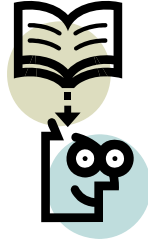


The Very Basics of Stata (Windows)



**Statistical Computing Group @ Research Data Services
University of Pennsylvania
Last modified: 2/23/2010**

This online workshop is an introductory note for Stata beginners. Our goal is to give you a broad overview of Stata and help lower your “start-up cost” of learning. Our assumption is:

- (1) You want to start learning Stata and want to know how to do simple data management and statistical tasks in this statistical software, or
- (2) You have tried Stata a bit before, but would like to get back into it and/or want to refresh your memories about oft-used basic procedures.

For such Stata-newbie people, this workshop is a good place to start.

[Stata](#) is a statistical package rapidly becoming popular in academic fields. Its strengths include a simple and consistent command structure and a wide variety of statistical procedures including complex sampling and panel/time-series analyses. Another good thing about Stata is a very helpful user community that keeps offering user developed procedures. It has a good balance of user-friendliness and analytic sophistication.

* For a comparison of different statistical packages, see [John Marcotte’s presentation](#).

If you come to Stata from SPSS looking for a more comprehensive statistical analytic tool, you will need a bit of change in your mindset. Stata does allow for a point-and-click approach, but you should definitely learn to use commands and programming to make full use of Stata’s strengths. This workshop will help you start using Stata comfortably and further learn Stata yourself according to your future research needs.

[Here](#) is a folder that contains the data sets we use in this workshop. In the examples below, I assume you place those files at “C:\”.

Contents

1. Getting Started	2
2. How To Read In Data	16
3. How To Modify Data.....	25
4. How To Get Descriptive Statistics.....	43
5. Analysis Example	56

1. Getting Started

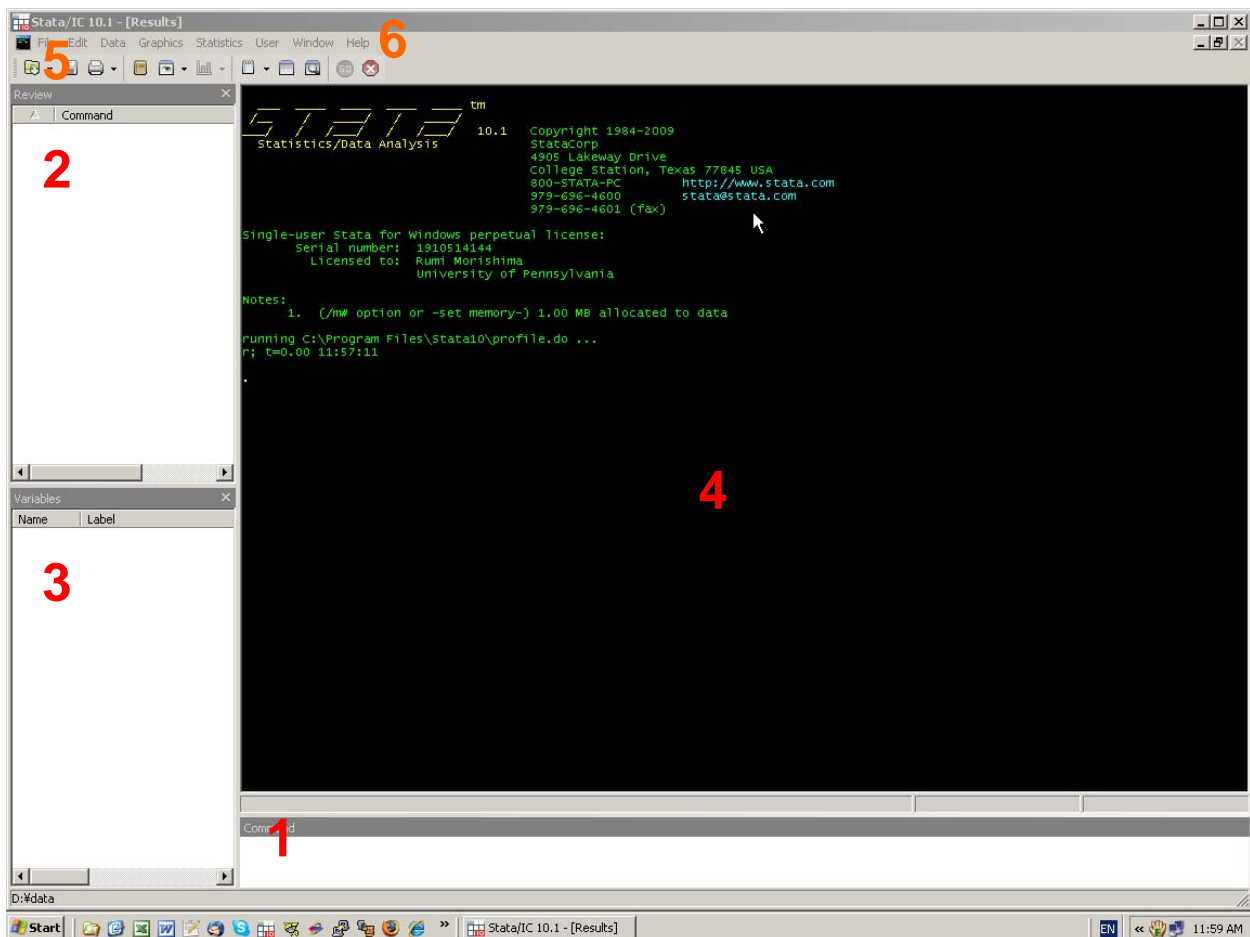
Stata has four different types: Stata/MP, Stata/SE, Stata/IC, and Small Stata. StataCorp's official website provides [details](#) how those types differ (you can get info [here](#) as well). Penn students and faculty can purchase a license through GradPlan. For more information, visit [Penn Business Services, Office of Software Licensing](#). SAS students, faculty, and staff also have access to Stata/MP and Stata/SE on our Linux server. For details, see [our notes on running Stata on Linux](#). Here, though the latest version is ver.11, we for now assume you are using StataIC version 10 (the standard version).

Quick Interface Tour

Let me first launch Stata and take you on a quick user interface tour.

Start > Programs > Stata 10 > StataIC 10

Here are your basic Stata windows below.



You have four main Stata windows now (in red) and useful tools at the top to perform basic tasks (in orange). Let's first take a look at the windows one by one.

1. **Command Window:** Here in this window, you can type Stata commands and run data management and analysis in an interactive manner. The use of this window ultimately should be limited to exploratory purposes, though. We will get to this soon below.
2. **Review Window:** Past commands you ran appear here. If you want to run the same commands you just ran, it can save you retyping in the command window.
3. **Variable Window:** Your variables in the data are listed here.
4. **Results Window:** Your computation results (and some ascii graphics) and messages from Stata (e.g., command used, warnings, error messages, etc) are displayed here. When you launch Stata, the first message you will get from Stata in this window is:

```
Notes:
1. (/m# option or -set memory-) 1.00 MB allocated to data
```

Just keep in mind you got this message from Stata. We will talk about what this means [later](#).

Let's use a quick example to get some good feel about how these windows work. Now, let's have one local rule: throughout this workshop, I will indicate Stata commands by placing two -'s (dash's) before and after them, like -use-, so you can instantly see they are commands.

Stata has various example datasets installed with it. So, for instance, let's load onto Stata an example data about automobiles. In the Command window, type:

```
Command
sysuse auto
```

Then hit the Enter key. -sysuse- is a command to load an example dataset onto Stata's memory, and this command is followed by a data file name. "auto" is the name of the data file we use here.

Your Stata is *web-aware*. Then instead of using the "auto" data installed in your computer, you can load it from online as well. -use- is to use Stata-format dataset (not just StataCorp's example files but *your* Stata data files, so you will see this command quite often hereafter, whenever you want to load your Stata data).

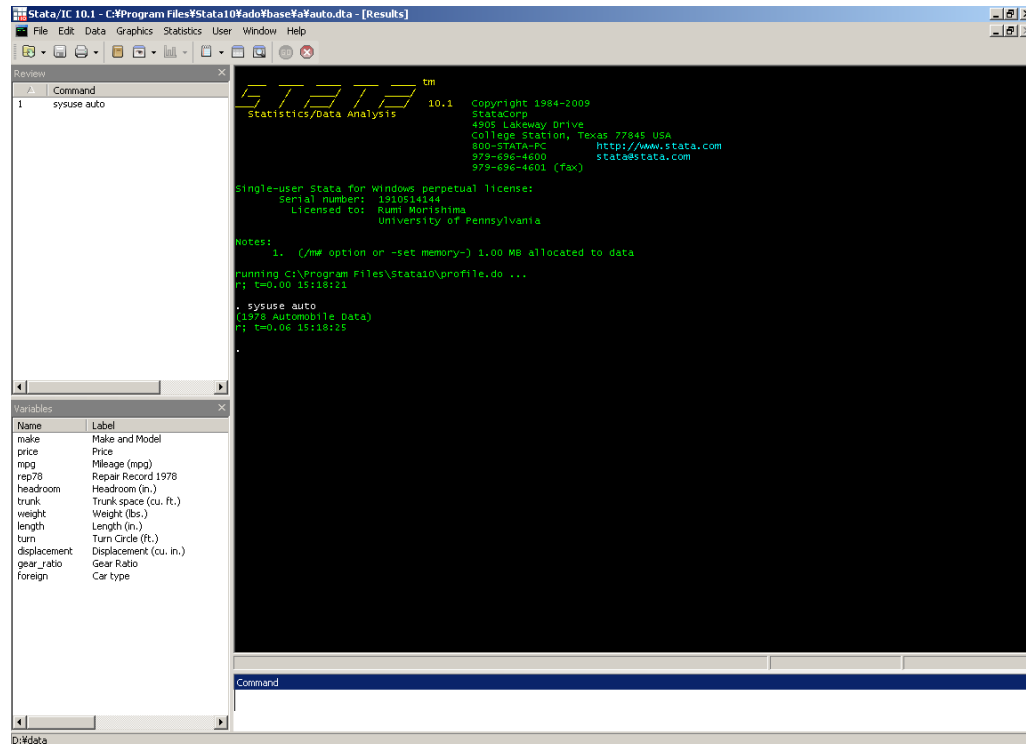
```
Command
use http://www.stata-press.com/data/r10/auto
```

Or alternatively, just simply type in the following and hit Enter.

```
Command
webuse auto
```

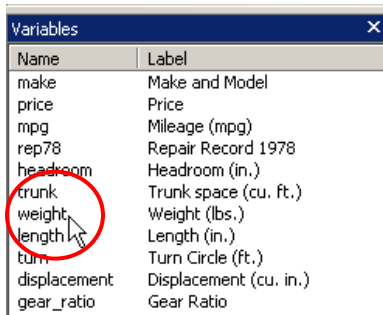
... instead of typing in a long URL. `-webuse-` is to use dataset from StataCorp's website. It works the same way.

Whichever command you typed in, what you do next to execute your command is to hit Enter. I executed `-sysuse auto-`, and here is what I get (you will get the same if you use either one of the other two command lines).

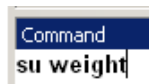


As you can see, the Review window lists the command line you just executed. And the variable names in the "auto" data file are all listed in the Variables window. And in the Results window, you see the command just run and a data label attached "auto""(1978 Automobile Data)."

Let's play with this data a little. First, let's get basic descriptive statistics for the variable *weight*. `-summarize-` is a command for this task. Notice that I underlined the `su` in `-summarize-`. Stata allows abbreviations for commands/command options, so with the underline I indicate the minimum allowed for the command. So, either `-summarize-` or `-su-` (or longer) in the Command window works the same way. Then, you could type in the variable name *weight* yourself right after. Or alternatively, you could use the Variables window. Place the pointer on the variable...



... and click on it. Then you see Stata inserts for you the variable *weight* after `-su-` in the Command window.

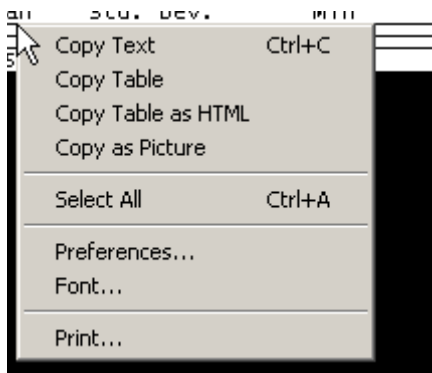


Hit Enter, and see what you have in the Review and Results windows now. First, let's see the result. You should have this output.

```
. su weight
```

Variable	Obs	Mean	Std. Dev.	Min	Max
weight	74	3019.459	777.1936	1760	4840

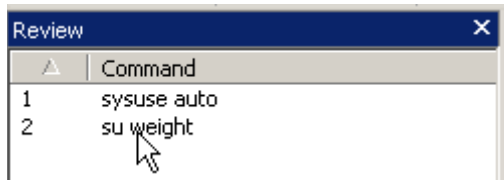
You can print or copy/paste your selected output. First, printing works just like any other applications like word processors or spreadsheet programs. Just highlight the part you want to print out, then right-click on it to select "Print..." Then in the Print dialogue box, select the radio button "Selection" under the "Page Range" heading and Click Print. As for copying/pasting, you have four options. Let's highlight the above output and right-click on it.



The first one "Copy Text" is the standard text copy you probably are familiar with. The second one "Copy Table" allows you to copy it as a table. Try copying the selected output with this option and pasting it onto a spreadsheet (e.g., Excel) and see what you get. Try comparing it with your paste output on an Excel sheet when you choose the "Copy Text" option. At any rate, these two are probably the most frequently used copy options.

The other two can also be useful, depending on your purpose. Feel free to play with them. Paste the selected output on a word processor document (e.g, MSWord) with different options and see what you get.

Now, let's turn our attention to the Review window now. You can see the command line we just submitted (-su weight-) is now listed there. Now, suppose you need to get a summary statistics for *weight* again. You could, of course, type the same command line again in the Command window, but in fact you don't have to. Place the pointer on "2 su weight" in the Review window and click on it.



Then you see this command is pasted in the Command window. Of course you can modify the pasted command. This feature is particularly useful if past commands you want to run again are long.

Now that you are getting used to those main windows, let's take a look at some useful tools at the top of the entire window.

5. **Tool Bar:** Some oft-used commands and features are also available with quick button access. Which include both general tasks (open files, save, print) and Stata-specific (log, Viewer, Do-file, Data Editor/Browser, etc). We will get to them later.
6. **Pull-down Menu:** Just standard Windows pull-down menus. Like the Tool bar, some are general, such as open/save files, cut/paste/copy, etc, whereas others are Stata-specific, such as those under "Run."

As mentioned, many of the tasks those tools/menus do can be done through commands. For example, suppose you want to see the "auto" data in a spreadsheet. You can click on the "Data Browser" button,



Or, alternatively just type `-browse-` in the command window and execute it. The data browser should open (feel free to try). As you are getting used to Stata, you will probably prefer to run commands. And that's what we would recommend too.

So far, we have been through Stata's user interface and played with the Stata main windows. We tried some features, ran commands interactively from the Command window, and got some output in the Result window. Now, as you may have noticed already, once you close this Stata session (i.e., close Stata), everything you did will be gone. The commands you entered in the Command window and the results you got in the Result window will all be gone, and you would

have to do everything from scratch. Of course, you wouldn't want that to happen. You want to save your work. You want to maintain reproducibility and documentation of your project. Stata has two types of files for those purposes.

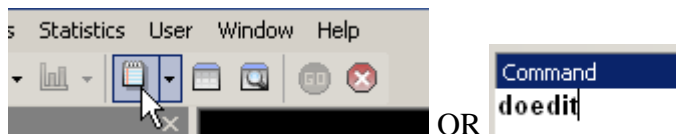
- .do file
- log file

Let's take a look at these one by one.

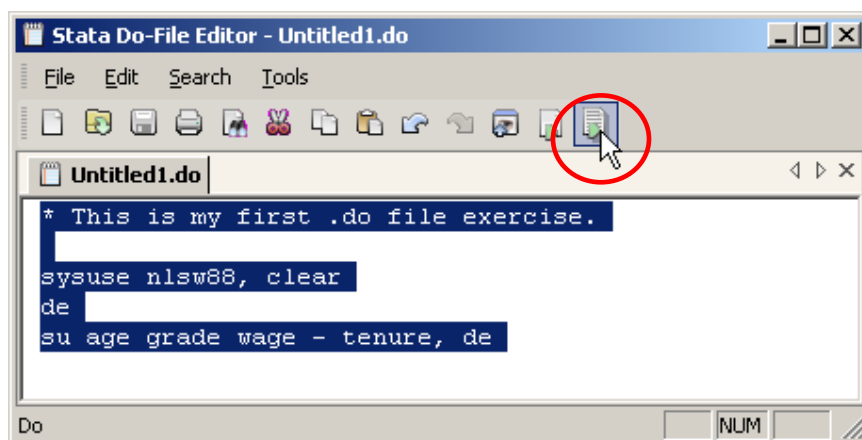
.do file

In the real-world data work and analysis, you as a Stata user will need to run many commands in Stata, and you certainly want to document all the work you did and to keep your ability to reproduce the same work again and again without typing the same commands again and again and running them one by one, repeatedly. Stata's do-file is a text file where you can write/save your commands AND have Stata read and execute them in sequence from there.

Let's open the do-file editor. If you want to take a point-and-click approach, click the Do-file Editor button in the Tool bar. The same task can be done more quickly by typing and executing the command `-doedit-` in the Command line.



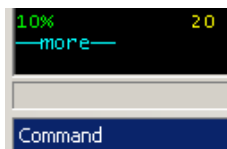
And you have a new .do file ("Untitled1.do") open for your work. As you can see, the Stata Do-File Editor also has standard pull-down menus and a tool bar. Now, let's type in the following command lines and highlight them.



We will issue all the highlighted lines to Stata. Again, you have two approaches. For a point-and-click approach, click the "Do Selected Lines" button in the Tool bar of Stata Do-File Editor (circled in red above). The quicker way, though, is to press the "Ctrl + D" keys.


See your output in the Result window (output omitted here for the sake of space, but take a look at your Result window). As you can see, Stata read all the lines you submitted to it and executed them all. The first line is just your [text comment](#), then in the next line we loaded a new Stata example data file called “nlsw88.” We added the option [, clear] to the command -sysuse-, in order to clear [the current data from Stata’s memory](#) before loading “nlsw88”. Then, we ran the command -describe-. This command produces a summary of the dataset in memory, including the number of observations and variables, the file size, the last time it was saved, variable names and types, formats, and value and variable labels. And in the final line, we ran -summarize- for six interval variables. Notice a shortcut is used to list those variables. *wage* through *tenure* are consecutive variables in the data file, so we can use a dash “-” to indicate that, instead of listing all the variables. Finally, unlike the previous example we used the [, detail] option for the command -summarize-. With this option, you get more detailed descriptive statistics, as you can see in your output.

You may be seeing -more- at the bottom of the Result window.



-more- tells you Stata has more to show you below. When you see it, there are a couple of things you can do.

- (1) To see the next line, press Enter or *l*.
- (2) To stop Stata from showing more output, press *q*.
- (3) To see the next entire screen, press space bar or any other key (any but Enter, *l*, or *q*).


Getting back to our .do file, we can save it so that we can see/continue our work and run the same analysis later, in the Windows’ standard manner. You can use the Save button  in the Tool bar, or go File > Save (or Save As..., depending) from the pull-down menu, or simply use “Ctrl + S” keys. Specify the directory you want to save the .do file (most certainly in your project folder; here let’s select the location we downloaded and saved the files at the beginning, i.e., c:/), and name the file (let’s here save it with “*mmdyy_myfirstdo*”), and click Save.

But what about saving the output we had Stata produce? That’s where your log file comes in.

Log file

A log file is a file to record what you type and what Stata outputs in response in your Stata session (i.e., your comments, data management process, analysis results, and the commands used for those results, etc), and you can save your log file for your documentation. In Stata, however, a log does not start automatically, like in SAS or SPSS. Therefore, you must explicitly tell Stata to open and start a log file for you. Let’s open a log file, give it the same name as our .do file’s

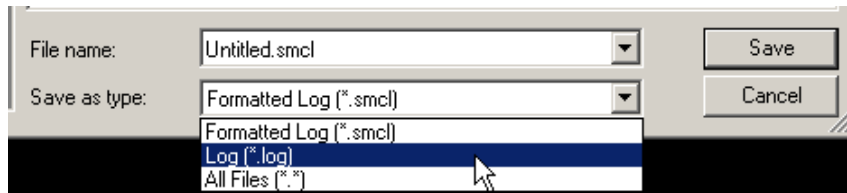
(i.e., “*mmddy_myfirstdo*”) so that you can see the two files are for the same work, and save it. There are three ways to start your Stata log.

- (1) From the pull-down menus, File > Log > Begin...
- (2) In the tool bar, click the log button .
- (3) Or, let’s add a command line to modify our do file, like below. The command `-log using-` is to start a log file. Specify the directory of your log file and the file name.

```
log using "[your project directory]/061109myfirstdo"  
  
* This is my first .do file exercise.  
  
sysuse nlsw88, clear  
de  
su age grade wage - tenure, de
```

Either way starts a log file for you. Yet if you have started using a `.do` file and save your work there, I would recommend you include the log command line at the top of your `.do` file like the above example so that you can track and document your work simultaneously in your `.do` file and log file.

Whichever way you take, you may have noticed that by default Stata starts a log file in a format with the “`.smcl`” file extension. This format is called Stata Markup and Control Language. This format can be displayed by the Stata Viewer only (we will talk a bit more about the Viewer shortly). You can save your output in simple ASCII format, however, which can be displayed in whatever text editor/word processor or the Viewer. To save your output log in ASCII format, select “Log (*.log)” from the “Save as type” pull-down menu ((1) and (2)).



And as for (3), add the file extension `.log` to your file name or use `[, text]` option in your `.do` file. Either works the same way.

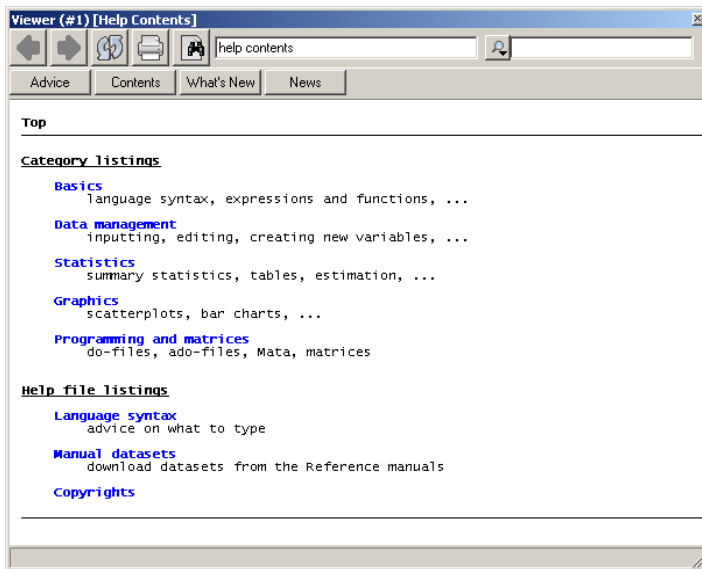
```
log using "[your project directory]/061109myfirstdo.log" , replace  
log using "[your project directory]/061109myfirstdo" , text replace
```

I also add the option `[, replace]` in the above example command lines. With this command, the log file is overwritten if it already exists. If you need to append your work to your existing log file, the option to use is `[, append]`.

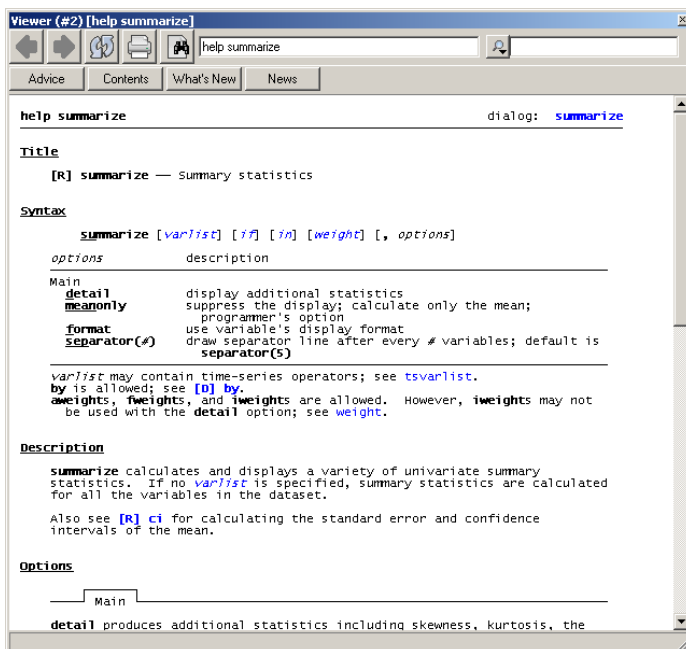
Help menu in the Viewer

We now know how to save your commands and outputs for reproducibility and documentation in Stata. In the discussion just above I mentioned the Stata Viewer. It was introduced as the way to display .smcl file. But the Viewer is probably more frequently used (especially if you choose to save your output log files in ASCII, which we would recommend) to display Stata's help menu.

In the Command window, type `-help-` and hit Enter. It brings up Help Contents in the Viewer (#1) window.



Now, how about getting help regarding specific commands? Suppose, for example, you need help with the command `-summarize-`. In such a case, type `-help summarize-` and hit Enter. You have the below Viewer (#2) window up, which gives you help about the usage of the command.



Now we are a little more familiar with Stata's user interface, there are two more things I would like you to know before we get down to actual data management and analysis examples. First, Stata's basic programming rules (a big picture thing, not specifics of each command), and secondly, how Stata deals with your working data in memory.

Programming Rules

Here are basic rules you need to know to write Stata command writing.

1. Stata's command basically takes the following structure.

```
command [varlist] [if] [in] [using filename] [weight] [, options]
```

2. You can add text comments to your Stata programs. There are the following four ways, but there are some differences in how each works.

- a. Start in a new line with an asterisk (*), place your comment. Everything after the * to the end of the current line is treated as a comment. Can be used both interactively and in .do files.
- b. Start with a slash and asterisk (/*), place your comment, and then close it with an asterisk and slash (*). Everything between these two delimiters is treated as a comment, and therefore, you can split very long lines across multiple lines. Can be used anywhere in .do files, but not interactively.

```
/*
    Analysis of nlsw88 data
    Jan 2010
*/
/* Read in data */
sysuse nlsw88, clear

su wage /* get summary statistics of wage */
```

- c. Start with two forward slashes (//). Everything after the // to the end of the current line is treated as a comment (except when // is part of http://...). Can be used anywhere in a single line in .do files, but not interactively. You will get the same output as the above by doing this:

```
*      Analysis of nlsw88 data
//     July 2009

// Read in data
sysuse nlsw88, clear

su wage // get summary statistics of wage
```

- d. Start with three forward slashes (///), then it works the same way as two forward slash in a single line. HOWEVER, when you use ///, the next line will join the current line

(see the next page for more details). Can be used in .do files, but not interactively. See what happens if you do the below.

```
/// Analysis of nlsw88 data
// July 2009

* Read in data
sysuse nlsw88, clear

su wage /// get summary statistics of wage.
su
```

For the last two lines, you have Stata telling you “**variable su not found**”. Stata interpret the last two lines as “-su- wage su” as if the second -su- is a variable name, because /// makes the next line join the first line. This is the difference between // and ///. Thus, /// is better used only as the line-join indicator; use *, /* */, or // for commenting.

3. By default, each and every Stata command ends with a carriage return.

By default, Stata commands end with a carriage return; as you saw in the previous example, there is no character or sign used as the delimiter. This is always the case when you type your commands in the interactive mode (i.e., in the Command window, you press Enter to end your command and thereby tell Stata to execute it). This also is by default the case in the batch mode, so in your .do file commands end with a line break.

What if your command line is very, very long, though? Do we have to type a line way wider than the screen till it ends then hit Enter? As shown below, there are some nice ways to get around this problem and split a long command line across multiple lines.

```
* (1) Change the end-of-line delimiter to `;'

#delimit ; /* This set up semicolon as delimiter */
sysuse nlsw88, clear ;
de ;
su <long long list of variables...>
   <long long list of variables...>, de ;
#delimit cr /* Clear the delimiter semicolon */

* (2) Comment out the line break by using /* */ delimiters

sysuse nlsw88, clear
de
su <long long list of variables...> /*
   /* <long long list of variables...>, de

(3) Use the /// (= line-join indicator)

sysuse nlsw88, clear
de
su <long long list of variables...> ///
   <long long list of variables...>, de
```

4. Stata is case-sensitive for variable names and commands.
This is very important. Unlike SAS and SPSS, Stata is case-sensitive for variable names and commands. First off, all Stata commands are in lower case (i.e., there is no -Summarize- or -SUMMARIZE-). Secondly, variable names with same letters but in different cases refer to different variables (i.e., *population* != *POPULATION*). Stata does not care about the case of filenames, however.
5. Variable names must start with a letters or an underscore (_).
Conversely, you cannot start your variable name with a number. Starting variable names with an underscore would technically be fine, but you should avoid that, because Stata's built-in variables start with an underscore and hence it makes things confusing.
6. Stata variable names can include letters, numbers, or underscores, but cannot include special characters (i.e., ~!@#%&^* <> ? = + \ () [] { }). Simply, Stata does not understand special characters in variable names.
7. Variable names must be equal to or fewer than 32 characters in length.

You are working on your data in Stata's memory!

Now, remember when you launched Stata at the beginning of this workshop you got [this message from Stata](#). That was actually a new concept for people who have primarily used SAS or SPSS before: Unlike SAS or SPSS, Stata allows you to “open” one data file per session only, or more accurately put, Stata actually *loads your data in memory*. This means that throughout a session *you are working on Stata data in memory, not the corresponding actual Stata data file itself*.

This leads to some important points to keep in mind.

(1) Clear memory before loading new data

You cannot have Stata load a data set when another one is already loaded (and hence occupies memory). If you want to load another one, you first must clear memory of the current data for that new data. To do so, use the command `-clear-` or the option `[, clear]` of the command `-use/sysuse-`. So, for example,

```

* First load auto, then load nlsw88 as clearing memory of auto.

use auto
<and suppose you do some data management work and analysis here...>

use nlsw88, clear /* use option [, clear] */

* You can do the same in the following way.

use auto
```

```
<and suppose you do some data management work and analysis here...>  
  
clear /* use command -clear- */  
  
use nlsw88
```

(2) Save your data in memory to file

Remember, however, that you are working on your data loaded in memory, not the corresponding actual data file itself; you make changes to data in memory (e.g., create new variables, change data structures, etc), not to the data file itself. This means that if you clear memory of the current data by using `-clear-` or `[, clear]`, then all the changes you made to the current data would be gone. *Thus, if you want to keep those changes made to data in memory, you need to save them to the actual file before clearing memory.* To do so, use the command `-save-`. See the below example.

```
/*  
    Load the data "auto", made some changes, and save the changes to  
    file "auto" (under the same file name).  
*/  
  
use auto, clear  
  
<Suppose you made some changes to data here, creating new vars, etc...>  
  
save, replace  
  
/*  
    Note the original "auto" file is replaced with the new one above,  
    hence altered. If you intend to keep the original file intact and  
    save changed data in another file under a different name, then  
    you need to specify new filename.  
    Suppose, for example, we save it with the filename "auto_mod",  
    then the following is what you do.  
*/  
  
use auto, clear  
  
<and suppose you made some changes to data here...>  
  
save auto_mod, replace  
  
/*  
    We add the option [, replace] just in case, but if there doesn't  
    exist a Stata data named "auto_mod" already and hence there is  
    nothing to be replaced, Stata issues a message that goes (note:  
    file auto_mod.dta not found) file auto_mod.dta saved.  
  
    Finally, if you are using Stata 10 and want to share your data  
    with Stata 9 or below users, you need to save your data in old  
    format (Stata 9 or below cannot read Stata 10 format).  
*/  
  
saveold auto_mod, replace
```

(3) Set memory size

That you have Stata load your data in memory means that memory needs to be of big enough size to accommodate your data (and work). First off, let's see what your Stata's memory is like now. In the Command window, first execute `-clear-` (if you haven't done so yet) to empty memory, then type `-memory-`. This command gives you information about your memory usage and helps you find if you have enough memory to do your work.

```
. memory
```

	bytes	
Details of set memory usage		
overhead (pointers)	0	0.00%
data	0	0.00%
data + overhead	0	0.00%
free	1,048,568	100.00%
Total allocated	1,048,568	100.00%
Other memory usage		
system overhead	1,600,258	
set matsize usage	337,600	
programs, saved results, etc.	105	
Total	1,937,963	
Grand total	2,986,531	

So, this is the current state of Stata's data area (the first big row "Details of set memory usage") when no data is loaded. Of course, with no data loaded, the usage figures all show 0's. And approximately 1 MB memory is free and available for use. This is the same as what Stata message told us when we first started the program at the beginning of this workshop. By default, in Ver.10, Stata/IC allocates 1 megabyte to Stata's data areas (note, if you are a Stata/SE user, the default memory size is 10 MB, and if you are using Small Stata, you have 300 KB). Now, let's load a data file and see the current memory usage.

```
webuse regsmpl
memory
```

```
memory
```

	bytes	
Details of set memory usage		
overhead (pointers)	114,136	10.88%
data	913,088	87.08%
data + overhead	1,027,224	97.96%
free	21,344	2.04%
Total allocated	1,048,568	100.00%
Other memory usage		
system overhead	1,600,258	
set matsize usage	5,190,400	
programs, saved results, etc.	6,443	
Total	6,797,101	
Grand total	7,845,669	

You have only 21 KB or so remaining, which is quite small. You can, however, increase the amount of memory allocated to Stata's data area (Note you *cannot* if you are using Small Stata). Suppose that you want to increase the memory area allocation to 10 MB. First clear memory of the current data, and then use the command `-set memory-`.

```
clear /* You first clear data area memory before making any change */  
set memory 10m
```

And you are good to go (feel free to run `-memory-` to see in details the increase you just made).

I would set an increased memory size at the beginning of my `.do` files. Or you can create a "profile.do" file that sets my memory size and have Stata execute it every time Stata is started. This is what I actually do, and I would recommend it. For more information, type `-help profilew-`.

2. How To Read In Data

Let's move on and talk about how to input your data in Stata.

We have two subsections here in this section.

- Reading Stata files.
- Reading external text (ASCII) files.

Stata has three commands to read text files: `-insheet-`, `-infile-`, and `-infix-`. We will learn how to use each of them in what data input situation. We will also see how to handle other external data format, such as Excel.

Reading Stata files

This is actually a quick review of what we just now learned in the example of the previous section. If you already have your data in the Stata format (`.dta`, like "auto" above), we can simply execute `-use-` as we did above (make sure the data area is cleared for your new Stata file to be read into) and you should be ready to go.

Now, however, suppose we are interested in examining what might affect each country's life expectancy, and decide to work with the Stata file "lifeexp.dta" which is located at the directory "c:\\" (or a directly you saved the files). And this is what we would get if we simply type "use lifeexp" in our `.do` file and run the line.

```
use lifeexp, clear
```



```
. use lifeexp, clear
file lifeexp.dta not found
r(601); t=0.00 12:18:21
```

Why did this error happen? r(601) is an error code and you can click on it to see what Stata thinks went wrong. This error is straightforward though; it happened because Stata does not know where to look for this “lifeexp” data, as r(601) says “Perhaps you mistyped the name, or it may be on another diskette or directory.” One way to tell Stata where to look is to tell the directory inside the -use- command. Our “lifeexp” data file is saved in “c:/”, so,

```
use "c:/lifeexp.dta", clear
```

But if you create a folder for your own research project in the future (you definitely should!), you can tell it to Stata first so it will look into it thereafter. To do so, use the command -cd-. Let’s create a project folder named “lifeexp” in “c:/” then specify it as our working directory:

```
mkdir "c:/lifeexp"
cd "c:/lifeexp"
pwd
log using lifeexp, text replace
use lifeexp, clear
```

-mkdir- (**make directory**) lets you create a new directory, and -cd- (**change directory**) lets you specify your working directory. -cd- (without any argument), or -pwd- tells you where your **present working directory** is. Once you tell Stata where your directory is, Stata sees what file you are referring to simply by file names, and everything you do in Stata about the current research project – data files, log files, etc – are saved in that directory unless you tell Stata to do otherwise.

Reading external text files

Now, how about reading ASCII (text) files? As mentioned above, we have three commands to use.

- -*insheet*-
- -*infile*-
- -*infix*-

And you use each of these commands depending on how your ASCII data file is formatted:

- (1) Delimiter-separated data
- (2) Space separated data (free format)
- (3) Column formatted data (fixed-column format)

Let’s see when to use what command to read each of these formats.

(1) How to read delimiter-separated data?

Use the `-insheet-` command to read in ASCII files created by spreadsheet programs, where the data files are “delimiter-separated.” Delimiter-separated files are raw data files in which the values in each row are literally separated with specific delimiter characters. The delimiter characters can be anything if defined properly, but the two most common formats we see are comma-delimited and tab-delimited, and many data files for social science research are available in those formats. To read this type of data files into Stata, the minimum you need to do is:

`insheet using filename`

Here, *filename* is the name of the ASCII file you want to read in. So, for example: we have a raw data file named “lifeexp_csv.dat” in the directory “c:/”. This file is comma-separated. It has seven variables: ISO-3 numeric country code (`iso3n`), region number (`region`), country name (`country`), population growth (`popgrowth`), life expectancy (`lexp`), per capita GNP (`gnppc`), and safe water access (`safewater`). Also, the data file has 68 observations. The following is a snapshot of the data (feel free to open “lifeexp_csv.dat” in whatever text editor you have).

```
8,1,Albania,1.2,72,810,76
51,1,Armenia,1.1,74,460,
40,1,Austria,.4,79,26830,
31,1,Azerbaijan,1.4,71,480,
112,1,Belarus,.3,68,2180,
56,1,Belgium,.2,78,25380,
70,1,Bosnia and Herzegovina,-.5,73,,
...
890,1,"Yugoslavia, FR (Serb./Mont.)",.5,72,,
...
```

There actually aren’t many conditions to meet for reading this type of data. I just want to draw your attention to the following three points before getting down to work.

1. Each data value is delimited by commas.
2. There is one observation per data line.
3. Notice that Yugoslavia, FR (Serb./Mont.), which include a comma, is enclosed in quotes. This is to tell Stata the comma after Yugoslavia does not serve as the delimiter but is just a part of a string value. You always must be careful about distinguishing separators as value separators from those as part of string values. Also notice that the highlighted and non-highlighted observation lines have one difference. The highlighted observations have two commas in a row (e.g., “73, ,”) indicating they have the last two variables missing (compared to, for example, Armenia which has only the last one variable missing).
4. Although we know it includes the seven variables, (`iso3n`, `region`, `country`, `popgrowth`, `lexp`, `gnppc`, and `safewater`), this “lifeexp_csv.dat” does not include variable name header.

Now let’s read this text file into Stata.

```
insheet using lifeexp_csv.dat, clear

* Data content?
de

* Let's list first 10 observations
list in 1/10
```

The `-list-` command is literally to list values of your variables, and with the `[in]` qualifier we specify the range of observations to list (1st to 10th observations from the top, in this case; type in `-help list-` for more details). You have this output.

```

Contains data
  obs:          68
  vars:          7
  size:        3,060 (99.9% of memory free)

variable name  storage  display  value  variable label
              type    format   label
v1             int     %8.0g
v2             byte    %8.0g
v3             str28   %28s
v4             float   %9.0g
v5             byte    %8.0g
v6             long    %12.0g
v7             byte    %8.0g

Sorted by:
Note: dataset has changed since last saved

```

```

      | v1  v2  v3  v4  v5  v6  v7
-----+-----
  1.  |  8   1  Albania  1.2  72  810  76
  2.  | 51   1  Armenia  1.1  74  460  .
  3.  | 40   1  Austria  .4   79 26830 .
  4.  | 31   1  Azerbaijan 1.4  71  480  .
  5.  |112   1  Belarus  .3   68 2180  .
-----+-----
  6.  | 56   1  Belgium  .2   78 25380 .
  7.  | 70   1  Bosnia and Herzegovina -.5  73  .  .
  8.  |100   1  Bulgaria -.4  71 1220  .
  9.  |191   1  Croatia  -.1  73 4620  63
 10.  |203   1  Czech Republic  0  75 5150  .

```

It looks fine. All the variable types are read in correctly. As you can see, however, the variables do not have meaningful names. This is because Stata sees whether or not the original ASCII file has variable names at the top row, and on deciding it doesn't, it gives them temporary names (i.e., `v1`, `v2`..., `vn`). You can include your variables names in the `-insheet-` command.

```

insheet iso3n region country popgrowth lexp gnppc safewater ///
        using lifeexp_csv.dat, clear

list in 1/10

```

	iso3n	region	country	popgro~h	lexp	gnppc	safewa~r
1.	8	1	Albania	1.2	72	810	76
2.	51	1	Armenia	1.1	74	460	.
3.	40	1	Austria	.4	79	26830	.
4.	31	1	Azerbaijan	1.4	71	480	.
5.	112	1	Belarus	.3	68	2180	.
6.	56	1	Belgium	.2	78	25380	.
7.	70	1	Bosnia and Herzegovina	-.5	73	.	.
8.	100	1	Bulgaria	-.4	71	1220	.
9.	191	1	Croatia	-.1	73	4620	63
10.	203	1	Czech Republic	0	75	5150	.

Aside from determining whether your ASCII file has variable names and what types of variables there are (i.e., numeric, string, etc), `-insheet-` command also determines for you whether your ASCII file is comma or tab delimited. So, you can read in the “lifeexp_tab.dat” file this way. As you can see (feel free to open “lifeexp_tab.dat” in whatever text editor you have), this file has the variable names in the first row. In that case, `insheet using filename` can most certainly handle everything because Stata sees that the first row includes variable names and treat the row as such. But by adding the `[, names]` option, you can explicitly tell Stata to treat the first row as a list of variable names and thereby ensure and speed up the processing. Feel free to give it a try.

```
* This should be enough
insheet using lifeexp_tab.dat, clear

* But this speeds it up
insheet using lifeexp_tab.dat, names clear

* List (with no varname abbreviation)
list in 1/10, ab(10)
```

One sidenote: you can see two of the variable names in the above output are abbreviated (i.e., `popgro~h` and `safewa~r`). This is because Stata automatically shortens variable names to unique abbreviations when they are longer than eight characters. To avoid it when listing, use the `[, abbreviate(#)]` option. In the above example command line, I allow for 10-character long names.

[Like I said](#), the delimiter does not have to be either comma or tab. If your data values is separated by anything else though, you need to tell Stata what it is by using the `-insheet-` command’s option `[, delimiter(“char”)]`, where `char` is a user-defined delimiter.

(2) How to read space-separated data (free format)?

To read in text files where the data are separated by spaces (free format), the `-infile-` command is the way to go. To handle free formatted data, Stata (literally) does not need the formatting information. Thus, as long as the data in your text file are separated by spaces, and:

1. do not have non-numeric characters, OR
2. do have non-numeric variables but they are just one word, OR
3. do have non-numeric variables and they are more than one word but enclosed in quotes.

... then things are pretty straightforward.

The very basic form of the `-infile-` command is pretty similar to the `-insheet-` command.

```
infile using filename
```

Now we use this command to read space separated data files, each of which corresponds to the above conditions 1 to 3.

Let's first start with the pattern 1. You have a text data file named "cancer_space_n.raw" in our workshop directory. This data has four variables, "studytime," "drug (=1)," "age," and "died (=1)," in the order that appears in the data file. The data does not have those variable names in the header, though. Here's a snapshot.

```
1 1 . 1
1 1 65 1
2 1 59 1
3
1
.
1
4 1 56 0
4 1 67 1
5 1 63 1
5 1 ? 1
...
```

And here are the command lines to run below. After the `infile` command line, I run the `-browse-` command to open the data browser and see how the data is read in.

```
* Use infile to read space separated files.

infile studytime drug age died using cancer_space_n, clear
browse
```

There are a couple of things you should note here about what we just did.

- As you can see in the data snapshot above, numeric missing values are indicated with eriods ("."). Compare this with the way missing values are indicated in delimiter-separated files.
- As the first highlighted part in the snapshot shows, the forth observation is not on a single line. That's perfectly fine; unlike the case of delimiter-separated data, it doesn't matter in the case of free format data. What matters is data values are separated by one (or more) space (blanks, tabs, or whatever new lines). Thus, the location of line breaks does not affect Stata's data reading work. Stata understands that there are four variables, and moves on to the next line to read all the four variables.
- "using cancer_space_n" is the same as "using cancer_space_n.raw" because Stata assumes the file extension is ".raw" when nothing is specified in the `-infile-` command line.
- The age variable for 8th observation in the snapshot is somehow expressed with a "?" sign. Stata, of course, does not understand it. You should be seeing this message from Stata in the Result window.

```
. infile studytime drug age died using cancer_space_n
'?' cannot be read as a number for age[8]
(48 observations read)
```

All Stata knows is that the variable “age” is numeric. Thus, Stata replaces this incomprehensible character with a numeric missing value “.” (confirm it in the data browser).

Let’s next read a pattern 2 text file, “cancer_space_c1.txt,” where the variable “died” is a string variable with no space inside (Yes/No).

```
* Use infile to read space separated files, with consec char var.
infile studytime drug age str3 died using cancer_space_c1.txt, clear
browse
```

This time the file extension is “.txt” so we let Stata know. The key point here is the highlighted part: `str3` that precedes the variable name “died.” This tells Stata that this variable is a string variable with its length = 3. See how it reads in the data browser. Notice that in Stata data browser string variables are indicated in red.

Now, let’s take a look at the pattern 3. The logic is the same as [what I already pointed out](#). That is, if your string values include the same character as the one serving as the separator, they need to be enclosed in quote to have Stata distinguish the character and the separator. So, take a look at the data file “cancer_space_c2.raw.” Those observations that died of cancer are classified by some internal codes, which are enclosed in quotes because the values include spaces.

```
1 1 . "C 651 K74"
1 1 65 "B 995 J50"
2 1 59 "U 276 S81"
3 1 . "T 951 L26"
4 1 56 No
...
```

And you now what you need to do.

```
* Use infile to read space separated files, with char var with spaces.
infile studytime drug age str11 died using cancer_space_c2, clear
browse
```

(3) How to read column-formatted data (fixed-column format)?

Now, we are getting back to the “lifeexp” data series. Take a look at the following data, “lifeexp_column.raw”.

```
8 1Albania 1.200000048 72 810 76
51 1Armenia 1.100000024 74 460
40 1Austria 0.400000006 7926830
31 1Azerbaijan 1.399999976 71 480
```

```

1121Belarus      0.300000012 68 2180
56 1Belgium     0.200000003 7825380
70 1Bosnia and Herzegovina -0.5      73
1001Bulgaria    -0.40000000671 1220
1911Croatia    -0.10000000173 4620 63
2031Czech Republic 0      75 5150
2081Denmark    0.200000003 7633040
...

```

These data lines give you some new challenges, at a glance: (1) the first to third variables look strung together without any separator; (2) the data values of the string variable “country” are not enclosed in quotes though some country names include spaces; (3) no separator, no comma, no whatsoever, to indicate missing values. In this text file, however, each data value is arranged and fixed in columns. Here is a snapshot of the data. Look how the gauge in red corresponds to the data lines.

```

1-----10-----20-----30-----40-----50-----60
8  1Albania      1.200000048 72 810 76
51 1Armenia     1.100000024 74 460
40 1Austria     0.400000006 7926830
31 1Azerbaijan 1.399999976 71 480
1121Belarus    0.300000012 68 2180
56 1Belgium     0.200000003 7825380
70 1Bosnia and Herzegovina -0.5      73
1001Bulgaria   -0.40000000671 1220
1911Croatia   -0.10000000173 4620 63
2031Czech Republic 0      75 5150
2081Denmark    0.200000003 7633040
...

```

Data files formatted this “fixed” way always come with a codebook which explains where in the data each of the variables is placed. Here is your codebook for this data.

Variable	Column number
iso3n	1-3
region	4
country	5-33
popgrowth	34-45
lexp	46-47
gnppc	48-52
safewater	53-55

Then, clearly, what you can do is to tell Stata to read the data by using this information. The commands to use here are either `-infix-` or `-infile-`. Let’s first try the `-infix-` command.

```

* Read formatted text file by -infix-.

infix iso3n 1-3 region 4 str country 5-33 popgrowth 34-45 lifeexp 46-47 ///
gnppc 48-52 safewater 53-55 using lifeexp_column, clear

```

As you can see, we specify column numbers for Stata when listing the variable names. Notice that for the “country” variable we again also need to provide the variable type information (i.e., string). See the data browser or list the data in the result window to check if the data is read as intended.

Fixed-column data can also be read by using a text file called “dictionary (.dct),” where you provide Stata with detailed formatting information. If you use the `-infix-` command with dictionary, then the content of your `.do` file would be like:

```
* Use dictionary.
infix using lifeexp_colinfx.dct, clear
```

Unlike the previous example, the highlighted part is NOT the raw file name, but the dictionary name used to read the file. The content of this dictionary file “`lifeexp_colinfx.dct`” is:

```
infix dictionary using lifeexp_column.raw {
    iso3n      1-3
    region     4
    str  country 5-33
    popgrowth 34-45
    lifeexp    46-47
    gnppc     48-52
    safewater  53-55
}
```

What this does is to create a dictionary of formatting information of the raw data file (“`lifeexp_column.raw`”) for the use of the `-infix-` command. Notice the same information as that in the previous example is provided between the `{ }` parenthesis.

The `-infile-` command, which we met just a moment ago when we learned how to read free-format data, can also read fixed-column format data as well with the help of a dictionary file.

```
* -infile- can also read fixed-column format with a dictionary
infile using lifeexp_colinfl.dct, clear
```

The content of this dictionary file for the use of the `-infile-` command is a bit different from the above one for the `-infix-` command.

```
dictionary using lifeexp_column.raw {
    _column(1) iso3n      %3f
    _column(4) region     %1f
    _column(5) str29 country %29s
    _column(34) popgrowth %12f
    _column(46) lifeexp    %2f
    _column(48) gnppc     %5f
    _column(53) safewater  %3f
}
```

`_column(#)` tells Stata to jump to that column number and read the data for the variable. We then provide variable name information, and then variable type.

Data arranged/fixed in columns has some obvious advantages over space-separated or delimiter-separated data; because you have power to specify columns where data values are located, you

have control with regard to data values you want Stata to read. For example, a single observation can be on more than one line. Also, you can also jump forward or backward within the line you now read, meaning you can for example read “country” first and then “safewater” and then “lifeexp” – you just tell Stata where the data is. So, you have more flexibility when reading fixed-column data. For more details, see `-help infile-` and `-help infix-`.

Reading Excel files

Data in Excel may be one of the most frequently encountered ones these days when you search and download from online. Stata, unfortunately, does not allow you to read Excel files directly. There are, however, a couple of ways to get around. One way if you have access to it is to use [StatTransfer](#) program. This program provides an easy, quick, reliable way to convert your data files from one file format to another (see the link to find what formats StatTransfer can handle). Another (a little cumbersome but still easy) way is to save the excel file in a delimiter-separated format and use the `-insheet-` command.

Now that we know the basics of bringing data in Stata, let’s move on how to work with your data.

3. How To Modify Data

Here are four topics we will cover in this section.

- How to create/recreate variables?
- How to subset your data?
- How to combine your Stata data sets?
- How to label and format your variables?

One thing I want you to remember here first; when we are talking about modifying/subsetting data, we are working on Stata data loaded in memory, not the Stata data file itself. Hence, unless you explicitly save changes you made with the same file name, the original data file remains intact. Conversely, if you clear the data currently in memory without saving it (with the same file name or a different file name), all the changes you made to the data in memory will be lost (Beware, SAS users!).

We will start with making new variables and in that process take a peek at some useful functions. Then we will learn how to subset data, i.e., how to conditionally select variables or observations from your data. Then, we will also learn the opposite, i.e., how to combine data sets, i.e., to add variables (merge) and to stack observations (append). And finally, we will discuss how to label values, variables, and data to help with good data management.

How to create/recreate variables?

The most frequently used Stata commands to create a new variable is `-generate-`. The basic form is:

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

So, for example, let's load the data "lifeexp.dta" in Stata's memory and create a couple of new variables by using this command.

```
use lifeexp, clear

gen datasource = "wb" // character constant
gen dataver = 2 // numeric constant
gen safewater2 = safewater // duplicate with different name
gen sqgnppc = gnppc*gnppc // multiplication, gnppc^2 also works here
```

The above codes *load* the data in memory, *create* new variables from the read-in variables, but let me stress again, these changes are not yet saved in your "lifeexp.dta" file or in any file with whatever name. They are just in Stata's memory now.

See [here](#) for other Stata arithmetic operators.

There are other tasks that the above arithmetic operators cannot handle, however. For example, you may want to test a curvilinear effect of per capita GNP specified as a logarithm, rather than a squared term, and thus want to create a logged version. Or you may want to turn all letters in the "country" variable to uppercase. Stata functions do such jobs for you.

```
* Let's use functions
gen lngnppc = log(gnppc) // log of gnppc, no zero values, so no worries //
gen country = upper(country) // upper case of country //

* And Stata complains the variable "country" already exists...
```

Two things happen here. First, by using the function `log()`, you just created a logged version of the per capita GNP variable. As you can see, the basic form of Stata functions is:

```
function_name(exp)
```

You always need parentheses for any Stata function. So you continue and try to make the string value of the *country* variable upper-case. You get the following error message, however.

```
country already defined
r(110);
```

Stata is telling you that the variable named *country* is already there (feel free to click on `r(110)` message there). There are two things you can do. First, you can create a new *country* variable with another name (say, *country_cap*) and Stata would be cool with it. By so doing, though, you would have two variables *country* and *country_cap* whose only difference is whether it's in all-caps or not. You can, instead (and this is "Secondly,"), replace the existing *country* variable with new values. The command to do so is `-replace-`. Thus,

```
replace country = upper(country)
```

Functions return missing (.) when the value of the function is undefined. Stata has so many other functions, of course, which let you handle numeric, character, and date variables. Type in `-help functions-` for more details.

Let's try one more thing here. Suppose, for example, you want to create a new character country code variable by combining *iso3n* and *country*. This means these two variables need to be concatenated. You don't see any string function in the `-help string functions-` menu to use with `-generate-`, though.

There is, actually, another Stata command that creates variables, which is called `-egen-`, which stands for "extensions for **generate**," which work with functions specifically written for `-egen-` only. `concatenate()` is one of those `egen` functions (feel free to run `-help egen-` for details).

So, we will do this:

```
* Create a new country code variable
egen ccode = concat(iso3n country), punct(" ")
```

We add the option `[, punct()]` to insert a space between *iso3n* and *country* in the new variable, because by default `concat()` adds variables end to end.

Finally, let's talk about how to create variables based on specific conditional logic. Suppose that you want to create a new variable about countries' development status. Suppose that you define developed countries as those with their per capita GNP over \$10,000 and code countries that meet the criterion as "developed." To do this, you can generate a new variable under that condition.

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

In Stata, you state action first, then condition under which the action is to take place. So, we first program as below and submit it to Stata.

```
* Create a string variable for development status.
gen devstatus = "developed" if gnppc > 10000
```

Let's list the first ten observations for the variable *country*, *gnppc*, and newly created *devstatus*.

```
list country gnppc devstatus in 1/10
```

And here's what we get.

```
. list country gnppc devstatus in 1/10
```

	country	gnppc	devstatus
1.	ALBANIA	810	
2.	ARMENIA	460	
3.	AUSTRIA	26830	developed
4.	AZERBAIJAN	480	
5.	BELARUS	2180	
6.	BELGIUM	25380	developed
7.	BOSNIA AND HERZEGOVINA	.	developed
8.	BULGARIA	1220	
9.	CROATIA	4620	
10.	CZECH REPUBLIC	5150	

Notice that the new variable “devstatus” is coded as “developed” for Bosnia and Herzegovina, despite the fact the per capita GNP variable is missing for the country. Why does this happen? It happens because in Stata, a missing value (“.”) represents the largest possible number (i.e., positive infinity) and hence in the above example Bosnia and Herzegovina met the condition of `if gnppc > 10000`. So we should have written:

```
gen devstatus = "developed" if gnppc > 10000 & gnppc < .
```

Then those observations with the gnppc variable missing are not coded as “developed” countries. As you can see, you need to be very careful when using conditional statements. Make sure to add `& varname < .` so that missing values are erroneously treated as very big valid values.

This example actually shows you how to specify multiple logical conditions as well. In the above example, the data values must meet both the conditions, so the logical operator to use is `&`. [Here](#) is a list of the comparison operators and the Boolean operators allowed in Stata.

How do we execute multiple actions under the same condition? This may be something SAS users find a bit annoying – in SAS, the way to go is to use the DO and END keywords inside the IF-THEN statement. This statement is made possible because of SAS’s built-in loop: SAS’s DATA steps execute line by line and observation by observation. In Stata, however, each command is executed on all observations before the next command is executed, and hence we have no “if-then do loop” equivalent.

Suppose that after working on the data for some time, someone kindly sent you the data of per capita GNP (let’s suppose it’s \$3,000) and access to safe water (likewise, 97%) for Bosnia and Herzegovina, which are missing in the original data. Add the two data values and also add the two curvilinear terms of the GNP variable for this country. One of the things we might do is:

```
replace gnppc = 3000 if country == "BOSNIA AND HERZEGOVINA"
replace safewater = 97 if country == "BOSNIA AND HERZEGOVINA"
replace sqgnppc = gnppc*gnppc if country == "BOSNIA AND HERZEGOVINA"
replace lngnppc = log(gnppc) if country == "BOSNIA AND HERZEGOVINA"
replace devstatus = "developing" if country == "BOSNIA AND HERZEGOVINA"
```

* In this example, we could replace the repeated if-clause with a macro variable in the above example, but we won't get into that subject here. Another way would be to merge this data with the update option.

Sometimes you may need to recode variables by grouping observations. For example, we coded countries with lower than \$10,000 per capita GDP as “developing” yet those countries that do not fall in this criterion were not coded. Suppose you want to classify *all* the countries into three categories: low-income (let's for now define it as lower than \$3,000 per capita GDP = 1), middle-income (\$3,000 to lower than \$10,000 = 2), high-income (\$10,000 = 3). Again, in SAS, you would go with IF-THEN/ELSE statements. In Stata, there is no equivalent to that for the same reason stated above. One of the things we could do instead is:

```
* Create categorical variable (1)
gen incomegrp = .
replace incomegrp = 1 if gnppc < 3000
replace incomegrp = 2 if gnppc >= 3000 & gnppc < 10000
replace incomegrp = 3 if gnppc >= 10000 & gnppc < .
```

Feel free to print and/or open the Stata data to see the new variable *incomegrp*. As noted, Stata treats a missing value (“.”) as the largest possible number (i.e., positive infinity), so it is always safe to cap the maximum with “& *varname* < .”, unless you are absolutely sure your variable does not have any missing value.

Another way you could take is to use the command `-recode-`.

```
* Create categorical variable (2)
recode gnppc ( 0/2999.999 = 1 "low-income" ) ///
             ( 3000/9999.999 = 2 "middle-income" ) ///
             ( 10000/max = 3 "high-income" ) ///
             ( missing = . ), gen(incomegrp) test
```

This command lets you recode variables into a new variable, and create value labels. In the above command, the variable *gnppc* is recoded into a three-category variable *incomegrp* based on our income group definition and value labels are created for each category number. The command `-recode-`'s ranges #1/#2 include everything between #1 and #2, AND the boundaries #1 and #2 themselves, so I added .9999 to the upper limits to ensure those countries with per capita GNP < 3000 or < 10000 are categorized correctly. `/max` provides the maximum value of the recoded variable (in this case, *gnppc*), and `missing` specifies all the missing values. The option `[, test]` is to see if there are no overlap in our rules.

Comparing the value of a floating-point variable like the above example can be [very tricky](#) in computers, however; you may not get exactly what you expect. To avoid the non-integer comparison issue in such cases as this one, you can use (1) above. Or you could use the `cond()` function to accomplish the same task as well.

```
* Create categorical variable (3)
gen incomegrp = cond(missing(gnppc), ., ///
                    cond(gnppc < 3000 , 1, ///
                          cond(gnppc >= 3000 & gnppc < 10000 , 2, ///
                                cond(gnppc >= 10000 & gnppc < . , 3, .))))
```

Anyway, let's save the data here with a new file name "lifeexp_v2.dta", now that we have modified the original by adding some variables.

```
save lifeexp_v2, replace
```

How to subset your data?

Sometimes we don't need write the whole data in our Stata data set. There are two such cases:

- (1) *Subset by columns: You may want to remove from your Stata data set variables that you are sure will not be part of your analysis.*
- (2) *Subset by rows: You may keep only those observations that meet certain criteria.*

Let's take a look at those cases one by one.

(1) Subset by columns

Let's keep using the life expectancy data for this example. The data contains fifteen variables, as we added some in the previous section. Suppose you decide you don't need *datasource*, *dataver*, *safewater2*, *ccode*, and *devstatus* in your current Stata data set. One way to perform this task is to use the commands `-keep-` or `-drop-` in order to, literally, keep or drop your variables from your Stata data set.

```
* Keep the only variables we need
use lifeexp_2, clear
keep iso3n - safewater sqgnppc lngnppc incomegrp
```

When we list consecutive variables, we can use the shorthand conversion "--" (dash) to reduce the amount of typing. In the above example, "iso3n - safewater" means *iso3n* through *safewater*.

Conversely, you can drop the unnecessary variables instead, of course.

```
drop datasource dataver safewater2 ccode devstatus
```

Remember, you just made this data modification to the data in memory, not the Stata data file itself. Hence, you need to save this change if you want to keep it (and do so with a different name if you want to keep the original "lifeexp.dta" intact).

Now, in the above example, what we did was (1) to load *all* the data content onto Stata's memory, and then (2) keep or drop variables. This is pretty straightforward. Sometimes, however, you don't want to load all the data onto memory in the first place. Rather, there may be some situations where you want to load only the variables you are interested in from the beginning.

For example, the data you want to bring in Stata may be huge and not easily manageable if you load all of the data. You can accomplish that task by doing the following.

```
* You can do selective reading for the same result.
use iso3n - safewater sqgnppc lngnppc incomegrp using lifeexp_v2, clear
```

That is,

```
use varlist using filename [, clear]
```

Let's save this file with a new file name "lifeexp_v3.dta".

```
save lifeexp_v3, replace
```

The commands `-infile-` and `-infix-` also allow you to do the same thing when your data source is in a text format (free or fixed-column).

Let's use the "cancer_space_c1.txt" data for this example. Previously, we used the command `-infile-` to read this space separated (free format) text file. It has four variables: *studytime*, *drug*, *age*, and *died*. Now, let's pretend this data is huge and we know the variable *age* is unnecessary. In that case, here is what you would do.

```
* Skip one variable (age).
infile studytime drug _skip str3 died using cancer_space_c1.txt, clear
```

See what is loaded in your Variables window. The variable *age* is not brought in, so you don't have to the once-load-it-and-then-drop-it work. To do this, we use `_skip` in our variable list following the `-infile-` command. You can specify `_skip` more than once. So if you want to read *drug* and *died* only, then

```
infile _skip drug _skip str3 died using cancer_space_c1.txt, clear
```

And if you want to read the first two only, specify `_skip(#)` where `#` is the number of variables to be skipped.

```
infile studytime drug _skip(2) using cancer_space_c1.txt, clear
```

Let's now turn to the "lifeexp_column.raw" data file and read the *iso3n*, *region*, *country*, *lifeexp*, and *gnppc* variables from there. To read this fixed-column formatted data file, we previously use the command `-infile-` with a dictionary. And [we talked about some advantages of the fixed-column format](#) there. So here is what we are going to do: (1) skip *popgrowth* and *safewater*, and (2) reorder the remaining variables so the *country* variable comes first as we are bringing the data in. Your dictionary file would look like this:

```
* Skip popgrowth and safewater, plus read country name first.
```

```

dictionary using lifeexp_column.raw {
    _column(5)  str29 country    %29s
    _column(1)  iso3n           %3f
    _column(4)  region          %1f
    _column(46) lifeexp         %2f
    _column(48) gnppc           %5f
}

```

Notice where we tell Stata to read the country variable, and also notice we do not specify the column numbers of *popgrowth* and *safewater*. Name this dictionary file “lifeexp_colinfl_skip.dct” and run the following line from your do. file.

```

* Read the fixed-column data lifeexp_column.raw.

infile using lifeexp_colinfl_skip.dct, clear

list in 1/10

```

See the Results window and confirm the data is read in as you meant.

(2) Subset by rows

So far, we learned how to subset data by column. Let’s now talk about subsetting data by row, i.e., selecting only the observations that meet your criteria. Suppose, for example, that in your cross-national research of life expectancy, you decide to focus on developing countries and hence want to keep developing countries only in your data set. Suppose also that you continue to use the definition of developing countries as those with their per capita GNP under \$10,000.

Again, the most straightforward way to understand is to use the commands `-keep-` or `-drop-` along with a conditional argument. We load the Stata data set “lifeexp_v3.dta” and then delete the observations that do not meet our condition below.

```

* Keep developing countries only.

use lifeexp_v3, clear
keep if gnppc < 10000

```

Or, you can drop observations whose per capita GNP is 10,000 or above.

```

. keep if gnppc < 10000
(22 observations deleted)

```

This way, you first load all the data in Stata’s memory, and then delete unnecessary observations. Just like we did in the previous subsection, though, you can load only those observations that meet your criteria.

```

* Subset by rows, read selected observations only.

use lifeexp_v3 if gnppc < 10000, clear

```


Selective reading can be done with the commands `-infile-` and `-infix-` for your text format data. Here, we will go over an example using `-infile-` to read the fixed-column data “`lifeexp_column.raw.`”

```
*      Subset by rows, read selected observations from fixed-column data.
infile using lifeexp_colinfl.dct if gnppc < 10000, clear
```

How to combine Stata data sets?

Now that we have talked about subsetting data, let’s talk about how to do the opposite, i.e., how to combine Stata data sets. Just like subsetting, there are two dimensions to work on.

- (1) *Merging related Stata data sets (combine columns): You may want to add new variables to your data.*
- (2) *Appending Stata data sets (combine rows): You may want to add more observations to your data.*

Let’s see how to do these two tasks.

(1) Merging related Stata data sets

Merging two related Stata data sets is a simple task. If you want to combine related Stata data sets, use the command `-merge-`. You need to use, as a unique identifier of each observation, one (or a combination of two or more) variable(s) that your Stata data sets to be merged have in common with the same variable name. Your matching variable must be sorted first in the same way in both of the source Stata data sets so that Stata can correctly match the observations.

Let’s see an example. Suppose we have received from a research colleague a Stata data set named “`physicians.dta`” that includes the number of physicians per 1,000 people. We want to combine this data with our “`lifeexp.dta`” Stata data set. The two data sets have a variable `iso3n` in common, coded in the same way, so we can use this variable as a matching variable. We need to sort those two Stata data sets by the `iso3n` variable and then combine the two Stata data sets into a new data set. The quickest way to perform this task is:

```
use lifeexp_v2, clear
merge iso3n using physicians, sort
```

Of course, you can accomplish the same by doing the following, too.

```
*      Merge data
use physicians, clear
sort iso3n
```

```
save, replace
use lifeexp_v3, clear
sort iso3n
merge iso3n using physicians
```

As you can see, the basic form of the command `-merge-` is:

```
merge varlist using filename [, sort]
```

In the first example, we use the `[, sort]` option of the command `-merge-`. This option tells Stata to sort the master and the using data sets by the matching variable (in this case, `iso3n`) before merging, if they aren't sorted yet. Of course, you could sort each of the data and then merge them, like in the second example. In that case, use the command `-sort-` first. With this command, Stata sorts your data in ascending order. You need to specify your matching variable right after this command. We first sort the “physicians” data set by `iso3n`. By so doing you only make a change to the data in memory, so you need to save it so that you can use this sorted data for your merge work. Next, we load the data “lifeexp.dta” to merge “physicians.dta” with, and again sort it by `iso3n`. And then, we merge “physicians.dta” with the data currently in memory, “lifeexp.dta,” by `iso3n`.

Place your matching variable(s) in `varlist`. This way you are telling Stata by what matching variable the two data sets are to be merged. Let's list `country` and `physicians` for the first ten observations.

```
. list country physicians in 1/10
```

	country	physic~s
1.	Albania	1.3741
2.	Azerbaijan	3.9172
3.	Argentina	.
4.	Austria	2.2
5.	Armenia	3.9244
6.	Belgium	3.3
7.	Bolivia	.45
8.	Bosnia and Herzegovina	1.5703
9.	Brazil	1.36
10.	Bulgaria	3.1696

Now, when you merge two Stata data sets this way, the resulting new data set can have three groups of observations: (1) Observations that only data set in memory (in this case, “lifeexp.dta”) contributed to the new data set; (2) observations that only using data set (“physicians.dta”) did; (3) observations that the both Stata data sets did. In Stata, the command `-merge-` automatically tracks which of the source data sets contributed each observation in the new data set and creates a tracking variable `_merge`. See the Variables window; it is there at the bottom of your variable list.

Let's see the breakdown of this variable. We will use the command `-tabulate-`.

```
* How many cases are from each data set?
tab _merge
```

```
. tab _merge
```

<code>_merge</code>	Freq.	Percent	Cum.
1	9	12.00	12.00
2	7	9.33	21.33
3	59	78.67	100.00
Total	75	100.00	

So 59 countries are in both of the data sets, and 9 are in “lifeexp.dta” only, whereas 7 are in “physicians.dta” only. We know that we are interested in each country’s life expectancy, so we are sure we will not use the cases where `_merge == 2`.

```
drop if _merge == 2
drop _merge
```

In this example above, there is no overlapping variable between the two data sets. But if the master data set and the using data set have any variable in common, then by default the values from the master data take precedence. If the master data’s values are missing and you want to retain the using data’s values for such cases, use the option `[, update]`. Further, if you use the option `[, update replace]`, then non-missing values from the master data set are replaced with those from the using data set. For more details, see `-help merge-`.

(2) Appending Stata data sets

Suppose you obtained a Stata data set “lifeexp_moreobs.dta”, which contains more country observations for the same variables as those in “lifeexp.dta”. Suppose also you want to stack it with the Stata data set “lifeexp.dta” The “lifeexp_moreobs.dta” file has 10 observations (feel free to run `list` to see the content of the file).

Let’s see how to do in Stata. What you need is the command `-append-`.

```
* Stack more observations.
append using lifeexp_moreobs
```

The program stacks the two Stata data sets in the order of the master data and the using data, and now we have 78 observations in the data set. Here is the printout of the “country” variable (partly omitted for the sake of space).

```
. list country
```

	country
1.	ALBANI A
2.	AZERBAI JAN
[Omitted in-between for the sake of space]	

} Obs 1-68 are from the master Stata data set.

69.		Algeria
70.		Bangladesh

71.		Botswana
72.		Burundi
73.		Cambodia
74.		China
75.		India

76.		Kenya
77.		Oman
78.		Syrian Arab Republic

Obs 69-78 are from the
using Stata data set.

This means that whatever you have your master data sorted, that order will be lost if you append another data. If you want to have your new data sorted the same way the master was sorted, you need to run `-sort-` again.

How to label and format your variables?

Finally, we will briefly discuss two things, (1) how to label your variables, (2) how to label the values of your variables, and (3) how to label your data set. The command for these documentation tasks is `-label-`, but you need to use some variation of syntax depending on what you label.

(1) Variable labels

OK, let's first start with labeling variables. What you need is the command `-label variable-`. Stata allows for up to 80 characters (including blanks) long for a label. Needless to say, short and clear labels are better.

```
* Labeling variables
la var iso3n "ISO country code"
la var region "Region code"
la var country "country name"
la var popgrowth "Pop growth %"
la var lexp "Life expectancy"
la var gnppc "Per cap GNP $"
la var safewater "% of pop w/ safe water"
la var incomegrp "Income group (gnppc)"
la var physicians "# of docs/1K pop"
la var lngnppc "Log of gnppc"
la var sqgnppc "Sq term of gnppc"
```

Since `incomegrp` is created from the variable `gnppc`, it is a good idea to include that information in the variable label (“(gnppc)”). Once you do this, your variable labels will be stored in the data set. You now see these labels in the Variable Window.

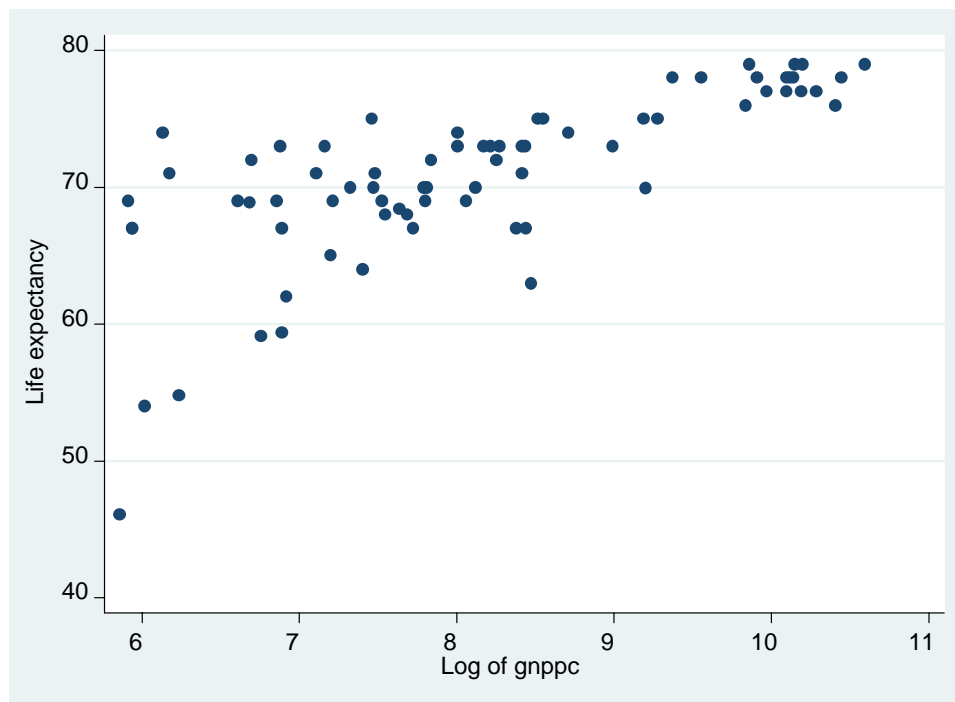
Name	Label
iso3n	ISO country code
region	Region code
country	country name
popgrowth	Pop growth %
lexp	Life expectancy
gnppc	Per cap GNP \$
safewater	% of pop w/ safe water
sqgnppc	Sq term of gnppc
lngnppc	Log of gnppc
incomegrp	Income level / gnppc
physicians	# of docs/1K pop

And they show up in your data description output or graphics. Let's try the followings.

* Labels show up...

```
de
graph twoway scatter lexp lngnppc
```

First, the command `-describe-` now outputs your variable label as well (see your Result window). Also, the graph has your variable labels as well.



(2) Value labels

Now, let's talk about Stata's value labels. We will work on the variables *region* and *incomegrp*.

```
tab region
tab incomegrp
```

```
. tab region
```

Region code	Freq.	Percent	Cum.
1	44	56.41	56.41
2	15	19.23	75.64
3	10	12.82	88.46
4	3	3.85	92.31
5	4	5.13	97.44
6	2	2.56	100.00
Total	78	100.00	

And assuming you created the income variable with [this method](#), here's what you get.

```
. tab incomegrp
```

Income level / gnppc	Freq.	Percent	Cum.
1	36	49.32	49.32
2	19	26.03	75.34
3	18	24.66	100.00
Total	73	100.00	

In the `-tabulate-` output, what the code means is unclear without the data's codebook. For the *region* variable, the data codebook would tell us:

region

- 1 = Europe
- 2 = North America
- 3 = South America
- 4 = Sub Saharan Africa
- 5 = Asia
- 6 = Middle East

And *incomegrp* is a variable we created, so we know what each category means.

incomegrp

- 1 = Low income (\$0 < 3,000 per cap GNP)
- 2 = Middle income (\$3,000 < 10,000 per cap GNP)
- 3 = High income (\$10,000 or beyond per cap GNP)

Yet, a better idea than to check the codebook every time would be to attach a set of value labels to those categories, so that the meanings of the coded values are clear at a glance. In Stata, you need to take two steps to accomplish this.

Step 1: Define (= create) a set of value labels.

Step 2: Associate the labels created in Step 1 with the appropriate variable(s).

So, this means labels and variables are not in a one-to-one, fixed relationship; they are separate processes. Therefore, you can attach a set of labels to multiple variables. Say, for example, if you create a set of labels 0 = “No” 1 = “Yes” and you have many variables where this set of labels is appropriate, you can attach this label set to all such variables. If a label, on the other hand, is for a particular variable only, it is a good idea to name the label name the same as the variable’s name so you can stay instantly see which value label matches what variable.

The command to use to do Step 1 is `-label define-` (literally). The basic form is:

```
label define lblname # "label" [# "label" ...] [, add modify nofix]
```

Let me briefly explain two of the three options there. First, [, add] lets you add new entries to a new or existing label set named *lblname*. Without this option, you can create only new *lblnames*. Second, [, modify] allows you to modify/delete existing entries from a label set or add new entries. As for [, nofix], see `-help label define-`.

So, here is what we can do to label the categorical values of the variable *region*.

```
label define region 1 "Europe" ///
                    2 "N Amr"  ///
                    3 "S Amr"  ///
                    4 "SS Afr"  ///
                    5 "Asia"   ///
                    6 "M East"  ///
```

The `label define region` part declares to Stata that you are now about to define the label set named *region*. Then you actually specify the content of the label set *region*, first the category number, then the corresponding text label. Of course, you can use the option [, add] and do this for the same result.

```
label define region 1 "Europe"
label define region 2 "N Amr" , add
label define region 3 "S Amr" , add
label define region 4 "SS Afr" , add
label define region 5 "Asia" , add
label define region 6 "M East" , add
```

Either way is fine. Now this label set *rgn* is stored in memory. You can take a look by listing the names and contents of value labels stored in memory. Use the command `-label list-`.

```
label list
```

And here you see the label set you just created. Looks everything is okay.

```
. label list
region:
      1 Europe
      2 N Amr
      3 S Amr
      4 SS Afr
      5 Asia
      6 M East
```

If you just need to have the names of value labels in memory, use the command `-label dir-` (feel free to try it through the interactive Command window).

Now, at this point, there is no connection whatsoever between this label set *rgn* and whatever variables in the data. So, you need to create such association(s). That's the next step (Step 2), where you attach this label set to (an) appropriate variable(s), in this case, *region*. To do so, we use the command `-label value-`. Here is the basic form.

```
label values varlist [lblname|. ] [, nofix]
```

Just a quick note about the option `[, nofix]`: it is just about display formats. It prevents display formats from being widened according to the maximum length of the value label. Anyway, let's attach our label set *rgn* to the variable *region*.

```
label values region region
```

Now, let's run `-describe- region`.

```
. de region
variable name      storage   display   value     variable label
                   type     format   label
region             byte     %8.0g    region    Region code
```

You can see the value label set *region* is now attached to the variable *region*. Tabulate this variable.

```
. tab region
Region code |      Freq.   Percent   Cum.
-----|-----|-----|-----
Europe      |        44    56.41    56.41
N Amr       |        15    19.23    75.64
S Amr       |        10    12.82    88.46
SS Afr      |         3     3.85    92.31
Asia        |         4     5.13    97.44
M East      |         2     2.56   100.00
-----|-----|-----|-----
Total       |        78   100.00
```

Now it's easier and quicker to see in your output what the code means.

As for the `[lblname|.]` in the above basic form, we just specify the label name (i.e., *rgn*) there. If instead a "." (period) is specified instead of a label name, any existing label set is detached from the variable(s) listed there. So, for example, if you do:


```
label values region .
```

This detaches the label set *region* from the variable *region*. Try it and confirm the label is detached by executing `-describe region-` interactively (i.e., from the Command window). Of course, the label set is still stored in memory, so you can re-attach it to *region* by running again:

```
label values region region
```

Let's create a label set and attach it to the variable *incomegrp* as well. Suppose we do the following.

```
label define incomegrp ///
  1 "low-income country group ($0 < 3,000 per cap GNP)" ///
  2 "middle-income country group ($3,000 < 10,000 per cap GNP)" ///
  3 "high-income country group ($10,000 or beyond per cap GNP)"
label value incomegrp incomegrp
```

And now see how it looks in our output...

```
tab incomegrp
```

And here is what you get.

```
. tab incomegrp
```

Income level / gnppc	Freq.	Percent	Cum.
low-income country group (\$0 < 3,000 pe	36	49.32	49.32
middle-income country group (\$3,000 < 1	19	26.03	75.34
high-income country group (\$10,000 or b	18	24.66	100.00
Total	73	100.00	

As you can see, the value labels are too long they are truncated in the output. In this case, you can still manage to see the key part of what each group is about (i.e., low, middle, or high income), but even that is not always the case. Suppose, for example, you have received a data file from a survey about people's attitudes about the federal government's role in certain policies. You know it asked respondents about their thought regarding welfare policies (the variable "welfare"), so you decide to check its distribution, and here is what you get...

```
. tab welfare
```

welfare	Freq.	Percent	Cum.
Yes, I believe that the U.S. government	496	16.66	16.66
Yes, I believe that the U.S. government	972	32.65	49.31
No, I don't believe that the U.S. gover	1,005	33.76	83.07
No, I don't believe that the U.S. gover	504	16.93	100.00
Total	2,977	100.00	

It may sound like a bit extreme example, but you get the point. The labels are truncated, AND the remaining part does not allow you to uniquely identify each of those categories. This type of problem happens more often than you might think.

With the command `-labelbook-`, Stata reports potential problems with value labels in more details.

```
labelbook welfare, length(10)
```

```
value label welfare

      values                labels
  range:  [1,4]           string length: [78,86]
      N:    4             unique at full length: yes
  gaps:   no             unique at length 10: no
missing .*: 0           null string: no
                        leading/trailing blanks: no
                        numeric -> numeric: no

  definition
    1  Yes, I believe that the U.S. government should definitely encourage the
       activity
    2  Yes, I believe that the U.S. government should somewhat encourage the
       activity
    3  No, I don't believe that the U.S. government should somewhat discourage the
       activity
    4  No, I don't believe that the U.S. government should definitely discourage
       the activity

  variables:  welfare
```

It gives you information about more different types of potential problems, but for now, our issue is the label's truncation (for more details about this command, `-help labelbook-`). The option `[, length(#)]` allows you to check if your value labels are unique to length `#`. By default `#` is set to 12, but here we check it to length 10. It says the label duplicates at the length 10 (“unique at length 10: no”). So, what lesson can we draw?

- (1) Place key part of the information at the top of the label! And
- (2) Make your label as short as possible, AND make their meanings clear at the same time.

As such, defining labels demands careful planning.

Now, back to our own life expectancy data file. Let's modify our truncated value labels “incomegrp”. You don't have to delete the old label, recreate it, and reattach it. Instead, you can simply use the option `[, modify]`.

```
label define incomegrp 1 "low-income" , modify
label define incomegrp 2 "middle-income" , modify
label define incomegrp 3 "high-income" , modify
```

Execute the above and try `tab incomegrp` again.

```
. tab income
```

Income level / gnppc	Freq.	Percent	Cum.
low-income	36	49.32	49.32
middle-income	19	26.03	75.34
high-income	18	24.66	100.00
Total	73	100.00	

Looking good. Don't forget to document in your research note or research log how you define each category though!

(3) Data labels

Finally, let's quickly go over how to label your Stata data set. The command to use is `-label data-`. It allows you to attach a label (up to 80 characters) to the dataset in memory. So, for example, we can do this:

```
label data "Expanded life expectancy data"
save lifeexp_v4, replace
```

Dataset labels are displayed when you use the dataset and when you describe it. So, it helps you with your data management and organization. Choose a quick, clear label to the data that tells you what the data is about or which data you are working with, etc. Finally we saved the updated data with a new name "lifeexp_v4".

Again, remember, you just made all those changes above to the data in memory. If you want to permanently save this in the file, you need to save it.

4. How To Get Descriptive Statistics

So far, we focused primarily on data management aspects of Stata basics. From this section on, we also start learning more about conducting analysis and producing report in Stata. In this section, we will learn how to explore data and get a good idea what it looks like—what your variables' central tendencies are, how they are dispersed, how they are distributed, etc.—before getting down to data analysis. The below are the commands we will cover in this section.

- (1) `describe`
- (2) `codebook`
- (3) `summarize`
- (4) `tabstat`
- (5) `mean`
- (6) `table`
- (7) `bysort` (or `by`, `sort`)
- (8) `tabulate`
- (9) `correlate`
- (10) `stem`

- (11) `histogram`
- (12) `sktest`
- (13) `graph`

Remember, we will cover the basic, most common usages of those procedures. For further information, use the help menu.

We will continue to use the Stata data set “lifeexp_v4”. Let’s first take a look at the information about the data in memory by using `-describe-` (you are now familiar with this command now). This is an easy and quick way to get a description of your data.

```
. de
Contains data from lifeexp_v4.dta
  obs:      78          Expanded life expectancy data
  vars:      11          16 Oct 2009 16:52
  size:    5,148 (99.9% of memory free)

-----
variable name  storage  display  value  variable label
              type  format  label
-----
iso3n         float  %8.0g
region        byte   %8.0g    region    Region code
country       str28  %28s    country  country name
popgrowth     float  %9.0g    Pop growth %
lexp          float  %8.0g    Life expectancy
gnppc         long   %12.0g   Per cap GNP $
safewater     byte   %8.0g    % of pop w/ safe water
sqgnppc       float  %9.0g    Sq term of gnppc
lngnppc       float  %9.0g    Log of gnppc
incomegrp     float  %57.0g   incomegrp Income level / gnppc
physicians    float  %9.0g    # of docs/1K pop

Sorted by:
```

In the above example output, I simply ran `-describe-` to get information of all the variables in the data, but you can also get descriptions of specific variables; just add variable names you want to describe (e.g., `describe gnppc`). The output above looks like a table of contents for the data; it shows you the information that is stored in the data: the data set name, the number of observations and variables in the data, the date the data was last modified, variable names, type, lengths, formats, labels, value labels. Notice also that the data label we just a moment ago attached (i.e., “Expanded life expectancy data”) is also displayed there.

`-describe-` has some useful options. Particularly helpful are `[, short]` and `[, fullnames]`. The `[, short]` option gives you general information of the data only (i.e., no information about each specific variables), and sometimes that’s all you want to know at a moment. The `[, fullnames]` option displays, literally, the full names of each variables without abbreviating long ones. This is useful when you want to see the complete variable names.

The command `-codebook-` is also to describe data contents, but it also gives you basic descriptive statistics as well for each variable in the data in memory. Just like `-describe-`, `-codebook-` alone outputs descriptions for all the variables, but if you list specific variables after the command, Stata returns a codebook for those variables only. For example,

```

. codebook gnppc lexp
-----
gnppc                                     Per cap GNP $
-----
      type: numeric (long)
      range: [350,39980]
unique values: 71                          units: 1
                                          missing .: 5/78
      mean: 7864.88
      std. dev: 10133.6
      percentiles:      10%      25%      50%      75%      90%
                       510      1290      3000      9920      25380
-----
lexp                                       Life expectancy
-----
      type: numeric (float)
      range: [46.070293,79]
unique values: 28                          units: 1.000e-06
                                          missing .: 0/78
      mean: 70.8536
      std. dev: 6.35482
      percentiles:      10%      25%      50%      75%      90%
                       62.9939  68.4361  72      75      78

```

[, `compact`], one of this command's option, give you a compact report on the variables.

```

. codebook, compact
-----
Variable  Obs Unique   Mean   Min   Max  Label
-----
iso3n     78   78 411.4744     8   890  ISO country code
region    78    6  1.897436     1     6  Region code
country   78   78      .         .     .  country name
popgrowth 78   40  1.197097    -.5  3.555356  Pop growth %
lexp      78   28  70.85359 46.07029     79  Life expectancy
gnppc     73   71  7864.877    350  39980  Per cap GNP $
safewater 50   31    76.38     28   100  % of pop w/ safe water
sqgnppc   73   71  1.63e+08  122500  1.60e+09  Sq term of gnppc
lngnppc   73   71  8.135164  5.857933  10.59613  Log of gnppc
incomegrp 73    3  1.753425     1     3  Income level / gnppc
physicians 66   63  2.144158  .0451  4.9256  # of docs/1K pop

```

Another option [, `problems`] tells you *potential* data problems, so you can use this option to check your data and detect possible data issues. Feel free to try it. For more details, type and execute `-help codebook problems-`.

The `-summarize-` is probably one of the first commands for you to use for data exploration purposes. It gives you simple descriptive statistics, including, by default, number of observations, means, standard deviations, minimum, and maximum for numerically-coded variables (Note: this means that your categorical variables, such as *region* and *incomegrp* in this case, are treated as if they were numeric variables. That's how Stata operates. Keep in mind they actually are not numeric variables – their values does not have any substantive meanings themselves).

```
. su
```

Variable	Obs	Mean	Std. Dev.	Min	Max
iso3n	78	411.4744	277.0349	8	890
region	78	1.897436	1.315144	1	6
country	0				
popgrowth	78	1.197097	1.075551	-.5	3.555356
lexp	78	70.85359	6.354819	46.07029	79
gnppc	73	7864.877	10133.55	350	39980
safewater	50	76.38	17.48222	28	100
sqgnppc	73	1.63e+08	3.24e+08	122500	1.60e+09
lgnppc	73	8.135164	1.3474	5.857933	10.59613
incomegrp	73	1.753425	.8296723	1	3
physicians	66	2.144158	1.28024	.0451	4.9256

This command's option [, detail] finds you additional descriptive statistics (including skewness, kurtosis, the four smallest and four largest values, and percentiles) for each variable you list after the command or each of all the variables in the data if no variable is specified. Let's for example get descriptive information of *gnppc* and *lexp*.

```
. su gnppc lexp, detail
```

Per cap GNP \$					
Percentiles	Smallest				
1%	350	350			
5%	380	370			
10%	510	380	Obs		73
25%	1290	380	Sum of Wgt.		73
50%	3000		Mean		7864.877
		Largest	Std. Dev.		10133.55
75%	9920	29240			
90%	25380	33040	Variance		1.03e+08
95%	29240	34310	Skewness		1.497053
99%	39980	39980	Kurtosis		3.999793
Life expectancy					
Percentiles	Smallest				
1%	46.07029	46.07029			
5%	54.93824	54			
10%	62.99385	54.79254	Obs		78
25%	68.4361	54.93824	Sum of Wgt.		78
50%	72		Mean		70.85359
		Largest	Std. Dev.		6.354819
75%	75	79			
90%	78	79	Variance		40.38372
95%	79	79	Skewness		-1.367315
99%	79	79	Kurtosis		5.539534

The commands for descriptive information thus far do not let you request for specific statistics, however. The command `-tabstat-` gives you more flexibility regarding that. The basic form is:

```
tabstat varlist [if] [in] [weight] [, options]
```

The [, statistics()] option allows you to specify which summary statistics you want to see. Without this option (by default), -tabstat- returns only means. The below table summarizes frequently used descriptive statistics you can specify with the [, statistics()] option.

<u>n</u>	Number of non-missing observations	<u>sum</u>	Sum
<u>count</u>	Same as n.	<u>median</u>	Median (Percentile 50)
<u>mean</u>	Mean	<u>variance</u>	Variance
<u>sd</u>	Standard deviation	<u>range</u>	Range
<u>cv</u>	Coefficient of variation	<u>semean</u>	Standard error of mean
<u>min</u>	Minimum	<u>skewness</u>	Skewness
<u>max</u>	Maximum	<u>kurtosis</u>	Kurtosis
<u>p#</u>	# = 1, 5, 10, 25, 50, 75, 90, 95, 99. Percentile. P50 = median.	<u>q</u>	Equivalent to specifying p25 p50, p75
<u>iqr</u>	Interquartile range (p75 – p25)		

You can perform separate analyses for different groups. Also, you can get information for specific variables. Suppose you want to get region-by-region descriptive statistics (N, mean, standard deviation, minimum, skewness) for life expectancy and per capita GNP.

```
tabstat lexp gnppc, stat(n mean sd skew) columns(statistics) by(region)
```

And you get the following output.

```
. tabstat lexp gnppc, stat(n mean sd skew) columns(statistics) by(region)
Summary for variables: lexp gnppc
by categories of: region
region |          N          mean          sd  skewness
-----|-----
Europe |          44  73.06818  4.150639  -.1771728
       |          42 10553.81 11709.98  .9148322
N Amr  |          15  70.93498  6.272047 -1.251487
       |          13  5704.308  8559.009  2.057661
S Amr  |          10    70.3    3.831159  -.8667745
       |          10   3645   2254.528  .6541359
SS Afr |           3  56.14836  8.912848  -.5786518
       |           3  2036.667  2396.588  .6525333
Asia   |           4  59.43316  6.607148  .8683174
       |           3  723.3333  187.1719  -.6262426
M East |           2  69.18902  1.064798         0
       |           2    5995  5550.788         0
Total  |          78  70.85359  6.354819 -1.367315
       |          73  7864.877 10133.55  1.497053
```

(The upper line is for *lexp*, the lower for *gnppc*, as in the order of the command line).

As you can see, I specified `n`, `mean`, `sd`, and `skewness` in the `[_statistics()]` option, used the `[_by()]` option to get region-by-region statistics, and added the `[_columns()]` option to specify how to display my output. `[_columns(statistics)]` put your statistics in the columns. The default is `[_columns(variables)]`.

Another point I want to mention is how to get descriptive information for a subset of the data. Say, suppose you want to get descriptive information for countries with per capita GNP below \$10,000. We can do that by simply adding the `[if]` qualifier.

```
tabstat lexp gnppc if gnppc < 10000, ///
      stat(n mean sd skew) columns(statistics) by(region)
```

The `[if]` qualifier is valid in most of the Stata commands, and it is a very efficient way because you don't have to create a lot of Stata data sets every time you want to examine different subsets.

Now we have obtained some information about the variables' central tendency, dispersion, and a glimpse of distributional characteristics. We can further explore and check how our variables are distributed. One of the common ways to do so is to use the command `-stem-`, which literally produces a stem-and-leaf plot. Let's for example display a stem-and-leaf for `gnppc`.

```
stem gnppc
```

```
. stem gnppc
Stem-and-leaf plot for gnppc (GNP per capita $)
0**** | 0350,0370,0380,0380,0410,0460,0480,0510,0740,0800,0810,0860, ... (28)
0**** | 2070,2180,2260,2420,2440,2470,2540,2990,3000,3160,3360,3530, ... (15)
0**** | 4350,4510,4520,4620,4630,4780,4990,5150
0**** | 6070
0**** | 8030,9780,9920
1**** | 0670,1740
1**** |
1**** | 4100
1**** |
1**** | 8710,9170
2**** | 0090,1410
2**** |
2**** | 4210,4280,4780,5380,5580
2**** | 6570,6830
2**** | 9240
3**** |
3**** | 3040
3**** | 4310
3**** |
3**** | 9980
```

The information in the vertical bar on the left side is the stem, and the digits to the right are the leaves. They in combination reconstruct a data value in your data set. So, in the first stem-and-leaf line, you have `00350 = 350`, `00370 = 370`... and the `(28)` at the right of the stem-and-leaf line means there are 28 leaves on this line (too many to display). The stem-and-leaf line at the bottom shows the largest value of the `gnppc` variable is 39980. In line with the skewness statistics we got above, the stem-and-leaf plot suggests the variable is somewhat skewed. Let's try and see its logged version.


```

. stem lngnppc
Stem-and-leaf plot for lngnppc (Log of gnppc)
lngnppc rounded to nearest multiple of .01
plot in units of .01

5** | 86,91,94,94
6** | 02,13,17
6** | 23
6** |
6** | 61,68,70,76
6** | 86,88,89,89,92
7** | 11,16
7** | 20,22,33
7** | 40,46,47,48,52,55
7** | 64,69,72,79
7** | 80,81,84
8** | 00,01,06,12,17
8** | 22,25,27,38
8** | 41,42,44,44,47,52,55
8** | 71
8** | 99
9** | 19
9** | 20,28,37
9** | 55
9** |
9** | 84,86,91,97
10** | 09,10,12,14,15,19
10** | 20,28
10** | 41,44
10** | 60

```

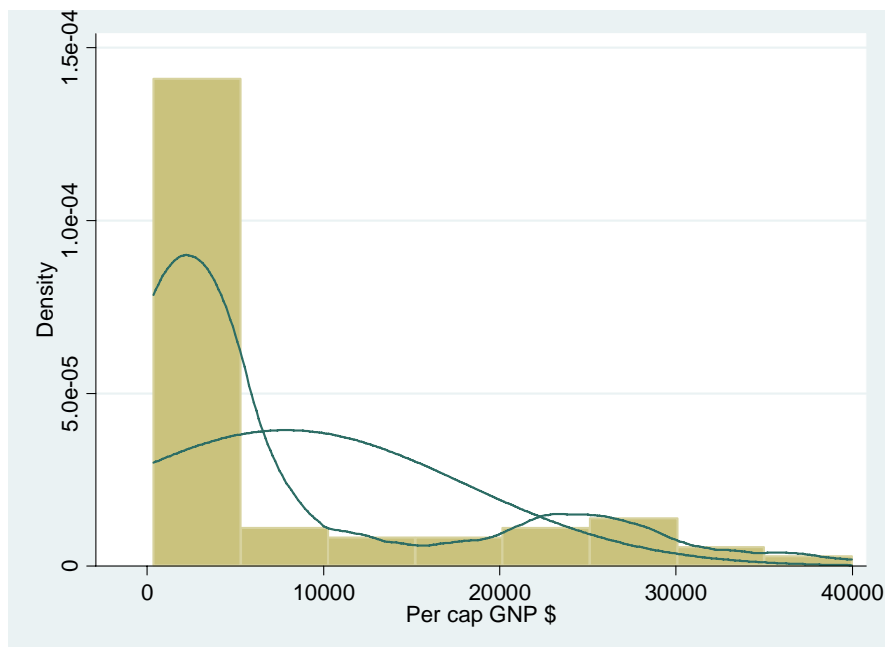
Notice that Stata gives you information about rounding and units. So you can see the smallest observation is about 5.86 and the largest observation is about 10.60. Looks like logging the variable helps correct for skewness somewhat.

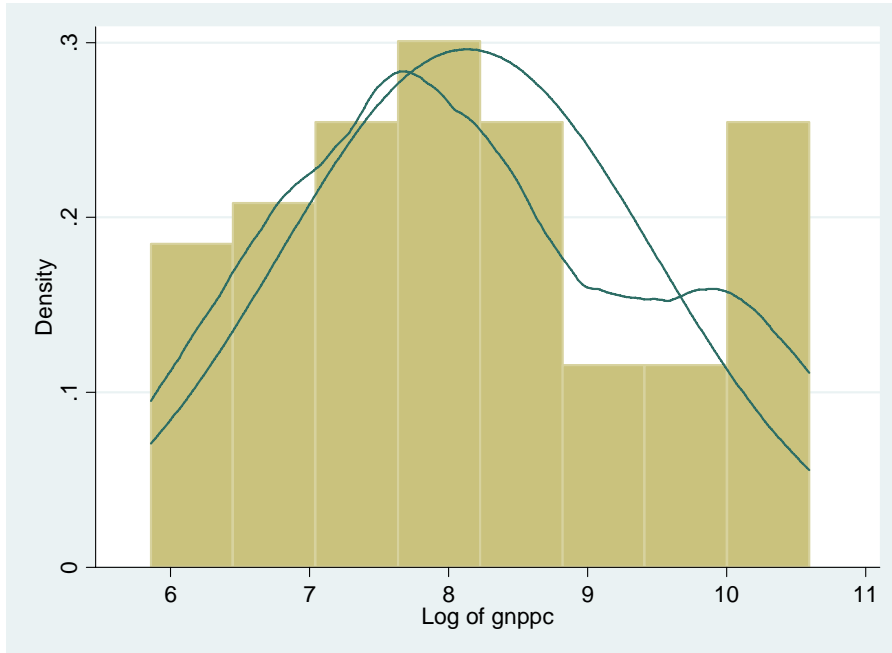
Another way to get similar information is to use the command `-histogram-`. Let's get a density plot for the same two variables, with a normal and a kernel densities.

```

histogram gnppc, normal kdensity
histogram lngnppc, normal kdensity

```

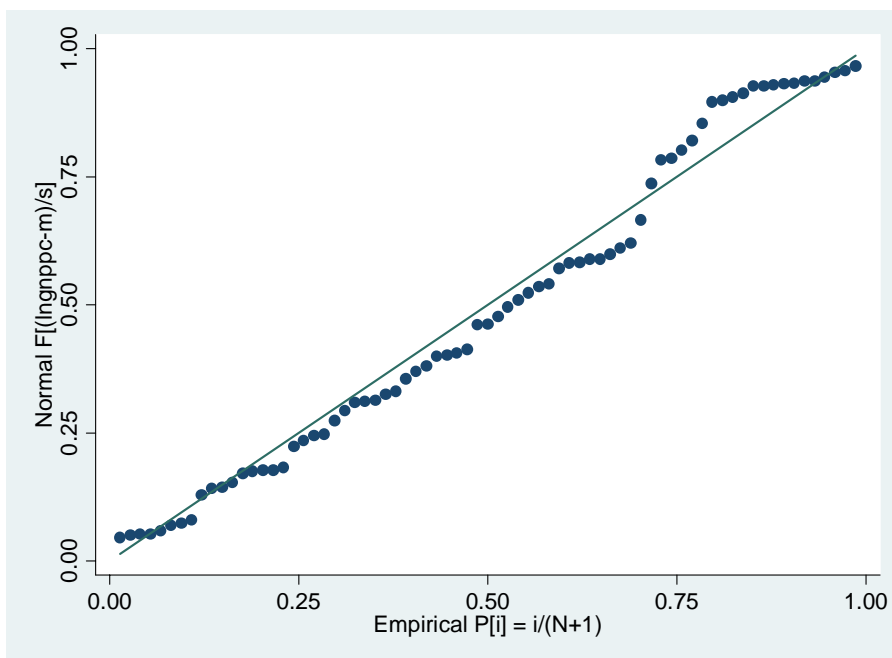




The command histogram has many options (for details, run `-help hist-`). Here, we used `[, normal]` and `[, kdensity]` to overlay our histogram with normal and kernel density lines.

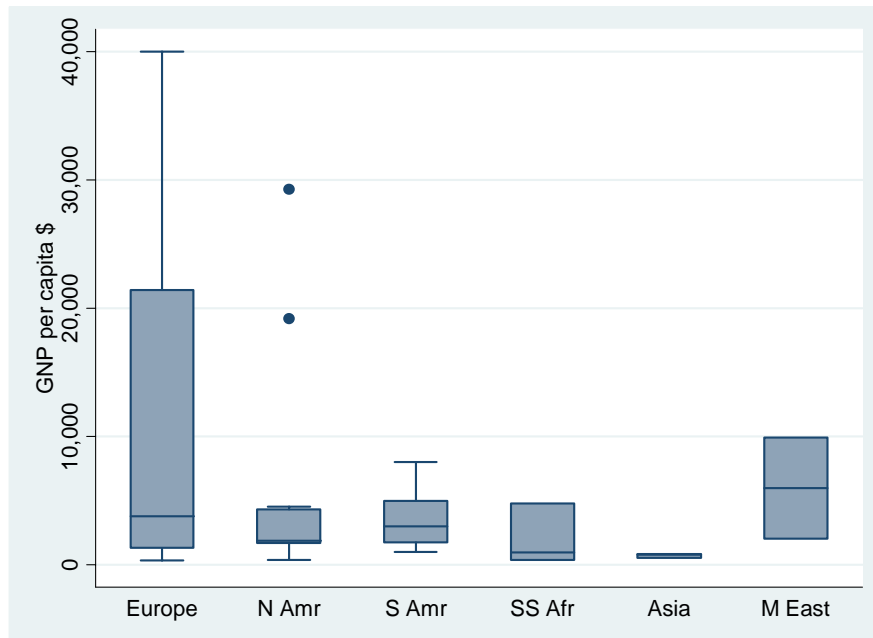
Stata has other standard distributional diagnostics tools. Let's get normal probability plots for *lngnppc*.

```
pnorm lngnppc
```



And let's also get box plots, region by region. To display your box plots by group, add the option `[, over(varname)]`.

```
graph box gnppc, over(region)
```



We just saw how to check distributional characteristics of continuous variables. We can of course explore distributions of categorical variables. [As we have already learned](#), `-tabulate-` generates frequency tables.

```
tab incomegrp
```

```
. tab incomegrp
```

Income level / gnppc	Freq.	Percent	Cum.
low-income	36	49.32	49.32
middle-income	19	26.03	75.34
high-income	18	24.66	100.00
Total	73	100.00	

Notice that the table does not show how many missing values this variable has. In Stata, missing values are not displayed in its frequency table by default, so you need to tell Stata to show them in the table by using the `[, missing]` option.

```
tab incomegrp, missing
```

```
. tab incomegrp, missing
```

Income level / gnppc	Freq.	Percent	Cum.
low-income	36	46.15	46.15
middle-income	19	24.36	70.51
high-income	18	23.08	93.59
.	5	6.41	100.00
Total	78	100.00	

In your data check and exploration, it is good idea to always ask Stata to display missing values in frequency tables, because it can help you detect data errors that you couldn't find otherwise. For example, because this *incomegrp* variable was created from another variable *gnppc*, you could check whether the missing values of the latter are not coded as valid values in this new variable, and so on.

By adding the [, plot] option, you can get a quick, albeit primitive, ASCII bar chart alongside the corresponding frequency table.

```
tab incomegrp, plot
```

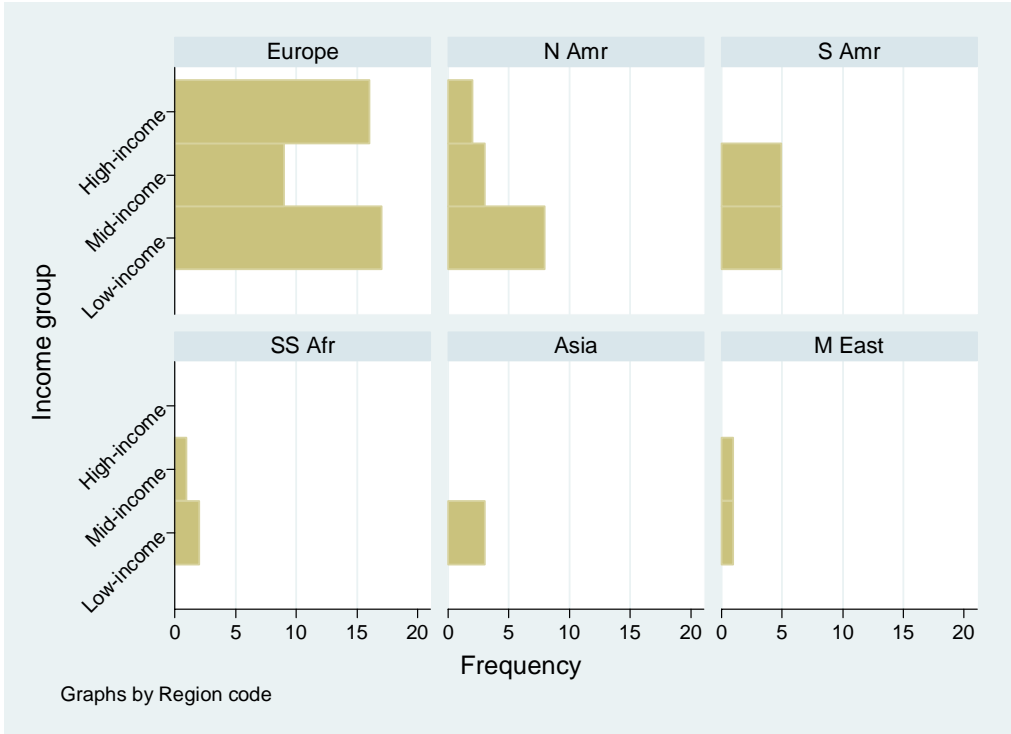
Income level / gnppc	Freq.	
low-income	36	*****
middle-income	19	*****
high-income	18	*****
Total	73	

For a more sophisticated frequency plots, you can use `-histogram-` command, or the `-graph bar-` or `-graph hbar-` commands. `-graph var-` produces vertical bar charts, whereas `-graph hbar-` generates horizontal ones. With `-histogram-`, because we want frequencies of each income group category, use the [, frequency] option. With `-graph bar-` or `-graph hbar-` commands, specify (stat) as (count) so that Stata plots the number of cases for each category.

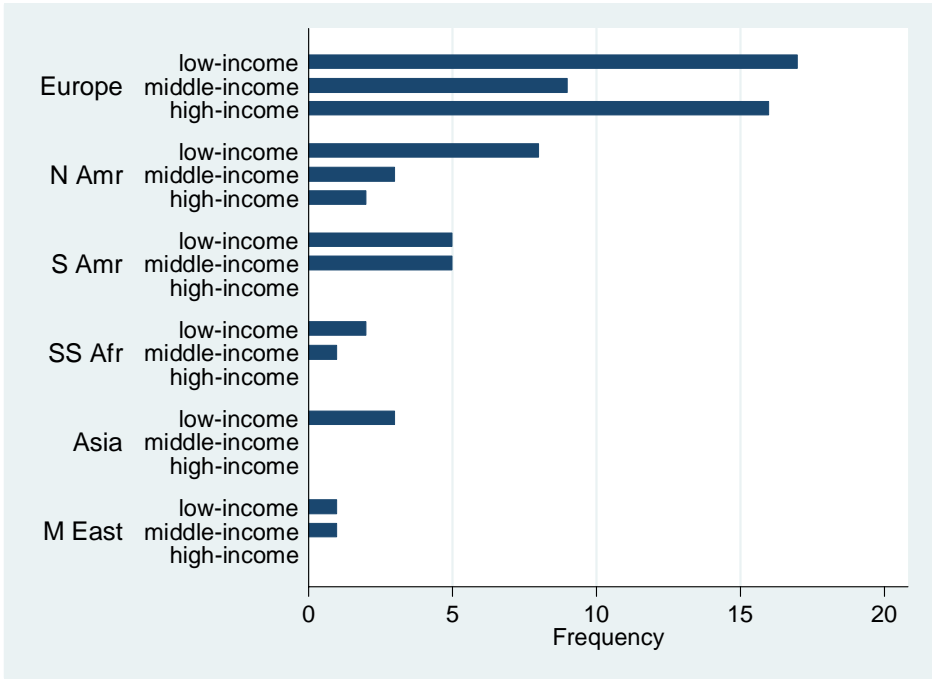
Let's try getting horizontal frequency plots of income groups by region.

```
* Use -histogram-
# delimit;
histogram incomegrp,
  by(region)
  freq discrete horizontal
  ylabel(0(5)20, grid)
  xlabel(1 "Low-income" 2 "Mid-income" 3 "High-income", angle(45))
  xtitle("Income group") ;
# delimit cr
```

This produces the following bar chart.



```
* Use -hbar-
* First generate 0/1 temp var for counting
gen inccount = !missing(incomegrp)
graph hbar (count) inccount, over(incomegrp) over(region) ///
  ytitle("Frequency")
drop inccount
```



One drawback of `-tabulate-` is that it can only handle one *varname* at a time. To get more than one categorical variable's frequency tables, use the command `-tab1-`. Try getting frequency tables for *incomegrp* and *region*.

```
tab1 incomegrp region, missing
```

(Output is omitted).

`-tabulate-` also allows you to crosstab two categorical variables; just list two *varnames* in the order of the row variable and then the column variable. Let's, for example, cross-tabulate *incomegrp* and *region*. We want to get relative frequency both by row and column, so we will add the `[, row]` and `[, column]` options.

```
tab incomegrp region, missing row column
```

Here's your output.

```
. tab incomegrp region, missing row col
```

Income level / gnppc	Region code						Total
	Europe	N Amr	S Amr	SS Afr	Asia	M East	
low-income	17 47.22 38.64	8 22.22 53.33	5 13.89 50.00	2 5.56 66.67	3 8.33 75.00	1 2.78 50.00	36 100.00 46.15
middle-income	9 47.37 20.45	3 15.79 20.00	5 26.32 50.00	1 5.26 33.33	0 0.00 0.00	1 5.26 50.00	19 100.00 24.36
high-income	16 88.89 36.36	2 11.11 13.33	0 0.00 0.00	0 0.00 0.00	0 0.00 0.00	0 0.00 0.00	18 100.00 23.08
.	2 40.00 4.55	2 40.00 13.33	0 0.00 0.00	0 0.00 0.00	1 20.00 25.00	0 0.00 0.00	5 100.00 6.41
Total	44 56.41 100.00	15 19.23 100.00	10 12.82 100.00	3 3.85 100.00	4 5.13 100.00	2 2.56 100.00	78 100.00 100.00

The upper left box gives you the legend what four figures in each cell represent, and as we set up in the command line, the rows are for the variable *incomegrp* and the columns are for *region*.

In Stata, you can obtain a chi2 test statistics by using the `[, chi2]` option of `-tabulate-`. In our example, there are a number of cells where the expected number of observations is less than 5 (`tab incomegrp region, expected`), which does not fulfill the test assumption, and hence the chi2 test is not appropriate. But if we had more than 5 cases in each cell, then we could do this below

```
tab incomegrp region, chi2
```

to obtain your chi2 test statistics.

We just now tried exploring relationship between categorical variables (i.e., cross-tabulation, the chi2 test). What about exploring relationship between continuous variables? To display bivariate correlations matrix among continuous variables with listwise deletion, we can use `-correlate-`. To obtain pairwise correlation coefficients we can use `-pwwcorr-`. Suppose we want to examine correlations among life expectancy, per capita GNP, and population growth.

```
* Correlation -corr-  
correlate popgrowth lexp gnppc
```

```
. * Correlation -corr-  
. correlate popgrowth lexp gnppc  
(obs=73)  
  
      | popgro~h   lexp   gnppc  
-----|-----  
popgrowth | 1.0000  
lexp      | -0.5297   1.0000  
gnppc     | -0.3588   0.6307   1.0000
```

Notice that the number of observations is 73. Observations are listwise deleted.

Let's also get pairwise correlation coefficients, along with the number of observations for each pair, significance test results with $* < .05$.

```
* Pairwise correlation coeff  
pwwcorr popgrowth lexp gnppc, obs sig star(0.05)
```

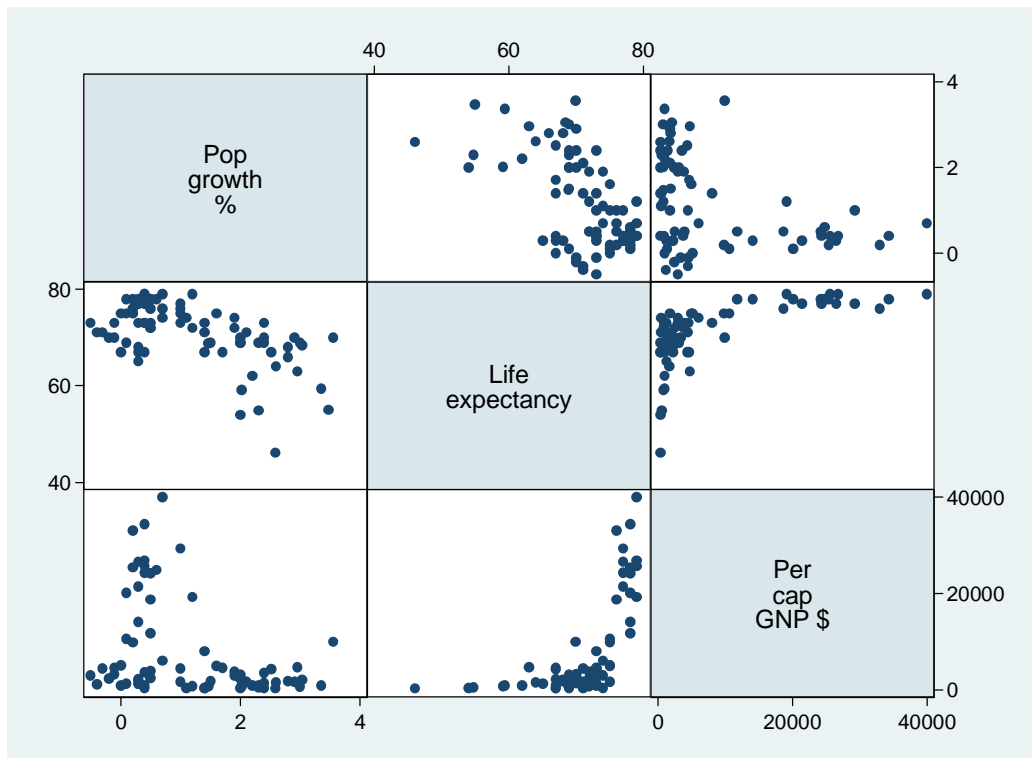
```
. * Pairwise correlation coeff  
. pwwcorr popgrowth lexp gnppc, obs sig star(0.05)  
  
      | popgro~h   lexp   gnppc  
-----|-----  
popgrowth | 1.0000  
          |      78  
lexp      | -0.5682*   1.0000  
          | 0.0000      78  
gnppc     | -0.3588*   0.6307*   1.0000  
          | 0.0018   0.0000      73  
          |      73      73      73
```

Each cell has three lines: Pearson correlation coefficients, probability, and the number of observations, from the top to the bottom.

If you want to obtain significance level information and the number of observations for each entry with observations listwise deleted, add the option `[, listwise]` to `-pwcrr-`. Then Stata drops all cases with missing values on any of the specified variables.

The command `-graph matrix-` generate a visual presentation of a correlation matrix.

```
* Scatter plots matrix  
graph matrix popgrowth lexp gnppc
```



5. Analysis Example

Now, finally, let's take a quick look at a simple and basic data analysis example by using OLS. Suppose we are interested in finding what accounts for life expectancy and decide to test for possible effects of economic development, social infrastructure, and the demographic trend.

From the literature and the descriptive exploration, we suspect that the relationship between economic development and life expectancy is curvilinear. That is, though economic development has a positive effect on life expectancy, beyond some threshold, each additional unit yields less and less impact on life expectancy. We decide to use per capita GNP as an indicator of development and test its logged term, but just in case also see how the result looks if it's specified as a squared term. As for social infrastructure, we decide to use the number of

physicians per 1,000 people as a proxy. This variable supposedly captures each society's institutional strength in educating and producing medical experts and providing health services to its members. Societies with good social infrastructure are expected to enjoy higher life expectancy. Finally, we use annual population growth to measure the demographic trend. We expect this variable has a negative relationship with life expectancy.

To run an ordinary least square model, use the command `-regress-`. The basic form is:

```
regress depvar [indepvars] [if] [in] [weight] [, options]
```

So, we are running the following two models (the difference is the functional forms of the GNP variable).

```
reg lexp lngnppc physicians popgrowth
reg lexp gnppc sqgnppc physicians popgrowth
```

Look at the outputs. From the fit statistics, you can see the more parsimonious model (i.e., the one with logged per capita GDP) performs better, so we will go with that model. Here's your output of that model.

```
. reg lexp lngnppc physicians popgrowth
```

Source	SS	df	MS			
Model	1464.52309	3	488.174364	Number of obs =	64	
Residual	886.894167	60	14.7815694	F(3, 60) =	33.03	
Total	2351.41726	63	37.3240835	Prob > F	= 0.0000	
				R-squared	= 0.6228	
				Adj R-squared	= 0.6040	
				Root MSE	= 3.8447	

lexp	Coef.	Std. Err.	t	P> t	[95% Conf. Interval]	
lngnppc	3.16008	.4027549	7.85	0.000	2.354451	3.96571
physicians	1.432497	.5263105	2.72	0.008	.3797197	2.485275
popgrowth	-.2107022	.6660593	-0.32	0.753	-1.543019	1.121615
_cons	42.71771	4.147466	10.30	0.000	34.42154	51.01387

The results give support to our hypotheses, except for population growth — which points in the right direction but is not statistically significant.

We could conduct further analytic and also diagnostics investigations. Let's get standardized beta coefficients to see the magnitude of each effect by using the option `[, beta]`. Let's also see if there is not any collinearity problem by using the postestimation command `-estat vif-`.

```
* Beta coeff and VIF
reg lexp lngnppc physicians popgrowth, beta
estat vif
```

The standardized (beta) coefficient tells you a standard deviation change in the dependent variable yielded by a one standard deviation decrease in each of your independent variable.

Standardized this way, the beta coefficients let you compare how much impact each of your predictors would have.

-estat- is a command to calculate and display statistics based on the estimated model right before (-predict-, which we will see shortly, is something similar but this command actually generates new variables). Here, we told Stata to display VIF, variance inflation factors, for our independent variables of the model we estimated right before. In the output, VIF and 1/VIF are displayed. 1/VIF is commonly called “tolerance.” Tolerance and VIF are inversely related and thus tell you the same information. Although there is no definite cut-off line, a rule of thumb is $VIF > 10$ (or tolerance 0.1) merits further investigation. In our example, the tolerance/VIF values all look okay.

```

. * Beta coeff and VIF
. reg lexp lngnppc physicians popgrowth, beta

```

Source	SS	df	MS		
Model	1464.52309	3	488.174364	Number of obs =	64
Residual	886.894167	60	14.7815694	F(3, 60) =	33.03
Total	2351.41726	63	37.3240835	Prob > F =	0.0000
				R-squared =	0.6228
				Adj R-squared =	0.6040
				Root MSE =	3.8447

lexp	Coef.	Std. Err.	t	P> t	Beta
lngnppc	3.16008	.4027549	7.85	0.000	.6805735
physicians	1.432497	.5263105	2.72	0.008	.298348
popgrowth	-.2107022	.6660593	-0.32	0.753	-.0369165
_cons	42.71771	4.147466	10.30	0.000	.

```

r; t=0.11 13:42:00
. estat vif

```

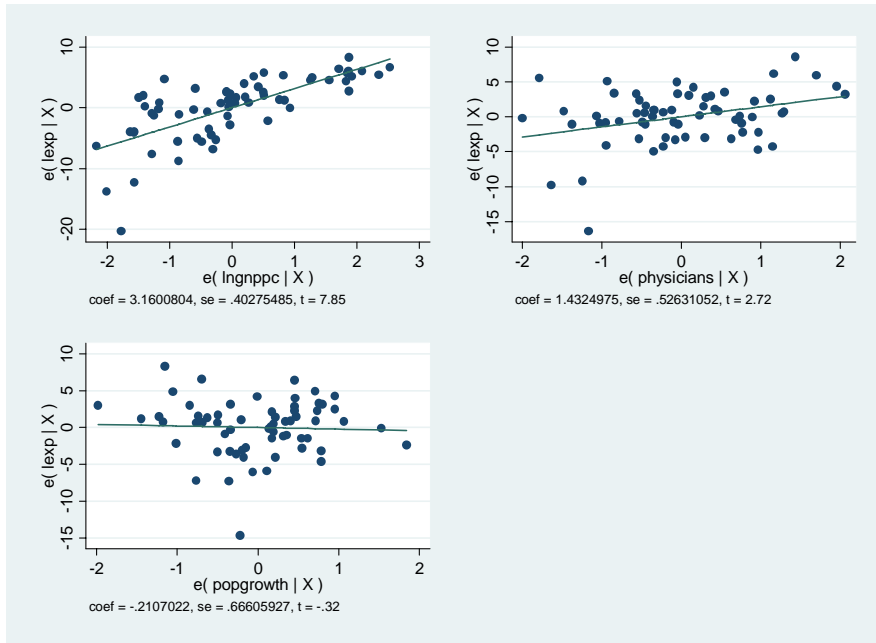
Variable	VIF	1/VIF
popgrowth	2.17	0.461595
physicians	1.91	0.523177
lngnppc	1.20	0.835517
Mean VIF	1.76	

OK, let's see how to examine residuals to check for the possible problems of unequal variance and influential cases. Here are some of the useful post estimation commands for the diagnostics.

- -avplot- or -avplots-, which give you partial regression plot for each regressor. The former followed by *varlist* so you get a partial regression plot for your specified variable.
- -rvpplot-, which gives you a residual-vs-predictor plot.
- -rvfplot-, which gives you a residual-vs-fitted value (predicted value) plot.
- -lvr2plot-, which gives you a leverage-vs-squared-residual plot.
- -dfbeta-, which gives you DFbeta influential statistics.
- -predict-, which generates predicted values, residuals, influence statistics, and other diagnostic variables.
- -estat hettest-, which gives you test statistics to check heteroskedasticity.
- -estat imtest [, white]-, which gives you test statistics to check heteroskedasticity

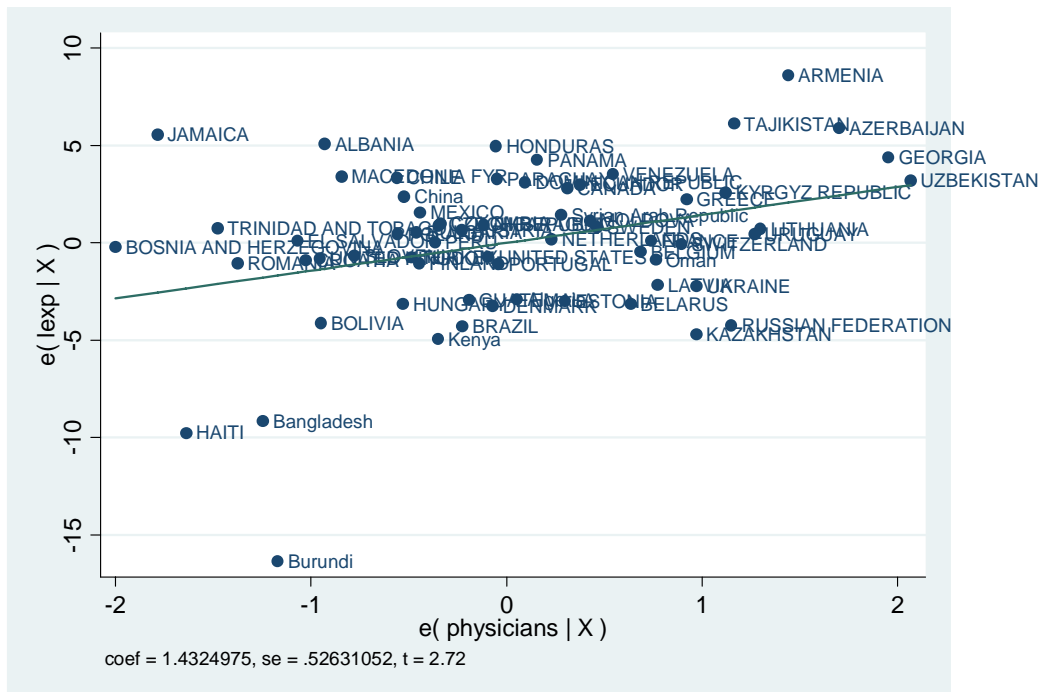
So, let's first display partial regression plots.

```
avplots
```



Examine the graphs; in some cases the plots of the variables show one case with a rather large residual.

```
avplot physician, mlabel(country) mlabsize(small)
```

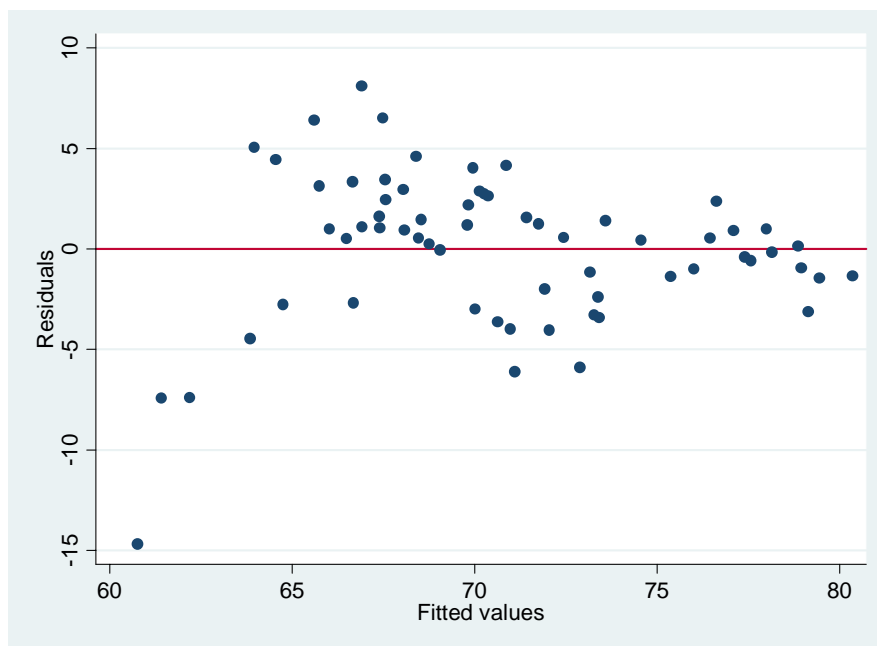


The `[, mlabel(varname)]` option is to show the marker variable in the graph (in this case, *country*). If we do this for all the predictors, we notice that Brundi is always a little off.

We then display residual versus fitted value (predicted value) plot.

```
rvfplot, yline(0)
```

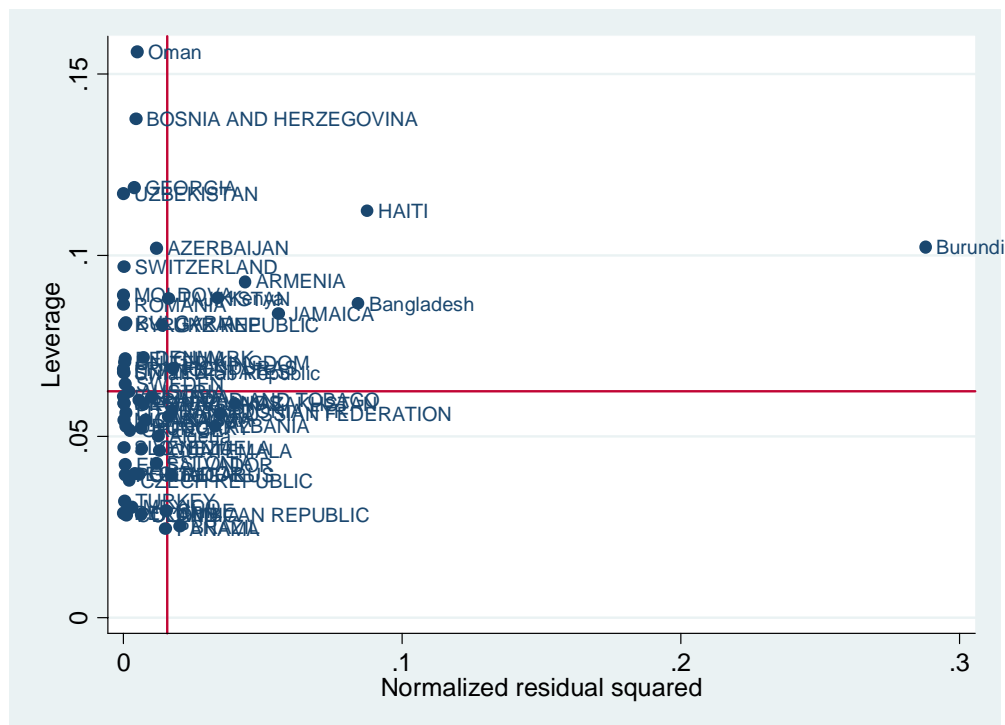
One thing that may capture our attention may be that residuals are not quite constantly distributed along the line. Also, one case again shows a large deviation from the prediction (lower left corner), which may merit a little more investigation. It turns out that this case is again Burundi (try adding `mlabel(country)` to the above code to see it).



We will do a little more investigation of possible influential cases, first by using the `-lvr2plot-` command. This command displays a plot to get a quick idea about outliers and potential influential observations.

```
lvr2plot, mlabel(country)
```

As you see in the below output, there are two reference lines in the graph. The horizontal line is the means for leverage statistics, and the vertical one is the means for the normalized residual squared. The cases which seems especially alarming are Burundi (the large residual squared), and Oman and Bosnia and Herzegovina (the large leverage values).



Let's see if the outlying observation we noticed above may have any significant impact on our regression result. Again, it is a good idea to start with some visual checks to get some rough idea. Below, we use the post estimation command `-predict-` to get the Cook's D statistics, the DFit statistics, the hat matrix (leverage), and studentized residuals and plot them for a visual check (the first two with the country number `iso3n`, and the leverage statistics with the studentized residuals).

```
* Visual check
* Cook's D
predict cd, cooks
graph twoway scatter cd iso3n, mlabel(country)

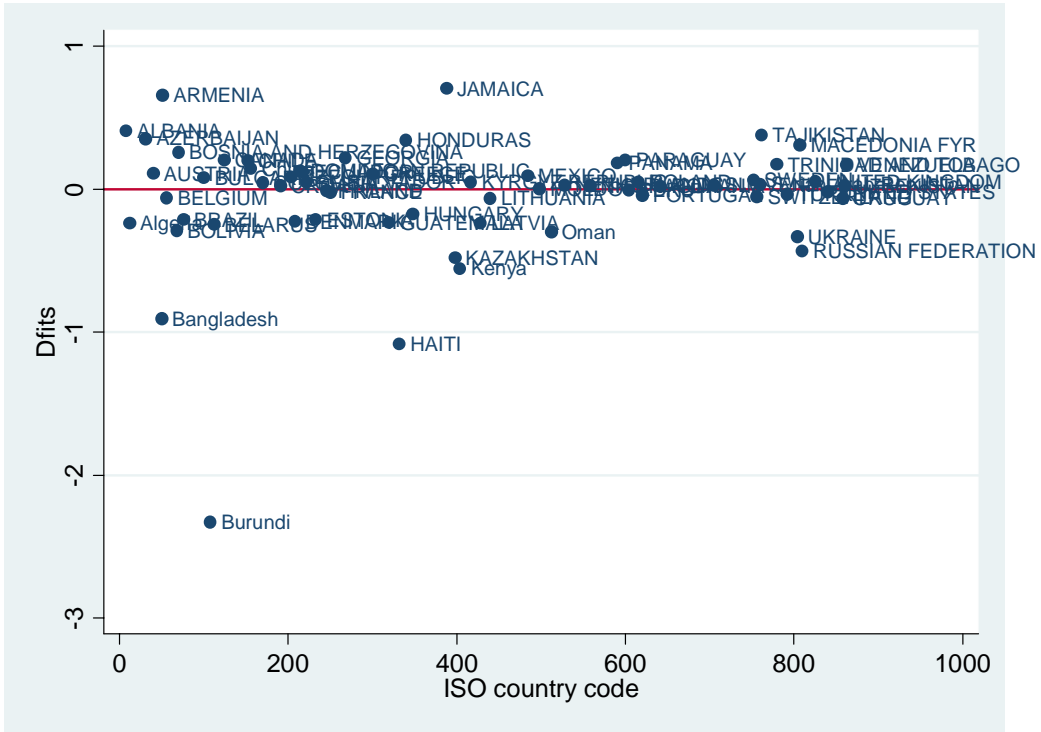
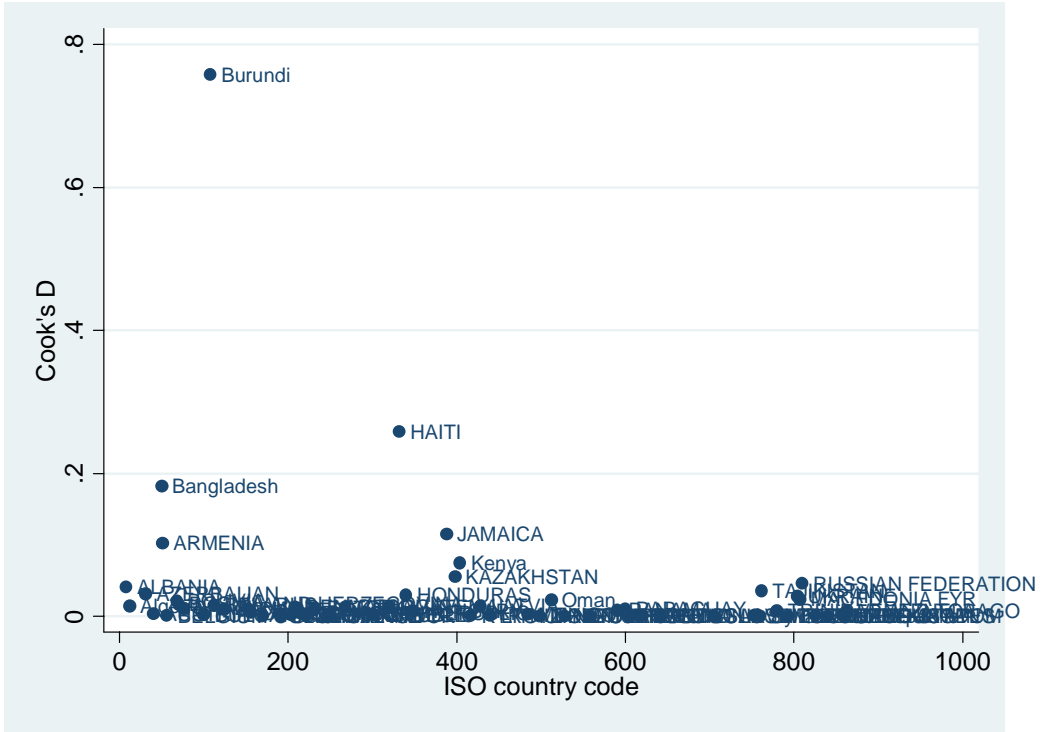
* DFit
predict dfit, dfits
graph twoway scatter dfit iso3n, yline(0) mlabel(country)

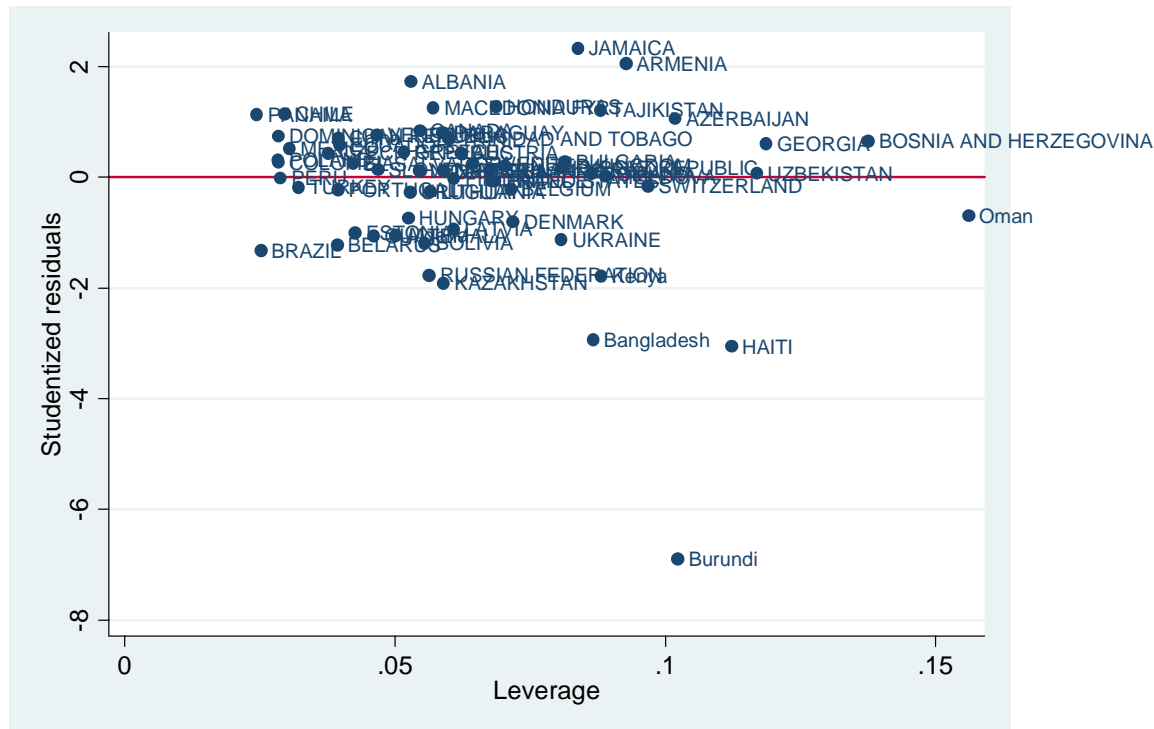
* Hat and studentized resid
predict rstu, rstudent
predict hat, hat
graph twoway scatter hat iso3n, yline(0)
graph twoway scatter rstu hat, yline(0) mlabel(country)
```

As you can see, the basic form of the command `-predict-` is:

```
predict newvar [if] [in] [, statistic]
```

A series of `-graph twoway scatter-` the following three graphs, in the order requested in the above program.





The same observations show up again and again as potential problems. We want to print cases that exceed the general rule-of-thumb thresholds of these influence statistics. Note that these are no absolute cut-off points, and because these statistics differ in their way of detecting potential issues, the final decision of observation removal depends partly on your judgment call.

```
* List cases beyond threshold
* rstudent rule of thumb: abs(rstudent) > 2
list iso3n country rstu if abs(rstu) > 2 & !missing(rstu)

* cookd rule of thumb: > n/4
list iso3n country cd if cd > 4/64 & !missing(cd), noobs

* dfit rule of thumb: > 2*sqrt(k/n) where k = # of iv's, n = # of obs
list iso3n country dfit if dfit > 2*sqrt(3/64) & !missing(dfit), noobs

* leverage rule of thumb: > (2k+2)/n
list iso3n country hat if hat > (2*3+2)/64 & !missing(hat), noobs
```

We also want to obtain the `dfbeta` statistics by using the postestimation command `-dfbeta-`, which show you how each coefficient is changed by removing the observation from the analysis. As such, this measure is created for each of the predictors, and the new variables are named `DFvarname`.

```
dfbeta
* dfbeta rule of thumb: > 2/sqrt(n)
list iso3n country DFlngnppc ///
    if DFlngnppc > 2/sqrt(64) & !missing(DFlngnppc), noobs
```

```
list iso3n country DFphysicians ///
      if DFphysicians > 2/sqrt(64) & !missing(DFphysicians), noobs
list iso3n country DFpopgrowth ///
      if DFpopgrowth > 2/sqrt(64) & !missing(DFpopgrowth), noobs
```

From the visual check of the overall pattern and the lists of the potentially problematic countries, we might particularly want to check Burundi, and perhaps Jamaica, Haiti, Bosnia and Herzegovina, and Oman.

```
* Try reestimating
reg lexp lngnppc physicians popgrowth if country != "Burundi"
```

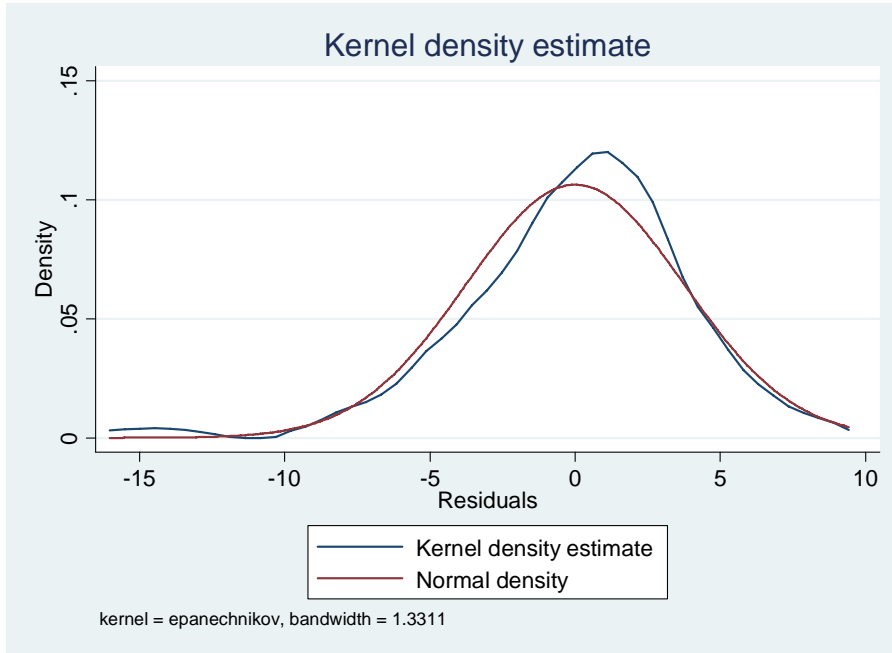
We can also try removing “Oman” in the same way. Also try different combinations of those potentially problematic cases. Examine your results (output omitted here). It seems the general conclusion remains the same; the logged per capita GNP and physicians are both statistically significant in the hypothesized direction and as such there does not seem to be any problem of influential cases so glaring as to change our theoretical argument entirely. It should be noted, however, Burundi seems to have a rather large impact on the effect size of the physician variable, whether by alone or by any combination of the questionable cases; the coefficient estimate drops by approximately 25-30% when this country is dropped from the analysis.

Let’s next check for normality of our residuals. Let’s do a quick check for the residual distribution. First, estimate the original model and obtain residuals by using -predict-.

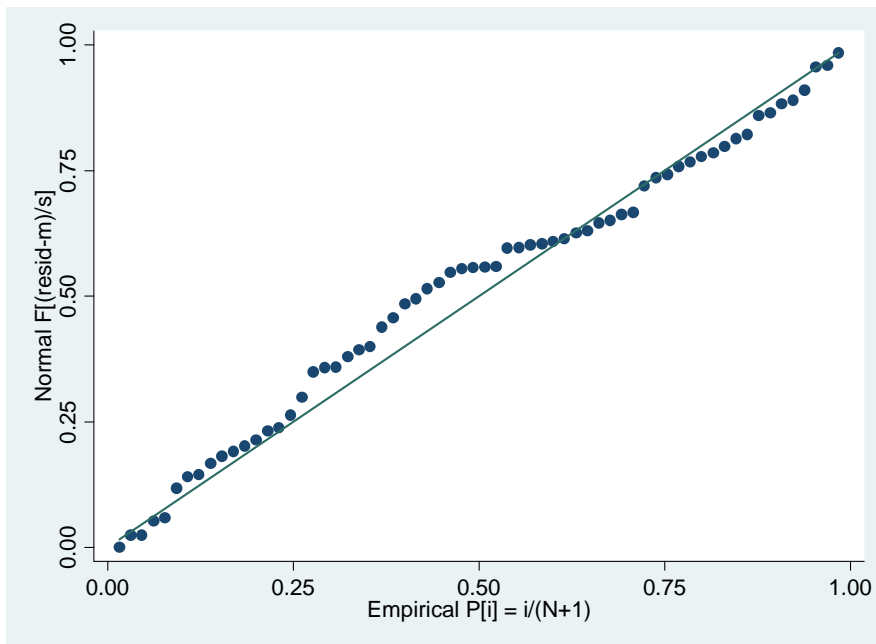
```
reg lexp lngnppc physicians popgrowth
predict resid, resid
```

Then display distributional diagnostics plot, [just like we did](#) when checking the variables’ distribution. Here, we first use the -kdensity- command with the [, normal] option to display both for visual comparison. Then we obtain a standardized normal probability (P-P) plot using the pnorm command, and then qnorm plots (the quantiles of a variable against the quantiles of a normal distribution) using -qnorm-.

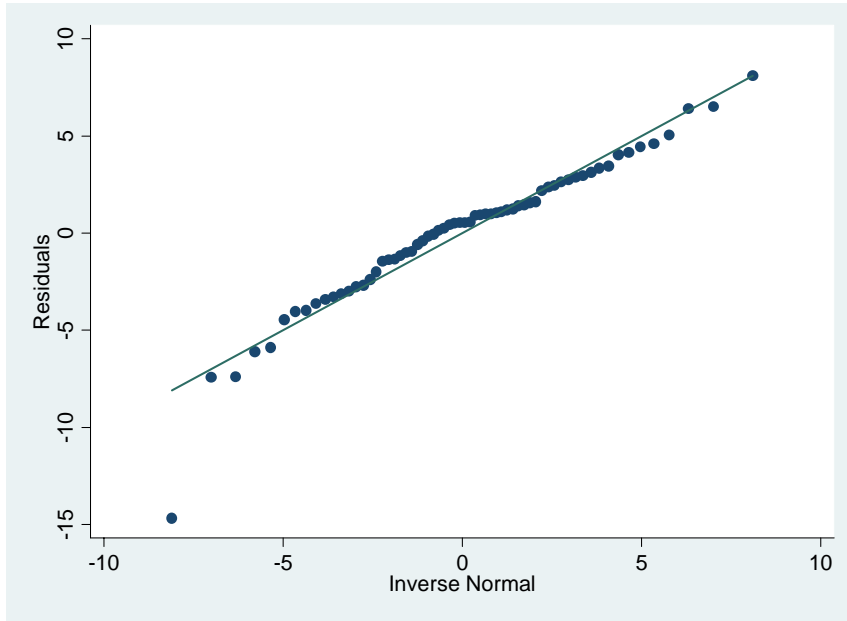
```
kdensity resid, normal
```

`pnorm resid`



`qnorm resid`



Although the residuals look roughly normally distributed, there are some deviation from normality (i.e., against the distributional reference line), in the middle in the P-P plot, and at the left tail of the qnorm. The P-P plot is sensitive to non-normality in the middle range of data and qnorm is sensitive to non-normality near the tails, so it seems our plots capture deviations around those areas.

Let's run a normality test.

```
swilk resid
```

The command swilk conducts the Shapiro-Wilk W test for normality.

```
. swilk resid
      Shapiro-Wilk W test for normal data
+-----+-----+-----+-----+-----+
Variable | Obs   | W      | V      | z      | Prob>z |
+-----+-----+-----+-----+-----+
resid   | 64   | 0.95016 | 2.853  | 2.268  | 0.01166 |
```

Shapiro-Wilk is used when the number of observations is less than 2000 (which is the case with our example), and its W statistics indicates normality when it's closer to 1.0. As you see, the test rejects the null hypothesis of normality. In this case, however, you can believe this is probably due to outliers. If you try removing one severe outlier Burundi the test fails to reject the null (output omitted).

Finally, let's learn how to check heteroskedasticity.

```
estat hettest
```

```
. estat hettest
Breusch-Pagan / Cook-Weisberg test for heteroskedasticity
Ho: Constant variance
Variables: fitted values of lexp

      chi2(1)      =      30.24
      Prob > chi2  =      0.0000
```

```
estat imtest, white
```

```
. estat imtest, white
White's test for Ho: homoskedasticity
  against Ha: unrestricted heteroskedasticity

      chi2(9)      =      31.38
      Prob > chi2  =      0.0003

Cameron & Trivedi's decomposition of IM-test
```

Source	chi2	df	p
Heteroskedasticity	31.38	9	0.0003
Skewness	8.63	3	0.0346
Kurtosis	1.07	1	0.3003
Total	41.08	13	0.0001

The first postestimation command `-estat hettest-` gives you a Breusch-Pagan / Cook-Weisberg test, and the second one `-estat imtest, white-` gives you a White's test for heteroskedasticity (the first item of Cameron & Trivedi's decomposition of IM-test is the same as the White's test). In either case, the null hypothesis of this test is homoskedasticity, and the tests reject it, meaning that we conclude the error variance is not constant.

So, we decide to take care of unequal error variance suggested by these test results. One way to address the issue is to obtain robust standard errors by using the `[, vce()]` option of `-regress-`.

```
* Robust estimators
reg lexp lngnppc physicians popgrowth, vce(robust)
```

```

. * Robust estimators
. reg lexp lngnppc physicians popgrowth, vce(robust)

```

Linear regression

Number of obs = 64
F(3, 60) = 22.09
Prob > F = 0.0000
R-squared = 0.6228
Root MSE = 3.8447

	lexp	Coef.	Robust Std. Err.	t	P> t	[95% Conf. Interval]
	lngnppc	3.16008	.4576828	6.90	0.000	2.244579 4.075582
	physicians	1.432497	.6549055	2.19	0.033	.1224914 2.742504
	popgrowth	-.2107022	.5740377	-0.37	0.715	-1.358948 .9375441
	_cons	42.71771	5.373342	7.95	0.000	31.96942 53.46599

To corrects for heteroskedasticity of unknown form, we can use [, vce(robust)] which calculates heteroskedasticity-robust standard errors of the coefficient estimators. If you have reasons to believe unequal error variance comes from some grouping unit (e.g., region), you can cluster your observations by that unit using [, vce(cluster *unit*)]. Overall, the robust standard errors are smaller than those of our OLS model, except for *popgrowth*.

That's all for this tutorial. Thanks for playing!