stata can use to some value (in MB) greater than the file size:

set mem 500m   /* sets the memory to 500MB *?

You can set the memory permanently by using the option

set mem 500m, perm

Getting Started

Executing commands and do-files (and 4 tricks)

A ***do-file*** is a file with the extension ".do" that contains commands that can be read by Stata. You can create a new do file by clicking a button on the toolbar, or by typing "`doedit`" into the command prompt. Stata can execute highlighted lines of command or the entire file using the menus on the do editors' toolbar. Do files can also be created in a text editor of your choice – I use Notepad++.[5]

A *quick trick* for transferring the, say, 10 past commands into your do file is to have Stata print them all up (rather than copying them one by one). This can be done by:

```
#review 10   /* gives 10 commands, can put any number there */
```

A *second trick* which helps keep you organized is to use the asterisk * if you want to make a **note to yourself** inside the do file that only covers one line, multiple lines, or that appears as a note in the data itself:

```
*note: stata will ignore lines with an asterisk in front
```

/* note: this note can span multiple lines, be sure to close it! */

note: you can just put a note like this. type "notes list" to see notes stored in the data.

Third trick: sometimes you have a command, say, to make a complicated graph, that is easier to read if it spans multiple lines. For this Stata uses a **delimiter**, which like punctuation that tells Stata not to execute the command until it sees the punctuation.

```
#delimit ;      /* this makes the semi colon the delimiter */
command 1 line 1
     command 1 line 2 ;
# delimit cr    /* this turns off the delimiter – important! */
```

Unless you plan to put the semi-colon at the end of every command, it is better to only turn it on when you need it. Don't forget to turn it off though because otherwise Stata will keep scrolling through the do-file without executing any of the commands that come after.

[5] Notepad++ is a free program that can integrate syntax highlighting, and which can be set-up to execute commands directly in Stata, just like the do-editor. A thorough discussion of text editors can be found here:
http://fmwww.bc.edu/repec/bocode/t/textEditors.html
For information on how to integrate Notepad++ with Stata, check out:
http://econ-server.umd.edu/~kranker/code/notepad_to_stata.php

Last trick: Do files automatically break when an error is found, but sometimes we have to tell Stata to ignore an error. For example, when part of our do file would ask a variable to be dropped (so that it can be written over if it exists), we first type:

capture drop variablename

here, **capture** allows Sata to drop the variable if it exists, or ignore the command if there is an error.

Log File

Stata can produce a **log** of everything that was done in a session (i.e. in a given time period that you are using the program. This can be useful for analyzing the results of commands, such as regressions, that take up a lot of screen space.

For example, if you run 6 regressions and want to look at the results together, you might not be able to do this in Stata's viewer which only retains so many lines of information. The log file is thus useful because you can look at what happened. Sometimes Professors ask you to hand the log files in.

You can start a log from the command prompt or a do file by typing (and then executing) the following:

log using *logname*, replace

In the command above give the log a name, and use the "replace" option if you are replacing an old log of the same name (i.e. you are re-running a do file). Replacing is a good idea so your folders don't get too hectic with tons of files. I like to name the logs "l.dofilename" so I know which do file created that log.

Make sure that you end a do file by closing the log, that way commands from other do-files won't end up in the wrong log.

log close

Command Syntax

In order to execute commands, all Stata commands must be written in proper ***syntax*.** Syntax is the pattern of formation of sentences or phrases in a language. Stata syntax is the structure of commands that the program can/will execute. In general, stata syntax is structured as follows:

```
command arguments, options
```

The “command” is the name of the command you want to execute. The “arguments” are things like variable names and restrictions on the range of variable that you want the command to be executed over. The “options” are additional pieces of information that you pass to Stata. Options always come after a “,” and in general there is only one comma per command.[6] If you do not want to invoke any options, leave the comma out.

As an example of reading Stata syntax, type “`help generate`” into the command prompt. The first term “`help`” is general and will bring up Stata’s help files in a separate window. Adding “generate” tells Stata to bring up the help file on the command “`generate`”, which is a command used to generate variables. The file looks as follows:

```
Syntax

Create new variable

generate  [type]  newvar[:lblname]  =exp
[if] [in]
```

Note first that the first “g” of “generate” is underlined. This means that Stata accepts the letter “g” as a substitute for the whole word “generate” (or another variation on the root “gen” “gener”). When programming on the fly, rather than typing the whole word (which takes time and subject to typos) you can use the shorter “g” or “gen” for convenience. All help-files start in the same way by showing the command name and underlining abbreviations when they exist.

The second argument in the syntax help is [type]. The brackets around this argument indicate that it is an optional part of your command. Because “type” is blue, you can click on it in Stata to more information about the concept. Here the word “type” refers to the type of storage that will be used for the variable you are creating. For example, if you are creating an alpha-numeric variable like CountryName which will not be used to tabulate tables etc., you would indicate that the variable is a ***string*** by “`generate`

[6] Sometimes an option uses a parenthetical (xx, xx) which might have a comma, but in general there will be at most one comma.

`str## arguments`". The ## should be substituted by a number (like 7) to show how many characters you want each cell in that variable to hold. For example if your largest country name is "America" you could type "str7". *In general, there is no need to worry about entering a variable type.* Stata is good at detecting this on its own.

The third element of the syntax help is newvar[:lblname]. "newvar" holds the place of what will be your new variable name, for example "CountryName". Note that variable names must not contain spaces, and they are case specific. If there are no other variables with "Country" as a root, you can name the variable "CountryName" but use the abbreviation "Country" when calling up information about that variable.

The option after "newvar" indicates that followed by a ":' a **label** can be attached to the variable. Information about the labels is stored internally, and can be attached, via the ":lblname" option to the variable through the generate command. Labels will be discussed in more depth below.

The fourth element of the example command is "`=exp`" "exp" stands for "expression". Both the equals sign and an expression are needed to complete the command. For example, suppose you want to generate a column of 1s.

```
generate ones=1
```

This command will generate a column of 1s as long as the number of rows in the dataset.

_n (a brief aside)

When you import your data into Stata it will be stored in something that looks like an excel spreadsheet. Each row of the dataset has a number assigned to it that is stored by an internal variable called "_n". If your data has 40 rows, _n will go from 1 to 40. Note that _n does not "uniquely identify" a row – if you do something to your data like "sort" on a particular variable (i.e. **sort** age will sort your data by the variable age), _n==1 will change from whatever happened to be in row 1 at the time to the row that has the person with the smallest value of the variable age.

_n has many important uses which will become apparent below: you can do things like condition on an observation, say, in rows 1-10, or print tables with the data from rows 1-10.

Remember that _n is not a permanent characteristic of a row but rather associated with the way that the data is currently sorted. !

if/in

The options after "=exp" are [if] [in] which are articles that restrict the variable generation in certain ways. For example, suppose you want to make a column of 1s only in front of records from America:

```
generate ones=1 if CountryName=="America"
```

Note that the word America, because it is a **string variable** must appear in quotation marks for it to be read. If, on the other hand, you want to generate a column of 1s in the first 20 cells:

```
generate ones=1 if _n==1/20
```

This tells Stata to call on the internal row counter "_n" and to put a 1 in the new variable when the row is between 1 and 20. "_n" is a very important tool for programming. In this example it acts like a variable of its own. Be warned that the values associated with the row (for example if _n==1 we are talking about the first row) change based on the data sorting. So if you first have your data sorted by country name, the first 20 rows might deal with America. But if you then sort on GDP, the first 20 rows might be Sierra Leone.

The final option in our generate example is "in". The "in" article refers to the internal row numbers, and is often a shortcut. The same command above can be executed by typing:

```
generate ones=1 in 1/20
```

If and in are some of the most useful modifiers. The general structure used, throughout the entire program is the following:

```
command arguments if variable==someval

command arguments if variable=="somephrase"
```

&/or

Besides “if” and “in”, sometimes we want to condition our commands on two things being true, or on one thing or another being true. For example, say we want to put a 1 in front of each observation that comes from America and where the variable “republican” takes on the value 1:

```
  generate            ones=1            if
CountryName==”America” & republican==1
```

Here the ampersand “&” serves as the “and” Note that the command should only be on one line (in Stata this isn’t a problem). If you want to put a one in front of ever case that is American or Spanish:

```
  generate ones if CountryName==”America”
| CountryName==”Spain”
```

Here the line above the backslash “|” serves as the “or” indicator. Now that we have reviewed the syntax (and learned a little about generating variables, it is time to work a little more with the data itself.

display

The command “display” can be used to either make a mathematical calculation, tell you something about a variable, or show you the result from some Stata output.

display 4*10 /* will perform the calculation */

display “hello, world” /* will print out “hello, world” on screen */

Display can be useful if you want to see a number generated by some Stata command that is stored internally (and not necessarily printed out by Stata on the screen).

Variable storage

Stata stores variables (i.e. vectors or columns of a spreadsheet) differently. It has a way of storing data that is non-numeric (string variables) like the names of countries in the dataset that will not allow numeric operations to be carried out on it.

Sometimes Stata will store data incorrectly, for example, if your original file has dollar or percent signs $, % in the cells, Stata will not immediately recognize the cells to contain numbers.

Some Programming Tips

Programs are saved as ".ado" files which is short for "automatic do file". These are different from do files because once they are written and saved they can be executed from within any dataset or do-file. If a program is saved as *name*.ado, then typing *name* automatically runs that program.[7]

When an ado file is changed, type **discard** on the command prompt so that Stata refreshes itself and ignores any previously stored copies of the file, e.g. so that Stata discards the old copy

Programs can be created right in the command lines, first by telling stata the name of the new program *newprog* and then by typing subsequent steps. Make sure to end the program with the last entry **end**.

```
program hello

display "Hello, world"

end

hello /* runs your new program */

program drop hello /* erases the program */

which hello   /* will tell you if a program
name is already taken*/
```

Problem: there is no way to edit or store programs (e.g. to save the exact language of the program) it is better to create and run programs through do files, that way the do file can be edited without having to re-type each line of the program.

[7] Programs are useful if you do the same thing many many times. For example, if every time you get a new dataset you tabulate all the variables one by one, or you summarize and describe and then print histograms, you can write a program that does these things and then tell just execute the program instead of typing all the commands (or, as is more likely, instead of copying and pasting from an old version of the do file.

Programs created in do files are helpful for doing something repetitive and making sure that it is repetitive – macros can be embedded in the do file so that the repetitive action can be simply repeated on different variables.

set more off

If your do file or program spits out a very long table (i.e. one that is longer than the Stata screen you might have to click the space bar to get Stata to keep running the command.

You can tell a do-file to keep executing the file (so you don't have to press space and can go back to reading the NYTimes) by typing:

```
    set more off
```

quietly, noisily

If you don't want Stata to display all of the output that a do file or program creates you can tell it to do something "quietly". This is useful because it sometimes saves computation time by not having to display everything.

```
    quietly regress y x1 x2   /*will not show you regression output
on screen */
```

But sometimes you want to see absolutely everything that Stata is doing

```
    noisly tabulate x1 x2
```

Seeing all the noise can be helpful if you are trying to learn how to write more sophisticated programs yourself.

trace (debugging)

Sometimes you have written a long command (in a do file or program) and there is some problem that is causing it not to execute. Using the "trace" command will allow Stata to show you all the noisy things that are going on internally.

Some typical problems in programs, for-loops, etc. are forgetting a parenthesis, or forgetting to turn the delimiter off. If you set trace on to look at a particular part of the code make sure to turn it off after that code is done because it takes a lot of time and prints a lot of output on the screen.

```
set trace on

set trace off
```

Working with Data

Organizing Data

To test and see whether a dataset is clean or if the dictionary that comes with it is true, use the **assert** command. Assert is useful to verify certain assumptions, especially when new data has been added. It also makes sure that things that should be true are (for example, that the only options for sex are “m” or “f”).

To prove that a string variable is always assigned use:

`assert varname != " "` -> i.e. it is not the case that varname is empty.

for int (i.e. numeric variables)

```
assert varname < . &  varname > 0
```

for binomial or categorical variables

```
assert varname==1 | varname==0

assert packs==0 if !smoker
```

If an assertion is false, Stata will tell you how many times the contradiction occurs. To see the individual occurrences:

`list varlist if !(varname < . & varname > 0)` → this will show all the occurrences, and give the observation number, where the assertion is not true, denoted by !

if the mistake was a simple one, you can change the dataset

`replace varname=1 in 68` → here 68 is the observation number given by the **list** output. If the list gives you a range, say that obs 68 through 78 are incorrect, use **in** 68/78

With some datasets it is also necessary to determine that there are no duplicate observations so that time series and panel regression commands can be used. For this, the **duplicates** command is useful to guarantee that each participant in a survey only has one observation for each year, for example.

```
duplicates tag var1 var2, generate(tagname)
```

→will identify duplicates e.g. same house, same year and generate a new variable, *tagname*, equal to 1 for each observation that has a duplicate. Then you may

`view if tagname == 1` and determine if the rows are true duplicates or if some error was made.

Arranging Dataset

You can order the variables in your variable viewer (sometimes convenient if you have a ton of variables)

order var1 var2….

order a* b* c* -> here the * indicates that you order the variables starting with those that being with an "a". Stata will order alphabetically.

Another useful command is the "by sort" option, which allows you do execute a command over categories (the "by" part) that you have sorted (which is necessary for Stata to make the aggregation you are asking for,

`bys group: command arguments` /* bys is short for "by sort" ; if you want stata to output information based on groups, they have to be sorted, thus bys sorts for you and takes out an extra line of code */

Looking at Data

```
browse varlist
```

`edit varlist` -> I recommend never editing data in the data editor but making all changes in the do-file.

```
describe varlist

summarize varlist
```

`summarize` `varname,` **`details`** → gives the 5th and 95th percentiles, stored as **r(p5)** and **r(p95)**

`return list` -> tells us what values Stata stores after a command. For example, the "summarize varname, details" stores things like mean, min, percentiles, which can be invoked by the **`r(mean), r(min)`** options. For example, after the summarize command you might want to make a histogram for lower than 75th percentile values.

```
hist var if var<=r(p75)
```

inspect var (also **insp** var)

→ inspect reports negatives, zeroes and a small historgram

list *varlist*

codebook *varlist*

→ gives summary stats by variable name, will do all variables if not specified

tabulate *department*

→ by *department* gives the frequency (number of obs) as well as the percent of the sample that is in each *department*

tab hideg, sum(yrseduc_total)

→ gives tabulation by hideg categories, but finds means based on the variable inside the sum() here, years of education. So this tells us what the average years of education are for different levels of degree.

table majorcodeOUR, c(n collegemajor mean b1 mean b2 mean compare) format(%9.2f)

→ gives a table by majorcodeOUR categories and has the columns of the tables representing the number (n) of collegemajor, mean of b1 mean of b2 etc.

count if *department*= = “Economics” & *sex*= = 1

count if mpg==22 | mpg==25 | mpg==34 | mpg==45

can be written as:

count if inlist(mpg,22,25,34,45)

Another function is inrange() eg

count if inrange(mpg, 23,34)

Labels

There are three types of labels that are helpful for beginning programmers: variable labels, value labels, and dataset labels.

A ***variable label*** operates like a caption on your variable. Sometimes your variable will be eccentrically named, e.g. “adch15”. What the name stands for might elude you when returning to the project, but a variable label can tell you that it stands for “adult children older than 15”. To attach this caption to your variable:

```
label variable adch15 “adult children >15”
```

Value labels are very useful if your variables contain numbers that correspond to certain categories, but which by themselves are not very informative. When tabulating, summarizing or describing the variable, it might be helpful to have a label on the underlying numbers. To see labels instead of numbers you first have to define a value label, and then attach the label to the relevant variables. For example, suppose that a survey response “0” means “Yes” and a response “1” means no. We will create a label called “yesno” and then apply that label to the variables that conform to this pattern:

```
label define yesno 0 “Yes” 1 “No”

label values republican Christian yesno
```

These two lines of command define the label “yesno” and then attach that label to the values of the variables “republican” and “Christian”. Now, when you open your data browser instead of seeing columns of 1 and 0 for those variables, you will see yes and no in blue. The blue indicates that the text you are seeing is a label. If you want to browse without the labels type

```
browse republican Christian, nolabel
```

You can also apply a ***dataset label*** internally, which, when you open the file, will appear at the top:

```
label data “compiled 12.2.09”
```

I integrate this type of command into each of my do files at the beginning, so that if I have questions about different versions of a file, I can open it and know when it was used the last time.

Variable labels and value labels will appear in when you “describe” a variable, and you should also be able to see the label, and the type, in the Variables window in the main Stata interface. Variable labels can be used in place of names when exporting regression commands using the “estout” command (more on this later). In general, a well-labeled dataset is a good idea for long term projects or those you work on with others.