

*Note: "A sample session" is adapted from the [Getting Started with Stata for Windows manual](#).*

## A sample session

The dataset we will use for this session is about vintage 1978 automobiles sold in the United States.

To follow along using point-and-click, note that the menu items are given by **Menu > Menu Item > Submenu Item > etc.** To follow along using the Command window, type the commands that follow a dot (.) in the boxed listing into the small window labeled **Command**. When the structure of a command is notable, it will be pointed out as a **Syntax note**.

Start by loading the **auto** dataset, which is included with Stata. To use the menus,

1. Select **File > Example Datasets....**
2. Click on **Example datasets installed with Stata**.
3. Click on **use** on **auto.dta**.

The result of this command is threefold:

- Some output appears in the large Results window.

```
▪ . sysuse auto  
▪ (1978 Automobile Data)
```

The output consists of a command and its result. The command is bold and follows the period (.): **sysuse auto**. The output is in the standard face here and is a brief description of the dataset.

Note: If a command intrigues you, you can type **help commandname** in the Command window to find help. If you want to explore at any time, **Help > Search...** can be informative.

- The same command, **sysuse auto**, appears in the small Review window to the upper left. The Review window keeps track of commands Stata has run, successfully and unsuccessfully. The commands can then easily be rerun. See chapter 4 for more information.
- A series of variables appears in the small Variables window to the lower left.

You could have opened the dataset by typing **sysuse auto** followed by *Enter* into the Command window. Try this now. **sysuse** is a command that loads (uses) example (system) datasets. As you will see during this session, Stata commands are often simple enough that it is faster to use them directly. This will be especially true once you become familiar with the commands you use the most in your daily use of Stata.

Syntax note: in the above example **sysuse** is the Stata command, whereas **auto** is the name of a Stata data file.

SHARE    ...

**Stata 12**

[Overview: Why use Stata?](#)

[Stata/MP](#)

**Capabilities**

[Overview](#)

**Sample session**

[User-written commands](#)

**New in Stata 12**

[Supported platforms](#)

[Which Stata?](#)

[Technical support](#)

[User comments](#)



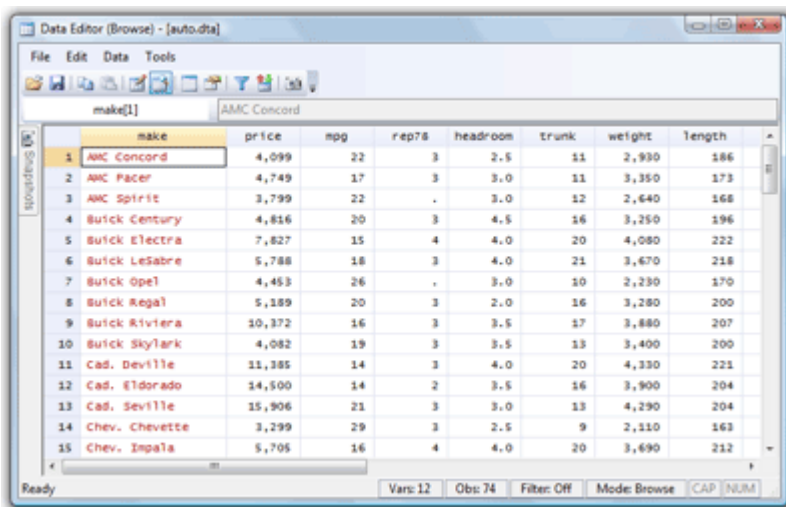
**Follow us** 

Copyright 1996–2013  
StataCorp LP | [Terms of use](#) | [Privacy](#) | [Contact us](#) |

## Simple data management

We can get a quick glimpse at the data in the **Data Browser**. Click on the Data Editor (Browse) button, ; select **Data > Data Editor (Browse)** from the menus; or type the command **browse**.

When the Data Browser window opens, you can see that Stata regards the data as one rectangular table. This is true for all Stata datasets. The columns represent *variables*, whereas the rows represent *observations*. The variables have somewhat descriptive names, whereas the observations are numbered.



The screenshot shows the Stata Data Editor (Browse) window for the file 'auto.dta'. The window displays a table with 15 observations (rows) and 8 variables (columns). The variables are: make, price, mpg, rep78, headroom, trunk, weight, and length. The 'make' variable is highlighted in yellow. The 'rep78' variable has some missing values represented by a period (.). The status bar at the bottom indicates 'Ready', 'Vars: 12', 'Obs: 74', 'Filter: Off', 'Mode: Browse', and 'CAP NUM'.

	make	price	mpg	rep78	headroom	trunk	weight	length
1	AMC Concord	4,099	22	3	2.5	11	2,930	186
2	AMC Pacer	4,749	17	3	3.0	11	3,350	173
3	AMC Spirit	3,799	22	.	3.0	12	2,640	168
4	Buick Century	4,816	20	3	4.5	16	3,250	196
5	Buick Electra	7,827	15	4	4.0	20	4,080	222
6	Buick LeSabre	5,788	18	3	4.0	21	3,670	218
7	Buick Opel	4,453	26	.	3.0	10	2,230	170
8	Buick Regal	5,189	20	3	2.0	16	3,280	200
9	Buick Riviera	10,372	16	3	3.5	17	3,880	207
10	Buick Skylark	4,082	19	3	3.5	13	3,400	200
11	Cad. Deville	11,385	14	3	4.0	20	4,330	221
12	Cad. Eldorado	14,500	14	2	3.5	16	3,900	204
13	Cad. Seville	15,906	21	3	3.0	13	4,290	204
14	Chev. Chevette	3,299	29	3	2.5	9	2,110	163
15	Chev. Impala	5,705	16	4	4.0	20	3,690	212

The data are displayed in multiple colors—at first glance it appears that the variables listed in black are numeric, whereas those that are in colors are text. This is worth investigating. Click on a cell under the **make** variable: the input box at the top displays the make of the car. Scroll to the right until you see the **foreign** variable. Click on one of its cells. Although the cell may display “Domestic”, the input box displays a 0. This shows that Stata can store categorical data as numbers but display human-readable text. This is done by what Stata calls *value labels*. Finally, under the **rep78** variable, which looks to be numeric, there are some cells containing just a period (.). As we will see, these correspond to missing values.

Syntax note: here the command is **browse**, and there are no other arguments.

Looking at the data in this fashion, though comfortable, lends little information about the dataset. It would be useful for us to get more details about what the data are and how the data are stored. Close the Data Browser by clicking its close button—while it is open, we cannot give any commands in Stata.

We can see the structure of the dataset by *describing* its contents. This can be done either by going to **Data > Describe data > Describe data in memory** in the menus and clicking **OK** or by typing **describe** in the Command window and pressing *Enter*. Regardless of which method you choose, you will get the same result:

```
. describe

Contains data from C:\Stata11\ado\base\auto.dta
   obs:              74              1978
Automobile Data
```

```

vars:          12                      13 Apr 2009 17:45
size:        3,478 (99.9% of memory free) ( dta has notes)

```

variable name	storage type	display format	value label	variable label
make	str18	%-18s		Make and Model
price	int	%8.0gc		Price
mpg	int	%8.0g		Mileage (mpg)
rep78	int	%8.0g		Repair Record 1978
headroom	float	%6.1f		Headroom (in.)
trunk	int	%8.0g		Trunk space (cu. ft.)
weight	int	%8.0gc		Weight (lbs.)
length	int	%8.0g		Length (in.)
turn	int	%8.0g		Turn Circle (ft.)
displacement	int	%8.0g		Displacement (cu. in.)
gear_ratio	float	%6.2f		Gear Ratio
foreign	byte	%8.0g	origin	Car type

Sorted by: foreign

If your listing stops short and you see a blue **--more--** at the base of the Results window, press the space bar or click on the blue **--more--** itself to allow the command to be completed.

At the top of the listing, some information is given about the dataset, such as where it is stored on disk, how much memory it occupies, and when the data were last saved. The bold **1978 Automobile Data** is the short description that appeared when the dataset was opened and is referred to as a *data label* by Stata. The phrase **\_dta has notes** informs us that there are notes attached to the dataset. We can see what notes there are by typing **notes** in the Command window:

```

. notes
_dta:
1. from Consumer Reports with permission

```

Here we see a short note about the source of the data.

Looking back at the listing from **describe**, we can see that Stata keeps track of more than just the raw data. Each variable has the following:

1. A *variable name*, which is what you call the variable when communicating with Stata.
2. A *storage type*, which is the way in which Stata stores its data. For our purposes, it is enough to know that types beginning with **str** are string or text variables, whereas all others are numeric. See [U] **12 Data**.
3. A *display format*, which controls how Stata displays the data in tables. See [U] **12.5 Formats: controlling how data are displayed**.
4. A *value label* (possibly). This is the mechanism that allows Stata to store numerical data while displaying text. See chapter 9 and [U] **12.6.3 Value labels**.
5. A *variable label*, which is what you call the variable when communicating with other people. Stata uses the variable label when making tables, as we will see.

A dataset is far more than the data it contains. It is also the information that makes the data usable by someone other than the original creator.

Although describing the data says something about the structure of the data, it says little about the data themselves. The data can be summarized by clicking **Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Summary statistics**, and clicking the **OK** button. You could also type **summarize** in the Command window and press **Enter**. The result is a table containing summary statistics about all the variables in the dataset:

```
. summarize
```

Variable	Obs	Mean	Std. Dev.	Min	Max
make	0				
price	74	6165.257	2949.496	3291	15906
mpg	74	21.2973	5.785503	12	41
rep78	69	3.405797	.9899323	1	5
headroom	74	2.993243	.8459948	1.5	5
trunk	74	13.75676	4.277404	5	23
weight	74	3019.459	777.1936	1760	4840
length	74	187.9324	22.26634	142	233
turn	74	39.64865	4.399354	31	51
displacement	74	197.2973	91.83722	79	425
gear_ratio	74	3.014865	.4562871	2.19	3.89
foreign	74	.2972973	.4601885	0	1

From this simple summary, we can learn a bit about the data. First of all, the prices are nothing like today's car prices—of course, these cars are now antiques. We can see that the gas mileages are not particularly good. Automobile aficionados can gain some feel for other characteristics.

There are two other important items here:

The variable **make** is listed as having no observations. It really has no numerical observations, because it is a string (text) variable.  
The variable **rep78** has 5 fewer observations than the other numerical variables. This implies that **rep78** has five missing values.

Although we could use the **summarize** and **describe** commands to get a bird's eye view of the dataset, Stata has a command that gives a good in-depth description of the structure, contents, and values of the variables: the **codebook** command. Either type **codebook** in the Command window and press **Enter** or navigate the menus to **Data > Describe data > Describe data contents (codebook)** and click **OK**. We get a large amount of output that is worth investigating. Look it over to see that much can be learned from this simple command. You can scroll back in the Results window to see earlier results, if need be. We'll focus on the output for **make**, **rep78**, and **foreign**.

To start our investigation, we would like to run the **codebook** command on just one variable, say, **make**. We can do this via menus or the command line, as usual. To get the **codebook** output for **make** via the menus, start by navigating as before to **Data > Describe data > Describe data contents (codebook)**. When the dialog appears, there are multiple ways to tell Stata to consider only the **make** variable:

- We could type **make** into the *Variables* field.
- The *Variables* field is actually a combobox control that accepts variable names. Clicking the button to the right of the *Variables* field displays a list of the variables from the current dataset. Selecting a variable from the list will, in this case, enter the variable name into the edit field.

A much easier solution is to type **codebook make** in the Command window and then press **Enter**. The result is informative:

```

. codebook make
make
Make and Model
type:  string (str18), but longest is
str17
unique values:  74
missing "":  0/74

examples:  "Cad. Deville"
           "Dodge Magnum"
           "Merc. XR-7"
           "Pont. Catalina"

warning:  variable has embedded blanks

```

The first line of the output tells us the variable name (**make**) and the variable label (**Make and Model**). The variable is stored as a string (which is another way of saying “text”) with a maximum length of 18 characters, though a size of only 17 characters would be enough. All the values are unique, so, if need be, **make** could be used as an identifier for the observations—something that is often useful when putting together data from multiple sources or when trying to weed out errors from the dataset. There are no missing values, but there are blanks within the makes. This latter fact could be useful if we were expecting **make** to be a single-word string variable.

Syntax note: telling the **codebook** command to run on the **make** variable is an example of using a *varlist* in Stata's syntax.

Looking at the **foreign** variable can teach us about value labels. We would like to look at the codebook output for this variable, and, on the basis of our latest experience, it would be easy to type **codebook foreign** into the Command window to get the following output:

```

. codebook foreign
foreign
Car type
type:  numeric (byte)
label:  origin
range:  [0,1]
units:  1
unique values:  2
missing .:  0/74

tabulation:  Freq.    Numeric  Label
              52         0  Domestic
              22         1  Foreign

```

We can glean that **foreign** is an indicator variable, because its only values are 0 and 1. The variable has a value label that displays “Domestic” instead of 0 and “Foreign” instead of 1. There are two advantages of storing the data in this form:

- Storing the variable as a byte takes less memory, because each observation uses 1 byte instead of the 8 bytes needed to store “Domestic”. This is important in large datasets. See [U] **12.2.2 Numeric storage types**.
- As an indicator variable, it is easy to incorporate into statistical models. See [U] **25.2 Using indicator variables in estimation**.

Finally, we can learn a little about a poorly labeled variable with missing values by looking at the **rep78** variable. Typing **codebook rep78** into the Command window yields

```
. codebook rep78
rep78
Repair Record 1978

          type:  numeric (int)

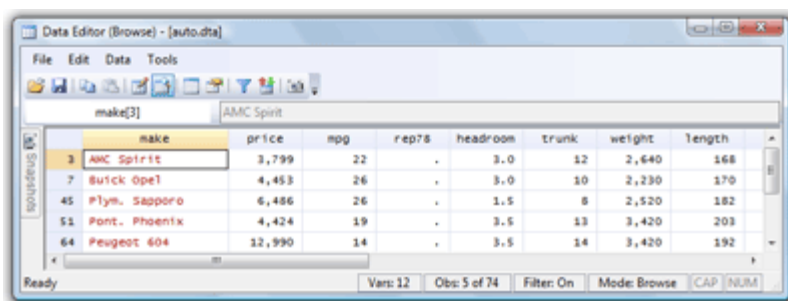
          range:  [1,5]

units:    1
          unique values:  5
missing .:  5/74

          tabulation:  Freq.    Value
                        2        1
                        8        2
                       30        3
                       18        4
                       11        5
                        5         .
```

**rep78** appears to be a categorical variable, but, because of lack of documentation, we do not know what the numbers mean. (To see how we would label the values, see chap. 10.) This variable has five missing values, meaning that there are 5 observations for which the repair record is not recorded. We can use the Data Browser to investigate these 5 observations—and we will do this using the Command window only. (Doing so is much simpler.) If you recall from earlier, the command brought up by clicking the Data Browser button was **browse**. We would like to browse only those observations for which **rep78** is missing, so we could type

```
. browse if missing(rep78)
```



From this, we see that the . entries are indeed missing values—though other missing values are allowable. See [U] **12.2.1 Missing values**. Close the Data Browser after you are satisfied with this statement.

Syntax note: using the *if* *qualifier* above is what allowed us to look at a subset of the observations.

Looking through the data lends us no clues about why these particular data are missing. We decide to check the source of the data to see if the missing values were originally missing or if they were omitted in error. Listing the makes of the cars whose repair records are missing will be all we need, because we saw earlier that the values of **make**

are unique. This can be done via the menus and a dialog.

1. Select **Data > Describe data > List data**.
2. Click the button to the right of the *Variables* field to show the variable names.
3. Click **make** to enter it into the *Variables* field.
4. Click the **by/if/in** tab in the dialog.
5. Type **missing(rep78)** into the *If: (expression)* box.
6. Click **Submit**. Stata executes the proper command, but the dialog box remains open. **Submit** is useful when experimenting, exploring, or building complex commands. We will use primarily **Submit** in the examples. You may click **OK** in its place if you like.

The same ends could be achieved by typing **list make if missing(rep78)**. The latter is easier, once you know that the command **list** is used for listing observations. In any case, here is the output:

```
. list make if missing(rep78)
```

	make
3.	AMC Spirit
7.	Buick Opel
45.	Plym. Sapporo
51.	Pont. Phoenix
64.	Peugeot 604

We go to the original reference and find that the data were truly missing and cannot be resurrected. See chapter 11 for more information about all that can be done with the **list** command.

Syntax note: this command uses two new concepts for Stata commands—the **if** qualifier and the **missing()** function. The **if** qualifier restricts the observations the command runs to only those observations for which the expression is true. See [U] 11.1.3 **if exp**. The **missing()** function tests each observation to see if it is missing. See [U] 13.3 **Functions**.

Now that we have a good idea about the underlying dataset, we can investigate the data themselves.

## Descriptive statistics

We saw above that the **summarize** command gave brief summary statistics about all the variables. Suppose now that we became interested in the prices while summarizing the data, because they seemed fantastically low (it was 1978, after all). To get an in-depth look at the **price** variable, we can use the menus and a dialog:

1. Select **Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Summary statistics**.
2. Enter or select **price** in the *Variables* field.
3. Select **Display additional statistics**.
4. Click **Submit**.

Syntax note: as can be seen from the Results window, typing **summarize price, detail** will get the same result. The portion after the comma contains *options* for Stata commands; hence, **detail** is an example of an option.

```
. summarize price, detail
```

Price				
Percentiles	Smallest			
1%	3291	3291		
5%	3748	3299		
10%	3895	3667	Obs	74
25%	4195	3748	Sum of Wgt.	74
50%	5006.5	Mean	6165.257	
	Largest	Std. Dev.	2949.496	
75%	6342	13466		
90%	1385	13594	Variance	8699526
95%	13466	14500	Skewness	1.653434
99%	15906	15906	Kurtosis	4.819188

From the output, we can see that the median price of the cars in the dataset is only about \$5,006! We could also see that the four most expensive cars are all priced between \$13,400 and \$16,000. If we wished to browse the cars that are the most expensive (and gain some experience with Stata's command syntax), we could use an **if** qualifier,

```
. browse if price > 13000
```

and see from the Data Browser that the expensive cars are two Cadillacs and two Lincolns, which have low gas mileage and are fairly heavy:

	make	price	mpg	rep78	headroom	trunk	weight	length
12	Cad. Eldorado	14,500	14	2	3.5	16	3,900	204
13	Cad. Seville	15,906	21	3	3.0	13	4,290	204
27	Linc. Mark V	13,594	12	3	2.5	18	4,720	230
28	Linc. Versailles	13,466	14	3	3.5	15	3,630	201

We now decide to turn our attention to foreign cars and repairs because, as we glanced through the data, it appeared that the foreign cars had better repair records. Let's start by looking at the proportion of foreign cars in the dataset along with the proportion of cars with each type of repair record. We can do this with one-way tables. The table for **foreign** cars can be done via menus and a dialog starting with **Statistics > Summaries, tables, and tests > Tables > One-way tables** and then choosing the variable **foreign** in the *Categorical variable* field. Clicking **Submit** yields

```
. tabulate foreign
```

Car type	Freq.	Percent	Cum.
Domestic	52	70.27	70.27
Foreign	22	29.73	100.00
Total	74	100.00	

We see that roughly 70% of the cars in the dataset are domestic, whereas 30% are foreign made. The value labels are used to make the table so that the output is nicely readable.



Syntax note: we also see that this one-way table could be made by using the **tabulate** command together with a one variable: **foreign**.

Making a one-way table for the repair records is simple—it will be simpler if done via the Command window. Typing **tabulate rep78** yields

```
. tabulate rep78
```

Repair Record 1978	Freq.	Percent	Cum.
1	2	2.90	2.90
2	8	11.59	14.49
3	30	43.48	57.97
4	18	26.09	84.06
5	11	15.94	100.00
Total	69	100.00	

We can see that most cars have repair records of 3 and above, though the lack of value labels makes us unsure what a “3” means. The five missing values are indirectly evident, because the total number of observations listed is 69 rather than 74.

These two one-way tables do not help us compare the repair records of foreign and domestic cars. A two-way table would help greatly, which we can get by using the menus and a dialog:

1. Select **Statistics > Summaries, tables, and tests > Tables > Two-way tables with measures of association**.
2. Choose **rep78** as the row variable.
3. Choose **foreign** as the column variable.
4. It would be nice to have the percentages within the **foreign** variable, so check the **Within-row relative frequencies** checkbox.
5. Click **Submit**.

Here is the resulting output.

```
. tabulate rep78 foreign, row
```

Key			
<i>frequency</i>			
<i>row percentage</i>			

Repair Record 1978	Car type		Total
	Domestic	Foreign	
1	2 100.00	0 0.00	2 100.00
2	8 100.00	0 0.00	8 100.00
3	27 90.00	3 10.00	30 100.00
4	9 50.00	9 50.00	18 100.00
5	2 18.18	9 81.82	11 100.00
Total	48 69.57	21 30.43	69 100.00

The output indicates that foreign cars are generally much better than domestic cars when it comes to repairs. If you like, you could repeat the previous dialog and try some of the hypothesis tests available from the dialog. We will abstain.

Syntax note: we see that typing the command **tabulate rep78 foreign, row** would have given us the same table. Thus using **tabulate** with two variables yielded a two-way table. It makes sense that **row** is an option. We went out of our way to check it in the dialog, so we changed the behavior of the command from its default.

Continuing our exploratory tour of the data, we would like to compare gas mileages between foreign and domestic cars, starting by looking at the summary statistics for each group by itself. We know that a direct way to do this would be to summarize **mpg** for each of the two values of **foreign**, by using **if** qualifiers:

<b>. summarize mpg if foreign==0</b>					
	Obs	Mean	Std. Dev.	Min	
Variable	Max				
mpg	52	19.82692	4.743297	12	34
<b>. summarize mpg if foreign==1</b>					
	Obs	Mean	Std. Dev.	Min	
Variable	Max				
mpg	22	24.77273	6.611187	14	41

It appears that foreign cars get somewhat better gas mileage—we will test this soon.

Syntax note: we needed to use a double equal sign (==) for testing equality. This could be familiar to you if you have programmed before. If it is unfamiliar, it is a common source of errors when initially using Stata. Thinking of equality as “really equal” can cut down on typing errors.

There are two other methods we could have used to produce these summary statistics. These methods are worth knowing, because they are less error prone. The first method duplicates the concept of what we just did by exploiting Stata's ability run a command on each of a series of nonoverlapping subsets of the dataset. To use the menus and a dialog, do the following:

1. Select **Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Summary statistics** and click the **Reset** button.
2. Select **mpg** in the *Variables* field.
3. Select *Standard display* option (if it is not already selected).
4. Select the **by/if/in** tab.
5. Check the *Repeat command by groups* checkbox.
6. Select or type **foreign** for the grouping variable.
7. **Submit** the command.

You can see that the results match those from above. They have a better appearance because the value labels are used rather than the numerical values. This method is more appealing because the results were produced without knowing the possible values of the grouping variable ahead of time.

<b>. by foreign, sort: summarize mpg</b>					
-> foreign = Domestic					
	Obs	Mean	Std. Dev.	Min	
Variable	Max				

mpg	52	19.82692	4.743297	12	
	34				
-> foreign = Foreign					
Variable	Obs	Mean	Std. Dev.	Min	Max
mpg	22	24.77273	6.611187	14	41

There is something different about the equivalent command that appears above: it contains a *prefix command* called a **by** prefix. The **by** prefix has its own option, namely, **sort**, to ensure that like members are adjacent to each other before being summarized. The **by** prefix command is important for understanding data manipulation and working with subpopulations within Stata. Store this example away, and consult [U] **11.1.2 by varlist**: and [U] **27.2 The by construct** for more information. Stata has other prefix commands for specialized treatment of commands, as explained in [U] **11.1.10 Prefix commands**.

The third method for tabulating the differences in gas mileage across the cars' origins involves thinking about the structure of desired output. We need a one-way table of automobile types (foreign versus domestic) within which we see information about gas mileages. Looking through the menus yields the menu item **Statistics > Summaries, tables, and tests > Tables > One/two-way table of summary statistics**. Selecting this, entering **foreign** for *Variable 1* and **mpg** for the *Summarize variable*, and submitting the command yields a nice table:

```
. tabulate foreign, summarize(mpg)
```

Car type	Summary of Mileage (mpg)		
	Mean	Std. Dev.	Freq.
Domestic	19.826923	4.7432972	52
Foreign	24.772727	6.6111869	22
Total	21.297297	5.7855032	74

The equivalent command is evidently **tabulate foreign, summarize(mpg)**.

Syntax note: this is a one-way table, so **tabulate** uses one variable. The variable being summarized is passed to the **tabulate** command via an option.

## A simple hypothesis test

We would like to run a hypothesis test for the difference in the mean gas mileages. Under the menus, **Statistics > Summaries, tables, and tests > Classical tests of hypotheses > Two-group mean-comparison test** leads to the proper dialog. Enter **mpg** for the *Variable name* and **foreign** for the *Group variable name*, and then **Submit** the dialog. The results are

```
. ttest mpg, by(foreign)
```

Two-sample t test with equal variances						
Group	Obs	Mean	Std.Err.	Std.Dev.	[95% Conf. Interval]	
Domestic	52	19.82692	.657777	4.743297	18.50638	21.14747
Foreign	22	24.77273	1.40951	6.611187	21.84149	27.70396
combined	74	21.2973	.6725511	5.785503	19.9569	22.63769
diff		-4.945804	1.362162	-7.661225	-2.230384	
diff = mean(Domestic) - mean(Foreign)      t = -3.6308						
Ho: diff = 0                                      degrees of freedom = 72						

```

      Ha: diff < 0      Ha: diff != 0      Ha: diff > 0
Pr(T<t) = 0.0003 Pr(|T|>|t|) = 0.0005 Pr(T > t) = 0.9997

```

From this, we could conclude that the mean gas mileage for foreign cars is different from that of domestic cars (though we really ought to have tested this before snooping through the data). We can also conclude that the command, **ttest mpg, by(foreign)** is easy enough to remember. Feel free to experiment with unequal variances, various approximations to the number of degrees of freedom, and the like.

Syntax note: the **by()** option used here is not the same as the **by** prefix command used earlier. Although it has a similar conceptual meaning, its usage is different because it is a particular option for the **ttest** command.

## Descriptive statistics, correlation matrices

We now change our focus from exploring categorical relationships to exploring numerical relationships: we would like to know if there is a correlation between miles per gallon and weight. We select **Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Correlations and covariances** in the menus. Entering **mpg** and **weight**, either by clicking or typing, and then submitting the command yields

```

. correlate mpg weight
(obs=74)

```

	mpg	weight
mpg	1.0000	
weight	-0.8072	1.0000

The equivalent command for this is natural: **correlate mpg weight**. There is a negative correlation, which is not surprising, because heavier cars should be harder to push about.

We could see how the correlation compares for foreign and domestic cars by using our knowledge of how the **by** prefix works. We can reuse the *correlate* dialog or use the menus as before if the dialog is closed. Click the **by/if/in** tab, check the *Repeat command by groups* checkbox, and enter the **foreign** variable to define the groups. As done above on page 19, a simple **by foreign, sort:** prefix in front of our previous command would work, too:

```

. by foreign, sort: correlate mpg weight
-> foreign = Domestic (obs=52)

```

	mpg	weight
mpg	1.0000	
weight	-0.8759	1.0000

```

-> foreign = Foreign (obs=22)

```

	mpg	weight
mpg	1.0000	
weight	-0.6829	1.0000

We see from this that the correlation is not as strong among the foreign cars.

Syntax note: although we used the **correlate** command to look at the correlation of two variables, Stata can make correlation matrices for an arbitrary number of variables:

```

. correlate mpg weight length turn displacement
(obs=74)

```

	mpg	weight	length	turn	displa~t
mpg	1.0000				
weight	-0.8072	1.0000			
length	-0.7958	0.9460	1.0000		
turn	-0.7192	0.8574	0.8643	1.0000	
displacement	-0.7056	0.8949	0.8351	0.7768	1.0000

This can be useful, for example, when investigating collinearity among predictor variables. In fact, simply typing **correlate** will yield the complete correlation matrix.

## Graphing data

We have found several things in our investigations so far: We know the average MPG of domestic and foreign cars differs. We have learned domestic and foreign cars differ in other ways as well, such as in frequency-of-repair record. We found a negative correlation of MPG and weight—as we would expect—but the correlation appears stronger for domestic cars.

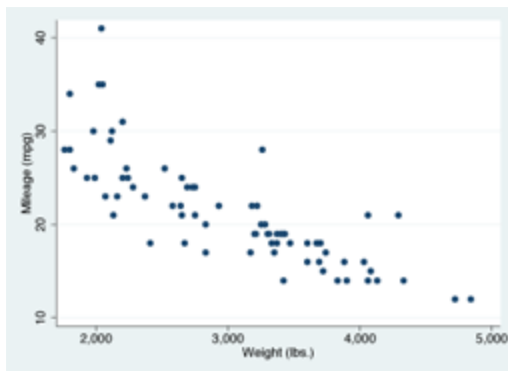
We would now like to examine, with an eye toward modeling, the relationship between MPG and weight, starting with a graph. We can start with a scatterplot of **mpg** against **weight**. The command for this is simple: **scatter mpg weight**. Using the menus requires a few steps because the graphs in Stata can be customized heavily.

1. Select **Graphics > Twoway graph (scatter, line, etc.)**.
2. Click the **Create...** button.
3. Select the *Basic plots* radio button (if it is not already selected).
4. Select *Scatter* as the basic plot type (if it is not already selected).
5. Select **mpg** as the *Y variable* and **weight** as the *X variable*.
6. Click the **Submit** button.

The Results window shows the command issued from the menu:

```
. twoway (scatter mpg weight)
```

The command issued when the dialog was submitted is a bit more complex than the command suggested above. There is good reason for this: the more complex structure allows combining and overlaying graphs, as we will soon see. In any case, the graph that appears is



We see the negative correlation in the graph, though the relationship appears to be nonlinear.

**Note:** when you draw a graph, the Graph window appears, probably covering up your Results window. Click on the main Stata window to get the Results window back on top. Want to see the graph again? Click the **Graph** button. See chapter 14 for more information about the **Graph** button.

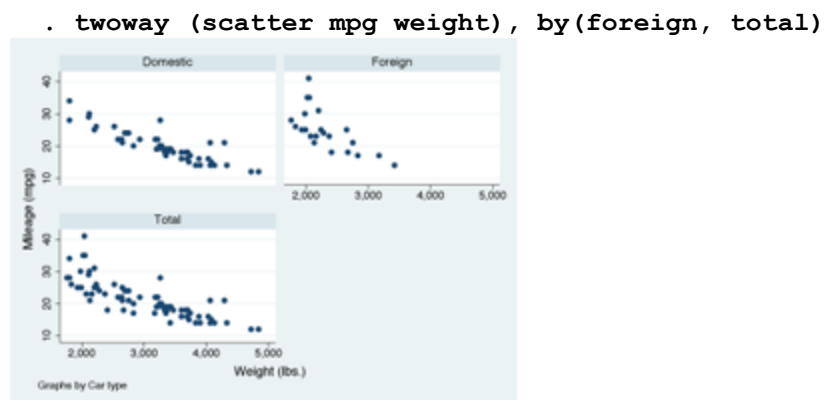
We would now like to see how the different correlations for foreign and domestic cars are manifested in scatterplots. It would be nice to see a scatterplot for each type of car, along with a scatterplot for all the data.

Syntax note: because we are looking at subgroups, this looks like it is a job for **by**. We want one graph—think about whether it should be a **by()** option or a **by** prefix.

Start as before:

1. Select **Graphics > Twoway graph (scatter, line, etc.)** from the menus.
2. If *Plot 1* is displayed under the *Plot definitions* and **(scatter mpg weight)** appears below the *Plot definitions* box, select it and skip to step 4.
3. Go through the process to create the graph on the previous page.
4. Click the **By** tab.
5. Check the *Draw subgraphs for unique values of variables* checkbox.
6. Enter **foreign** as the *Variable*.
7. Check the *Add a graph with totals* checkbox.
8. Click the **Submit** button.

The command and the associated graph are



The graphs show that the relationship is nonlinear for both types of cars.

Syntax note: this is another use of a **by()** option. If you had used a **by** prefix, two separate graphs would have been generated.

## Model fitting: Linear regression

After looking at the graphs, we would like to fit a regression model that predicts MPG from the weight and type of the car. From the graphs, the relationship is nonlinear, so we will try modeling MPG as a quadratic in weight. Also from the graphs, we judge the relationship to be different for domestic and foreign cars. We will include an indicator (dummy) variable for foreign and evaluate afterward whether this adequately describes the difference. Thus we will fit the model

$$\text{mpg} = \beta_0 + \beta_1 \text{weight} + \beta_2 \text{weight}^2 + \beta_3 \text{foreign} + \epsilon$$

**foreign** is already an indicator (0/1) variable, but we need to create the weight-squared variable. This can be done via the menus, but here using the command line is simpler:

```
. gen wtsq = weight^2
```

Now that we have all the variables we need, we can run a linear regression. We'll use the menus and see that the command is also simple. To use the menus, select **Statistics > Linear models and related > Linear regression**. In the resulting dialog, choose **mpg** as the dependent variable and **weight**, **wtsq**, and **foreign** as the independent variables. **Submit** the command. Here is the equivalent simple **regress** command and the resulting analysis-of-variance table.

```
. regress mpg weight wtsq foreign
```

				Number of obs = 74		
				F( 3, 70) = 52.25		
				Prob > F = 0.0000		
				R-squared = 0.6913		
				Adj R-squared 0.6781		
				Root MSE =3.2827		
Source	SS	df	MS			
Model	1689.15372	3	63.05124			
Residual	754.30574	70	0.7757963			
Total	2443.4594673	33.4720474				

mpg	Coef.	Std.Err.	t	P> t	[95% Conf. Interval]	
weight	-.0165729	.0039692	-4.18	0.000	-.0244892	-0.0086567
wtsq	1.59e-06	6.25e-07	2.55	0.013	3.45e-07	2.84e-06
foreign	-2.2035	1.059246	-2.08	0.041	-4.3161	-.0909002
cons	56.53884	6.197383	9.12	0.000	44.17855	68.89913

The results look encouraging, so we will plot the predicted values on top of the scatterplots for each of the types of cars. To do this, we need the predicted, or fitted, values. This can be done via the menus, but doing it by hand is simple enough. We will create a new variable **mpghat**:

```
. predict mpghat
(option xb assumed; fitted values)
```

The output from this command is simply a notification. Go over to the Variables window and scroll to the bottom to confirm that there is now an **mpghat** variable. If you try this command when **mpghat** already exists, Stata will refuse to overwrite your data:

```
. predict mpghat
mpghat already defined
r(110);
```


The **predict** command, when used after a regression, is called a *postestimation command*. As specified, it creates a new variable called **mpghat** equal to

$$-.0165729\text{weight} + 1.59 \times 10^{-6}\text{wtsq} - 2.2035\text{foreign} + 56.53884$$

For careful model fitting, there are several features available to you after estimation—one is calculating predicted values. Be sure to read [U] **20 Estimation and**

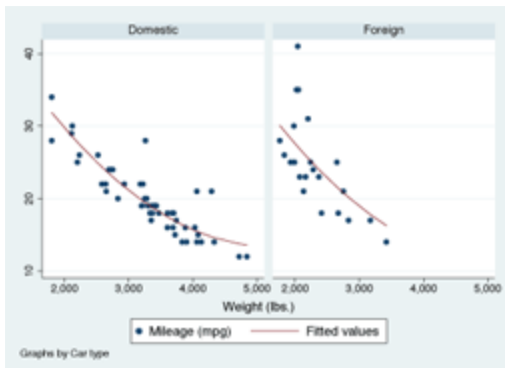
### postestimation commands.

We can now graph the data and the predicted curve to evaluate the fit on the foreign and domestic data separately to determine if our shift parameter is adequate. We can draw both graphs together. Using the menus and a dialog, do the following:

1. Select **Graphics > Twoway graph (scatter, line, etc.)** from the menus.
2. If there are any plots listed, click the **Reset** button, .
3. Create the graph for **mpg** versus **weight**.
  - a. Click the **Create...** button.
  - b. Be sure that *Basic plots* and *Scatter* are selected.
  - c. Select **mpg** as the *Y variable* and **weight** as the *variable*.
  - d. Click **Accept**.
4. Create the graph showing **mpg** versus **weight**.
  - a. Click the **Create...** button.
  - b. Be sure that *Basic plots* and *Line* are selected.
  - c. Select **mpg** as the *Y variable* and **weight** as the *X variable*.
  - d. Check the *Sort on x variable* box. Doing so ensures that the lines connect from smallest to largest **weight** values, instead of the order in which the data happen to be.
  - e. Click **Accept**.
5. Show two plots, one each for domestic and foreign cars, on the same graph.
  - a. Click the **By** tab.
  - b. Check the *Draw subgraphs for unique values of variables*.
  - c. Enter **foreign** in the *Variables* field.
6. Click the **Submit** button.

Here are the resulting command and graph.

```
. twoway (scatter mpg weight) (line mpghat weight, sort), by(foreign)
```



Here we can see the reason for enclosing the separate **scatter** and **line** commands in parentheses: they can then be overlaid by submitting them together. The fit of the plots looks good and is cause for initial excitement. So much excitement, in fact, that we decide to print the graph and show it to an engineering friend. We print the graph, being careful to print the graph and not all our results: **File > Print** from the Graph window menu bar.

When we show our graph to our engineering friend, she is concerned. “No,” she says. “It should take twice as much energy to move 2,000 pounds 1 mile compared with moving 1,000 pounds the same distance; therefore, it should consume twice as much gasoline. Miles per gallon is not a quadratic in weight; gallons per mile is a linear



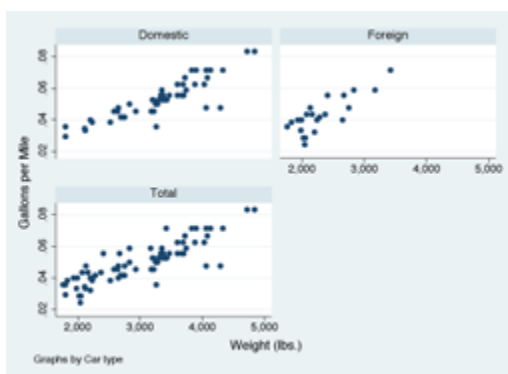
function of weight. Don't you remember any physics?"

We try out what she says. We need to generate a gallons-per-mile variable and make a scatterplot. Here are the commands we would need—note their similarity to commands issued earlier in the session. There is one new command: the **label variable** command, which allows us to give the **gpm** variable a *variable label* so that the graph is labeled nicely.

```
. generate gpm = 1/mpg

. label variable gpm "Gallons per mile"

. twoway (scatter gpm weight), by(foreign, total)
```



Sadly satisfied that the engineer is indeed correct, we rerun the regression:

```
. regress gpm weight foreign
```

Source	SS	df	MS	F( 2, 71)	Prob > F
Model	.009117618	2	.004558809	113.97	0.0000
Residual	.00284001	71	.00004		0.7625
Total	.011957628	73	.000163803		0.7558

Number of obs = 74  
R-squared = 0.7625  
Adj R-squared = 0.7558  
Root MSE = .00632

	gpm	Coef.	Std.Err	t	P> t	[95% Conf. Interval]
weight		.0000163	1.18e-06	13.74	0.000	.0000139 .0000186
foreign		.0062205	.0019974	3.11	0.003	.0022379 .0102032
_cons		-.0007348	.0040199	-0.18	0.855	-.0087504 .0072807

We find that although foreign cars had better gas mileage than domestic cars in 1978, it was because they were so light. In fact, according to our model a foreign car with the same weight as a domestic car would use an additional 1/160 gallon per mile driven. With this, we are satisfied with our analysis.

## Commands versus menus

In this chapter you have seen that Stata can operate either via menu choices and dialog boxes or via the Command window. As you become more familiar with Stata, you will

find that the Command window is typically much faster for often-used commands, whereas the menus and dialogs are faster when building up complex commands such as graphs.

One of Stata's great strengths is the consistency of its *command syntax*. Most of Stata's commands share the following syntax, where square brackets mean that something is optional and a *varlist* is a list of variables.

```
[prefix : ] command [varlist] [if] [in] [weight] [ , options ]
```

Some general rules:

- Most commands accept prefix commands that modify their behavior; see [U] **11.1.10 Prefix commands** for details. One of the more common prefix commands is **by**.
- If an optional *varlist* is not specified, all the variables are used.
- *if* and *in* restrict the observations on which the command is run.
- *options* modify what the command does.
- Each command's syntax is found in the online help and the reference manuals.
- Stata's command syntax includes more than we have shown you here, but this should get you started. For more information, see [U] **11 Language syntax** and **help language**.

We saw examples using all the pieces of this except for the **in** qualifier and the **weight** clause. The syntax for all commands can be found in the online help along with examples—see chapter 6 for more information. The explanation of the syntax can be found online using **help language**. The consistent syntax makes it straightforward to learn new commands and to read others' commands when examining an analysis.

Here is an example of reading the syntax diagram by using the **summarize** command from earlier in this chapter. The syntax diagram for **summarize** is typical:

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

This means that


<i>command</i> by itself is valid:	<b>summarize</b>
<i>command</i> followed by a <i>varlist</i> (variable list) is valid:	<b>summarize mpg</b>
	<b>summarize mpg weight</b>
<i>command</i> with <i>if</i> (with or without a <i>varlist</i> ) is valid:	<b>summarize if mpg&gt;20</b>
	<b>summarize mpg weight if mpg&gt;20</b>

and so on.

You can learn about **summarize** in [R] **summarize**, or select **Help > Stata Command...** and enter **summarize**, or type **help summarize** in the Command window.

## Keeping track of your work

It would have been useful if we had made a log of what we did so that we could conveniently look back at interesting results or track any changes that were made. You will learn to do this in chapter 16. Your logs will contain commands and their output—another reason to learn command syntax, so that you can remember what you've done.

To make a log file that keeps track of everything appearing in the Results window, click the button that looks like a lab notebook, . Choose a place to store your log file, and give it a name, just as you would any other document. The log file will save everything that appears in the Results window from the time you start a log file to the time that you close it.

## Conclusion

This chapter introduced you to Stata's capabilities. You should now read and work through the rest of this manual. Once you are done here, you can read the *User's Guide*.

See **New in Stata 12** for more about what was added in Stata Release 12.

## 1.2 Using Stata Effectively

While it is fun to type commands interactively and see the results straightaway, serious work requires that you save your results and keep track of the commands that you have used, so that you can document your work and reproduce it later if needed. Here are some practical recommendations.

### 1.2.1 Create a Project Directory

Stata reads and saves data from the working directory, usually `C:\DATA`, unless you specify otherwise. You can change directory using the command `cd [drive:]directory_name`, and print the (name of the) working directory using `pwd`, type `help cd` for details. I recommend that you create a separate directory for each course or research project you are involved in, and start your Stata session by changing to that directory.

Stata understands nested directory structures and doesn't care if you use `\` or `/` to separate directories. Versions 9 and later also understand the double slash used in Windows to refer to a computer, so you can `cd \\opr\shares\research\myProject` to access a shared project folder. An alternative approach, which also works in earlier versions, is to use Windows explorer to assign a drive letter to the project folder, for example assign P: to `\\opr\shares\research\myProject` and then in Stata use `cd p:`. Alternatively, you may assign R: to `\\opr\shares\research` and then use `cd R:\myProject`, a more convenient solution if you work in several projects.

Stata has other commands for interacting with the operating system, including `mkdir` to create a directory, `dir` to list the names of the files in a directory, `type` to list their contents, `copy` to copy files, and `erase` to delete a file. You can (and probably should) do these tasks using the operating system directly, but the Stata commands may come handy if you want to write a program to perform repetitive tasks.

### 1.2.2 Open a Log File

So far all our output has gone to the *Results* window, where it can be viewed but eventually disappears. (You can control how far you can scroll back, type `help scrollbufsize` to learn more.) To keep a *permanent* record of your results, however, you should `log` your session. When you

open a log, Stata writes all results to both the *Results* window and to the file you specify. To open a log file use the command

`log using filename, text replace`

where *filename* is the name of your log file. Note the use of two recommended options: `text` and `replace`.

By default the log is written using SMCL, Stata Markup and Control Language (pronounced "smicle"), which provides some formatting facilities but can only be viewed using Stata's *Viewer*. Fortunately, there is a `text` option to create logs in plain text (ASCII) format, which can be viewed in an editor such as Notepad or a word processor such as Word. (An alternative is to create your log in SMCL and then use the `translate` command to convert it to plain text, postscript, or even PDF if you are a Mac user, type `help translate` to learn more about this option.)

The `replace` option specifies that the file is to be overwritten if it already exists. This will often be the case if (like me) you need to run your commands several times to get them right. In fact, if an earlier run has failed it is likely that you have a log file open, in which case the `log` command will fail. The solution is to close any open logs using the `log close` command. The problem with this solution is that it will not work if there is no log open! The way out of the catch 22 is to use

`capture log close`

The `capture` keyword tells Stata to run the command that follows and ignore any errors. Use judiciously!

### 1.2.3 Always Use a Do File

A do file is just a set of Stata commands typed in a plain text file. You can use Stata's own built-in *do-file Editor*, which has the great advantage that you can run your program directly from the editor by clicking on the run icon or selecting `Tools|Run` from the menu. You can also select just a few commands and run them by selecting `Tools|Run Selection` in the menu. To access Stata's do editor use Ctrl-9 in version 12 (Ctrl-8 in earlier versions) or select `Window|Do-file Editor|New Do-file Editor` in the menu system.

Alternatively, you can use an editor such as Notepad. Save the file using extension `.do` and then execute it using the command `do filename`. For a thorough discussion of alternative text editors see <http://fmwww.bc.edu/repec/bocode/t/textEditors.html>, a page maintained by Nicholas J. Cox, of the University of Durham.

You could even use a word processor such as Word, but you would have to remember to save the file in plain text format, not in Word document format. Also, you may find Word's insistence on capitalizing the first word on each line annoying when you are trying to type Stata commands that must be in lowercase. You can, of course, turn auto-correct off. But it's a lot easier to just use a plain-text editor.

### 1.2.4 Use Comments and Annotations

Code that looks obvious to you may not be so obvious to a co-worker, or even to you a few months later. It is always a good idea to annotate your do files with explanatory comments that provide the gist of what you are trying to do.

In the Stata command window you can *start* a line with a `*` to indicate that it is a comment, not a command. This can be useful to annotate your output.

In a do file you can also use two other types of comments: `//` and `/* */`

`//` is used to indicate that everything that *follows* to the end of the line is a comment and should be ignored by Stata. For example you could write

```
gen one = 1 // this will serve as a constant in the model
```

`/* */` is used to indicate that all the text *between* the opening `/*` and the closing `*/`, which may be a few characters or may span several lines, is a comment to be ignored by Stata. This type of comment can be used anywhere, even in the middle of a line, and is sometimes used to "comment out" code.

There is a third type of comment used to break very long lines, as explained in the next subsection. Type [help comments](#) to learn more about comments.

It is always a good idea to start every do file with comments that include at least a title, the name of the programmer who wrote the file, and the date. Assumptions about required files should also be noted.

## 1.2.5 Continuation Lines

When you are typing on the command window a command can be as long as needed. In a do-file you will probably want to break long commands into lines to improve readability.

To indicate to Stata that a command continues on the next line you use `///`, which says everything else to the end of the line is a comment *and* the command itself continues on the next line. For example you could write

```
graph twoway (scatter lexp loggnppc) ///  
              (lfit lexp loggnppc)
```

Old hands might write

```
graph twoway (scatter lexp loggnppc) /*  
*/          (lfit lexp loggnppc)
```

which "comments out" the end of the line.

An alternative is to tell Stata to use a semi-colon instead of the carriage return at the end of the line to mark the end of a command, using `#delimit ;`, as in this example:

```
#delimit ;  
graph twoway (scatter lexp loggnppc)  
              (lfit lexp loggnppc) ;
```

Now all commands need to terminate with a semi-colon. To return to using `carriage return` as the delimiter use

```
#delimit cr
```

The delimiter can only be changed in do files. But then you always use do files, right?

## 1.2.6 A Sample Do File

Here's a simple do file that can reproduce all the results in our [Quick Tour](#), and illustrates the syntax highlighting introduced in Stata's do file editor in version 11. The file doesn't have many comments because this page has all the details. Following the listing we comment on a couple of lines that require explanation.

```
// A Quick Tour of Stata
// German Rodriguez - Fall 2011

version 12
clear
capture log close
log using QuickTour, text replace

display 2+2
display 2 * ttail(20,2.1)

// load sample data and inspect
sysuse lifeexp
desc
summarize lexp gnppc
list country gnppc if missing(gnppc)

graph twoway scatter lexp gnppc, ///
    title(Life Expectancy and GNP) xtitle(GNP per capita)
graph export scatter.png, width(400) replace // save the graph in PNG format
gen loggnppc = log(gnppc)
regress lexp loggnppc

predict plexp

graph twoway (scatter lexp loggnppc) (lfit lexp loggnppc) ///
    , title(Life Expectancy and GNP) xtitle(log GNP per capita)
graph export fit.png, width(400) replace

list country lexp plexp if lexp < 55, clean
list gnppc loggnppc lexp plexp if country == "United States", clean
log close
// make sure you hit enter for the last line
```

We start the do file by specifying the version of Stata we are using, in this case 12. This helps ensure that future versions of Stata will continue to interpret the commands correctly, even if Stata has changed, see [help version](#) for details. (The previous version of this file read version 11, and I could have left that in place to run under version control; the results would be the same because none of the commands used in this quick tour has changed.)

The `clear` statement deletes the data currently held in memory and any value labels you might have. The reason we need that is that if we had to rerun the program the `sysuse` command would fail because we already have a dataset in memory and it has not been saved. An alternative with the same effect is to type `sysuse lifeexp, clear`. (Stata keeps other objects in memory as well, including saved results, scalars and matrices, although we haven't had occasion to use these yet. Typing `clear all` removes these objects from memory, ensuring that you start with a completely clean slate. See [help clear](#) for more information. Usually, however, all you need to do is clear the data.)