

Powers, D. A. and Y. Xie. 2000. *Statistical Methods for Categorical Data Analysis*. San Diego: Academic Press. This book considers all the models discussed in our book, with the exception of count models, and includes loglinear models and models for event history analysis.

Train, K. 2003. *Discrete Choice Methods with Simulation*. Cambridge: Cambridge University Press. This is an outstanding review of models for a wide range of models for discrete choice and includes details on new methods of estimation using simulation.

## 2 Introduction to Stata

This book is about fitting and interpreting regression models using Stata, and to earn our pay we must get to these tasks quickly. With that in mind, this chapter is a relatively concise introduction to Stata 9 for those with little or no familiarity with the package. Experienced Stata users can skip this chapter, although a quick reading might be useful. We focus on teaching the reader what is necessary to work through the examples later in the book and to develop good working techniques for using Stata for data analysis. By no means are the discussions exhaustive; often we show you either our favorite approach or the approach that we think is simplest. One of the great things about Stata is that there are usually several ways to accomplish the same thing. If you find a better way than what we have shown you, use it!

*You cannot learn how to use Stata simply by reading.* Accordingly, we strongly encourage you to try the commands as we introduce them. We have also included a tutorial in section 2.17 that covers many of the basics of using Stata. Indeed, you might want to try the tutorial first and then read our detailed discussions of the commands.

Although people who are new to Stata should find this chapter sufficient for understanding the rest of the book, if you want further instruction, look at the resources listed in section 2.3. We also assume that you know how to load Stata on the computer you are using and that you are familiar with your computer's operating system. By this, we mean that you should be comfortable copying and renaming files, working with subdirectories, closing and resizing windows, selecting options with menus and dialog boxes, and so on.

(Continued on next page)

## 2.1 The Stata interface

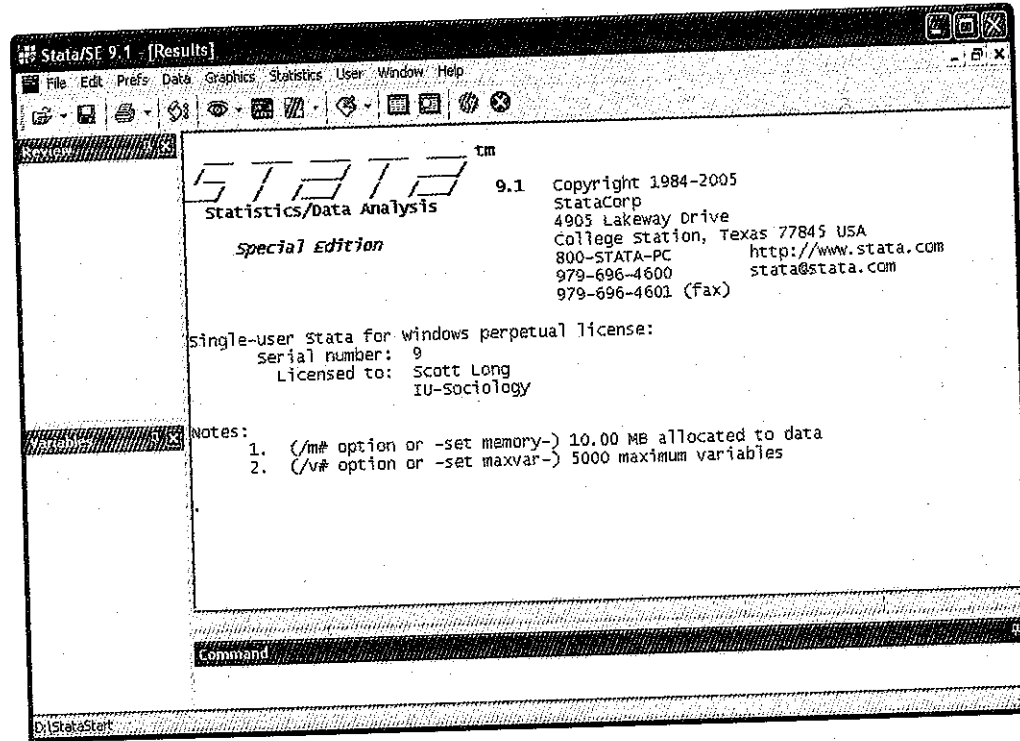


Figure 2.1: Opening screen in Stata for Windows.

When you launch Stata, you will see a screen with several smaller windows located within the larger Stata window, as shown in figure 2.1. This screen shot is for Windows using the default windowing preferences. If the defaults have been changed or you are running Stata under Unix or MacOS, your screen will look slightly different.<sup>1</sup> Figure 2.2 shows what Stata looks like after several commands have been entered and data have been loaded into memory. In both figures, four windows are shown.

1. Our screen shots and descriptions are based on Stata for Windows. Please refer to the books *Getting Started with Stata for Macintosh* or *Getting Started with Stata for Unix* for examples of the screens for those operating systems.

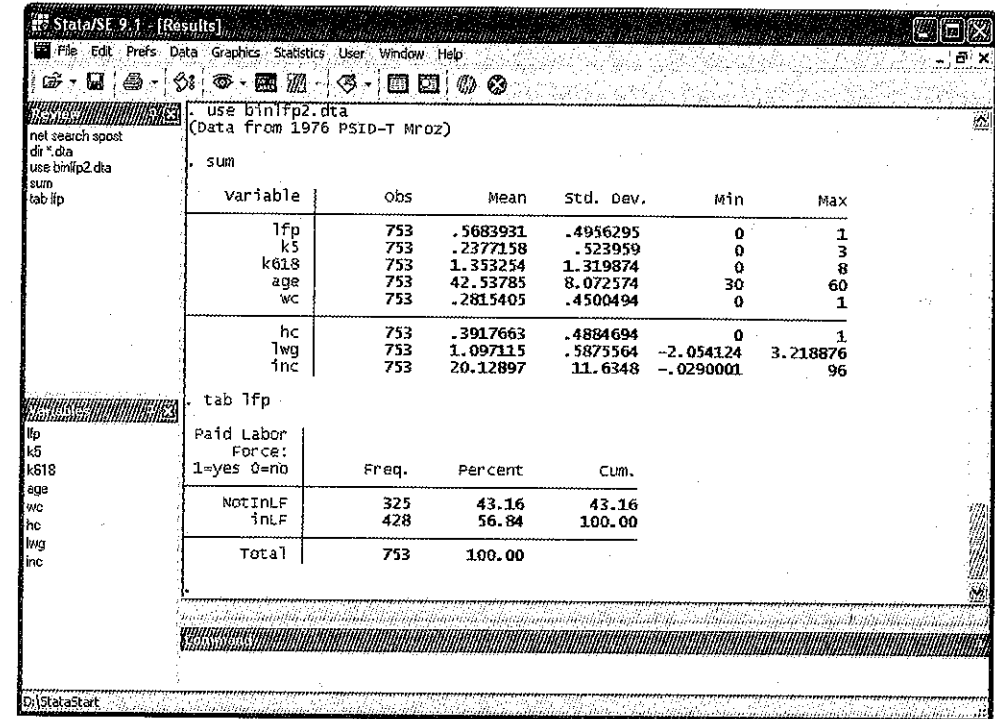


Figure 2.2: Example of Stata windows after several commands have been entered and data have been loaded.

**The Command window** is where you type commands that are executed when you press Enter. As you type commands, you can edit them at any time *before* pressing Enter. Pressing PageUp brings the most recently used command into the Command window; pressing PageUp again retrieves the command before that; and so on. Once a command has been retrieved to the Command window, you can edit it and press Enter to run the modified command.

**The Results window** contains output from the commands entered in the Command window. The Results window also echoes the command that generated the output, where the commands are preceded by a "." as shown in figure 2.2. The scroll bar on the right lets you scroll back through output that is no longer on the screen. You can also use the scroll wheel on the mouse to scroll back and forth. Within the window, you can highlight text and right-click the mouse to see options for copying the highlighted text. In Stata for Windows and Stata for Macintosh, the Copy Table option copies the selected lines to the clipboard, and Copy Table as HTML allows you to copy the selected text as an HTML table (see page 94 for more information). Right-clicking also gives you the option to print the contents of the window. Only the most recent output is available this way; earlier lines are


lost unless you have saved them to a log file (discussed below). Details on setting the size of the scrollbar buffer are given below.

**The Review window** lists the commands that have been entered from the Command window. If you click on a command in this window, it is pasted into the Command window, where you can edit it before execution of the command. If you double-click on a command in the Review window, it is pasted into the Command window and immediately executed.

**The Variables window** lists the names of variables that are in memory, including those loaded from disk files and those created with Stata commands. If you click on a name, it is pasted into the Command window.

The Command and Results windows illustrate the important point that Stata has its origins in a command-based system. This means that you tell Stata what to do by typing commands that consist of one line of text and then pressing Enter.<sup>2</sup> Beginning with Stata 8, there is a complete graphical user interface (GUI) for accessing all nonprogramming commands. At the risk of seeming old-fashioned, however, we still prefer the command-based interface. Although it can take longer to learn, once you learn it, you should find it much faster to use. If you currently prefer using pulldown menus, stick with us, and you will likely change your mind.

Because Stata 8 and later have a complete GUI, you can do almost anything in Stata by pointing and clicking. Some of the most important tasks can be performed by clicking on icons on the toolbar at the top of the screen. Although we on occasion mention the use of these icons, for the most part we stick with text commands. Indeed, even if you do click on an icon or issue a command from a dialog box, Stata shows you how this could be done with a text command. For example, if you click on the **Data**

**Browser** button,  Stata opens a spreadsheet for examining your data. Meanwhile, “. browse” is written to the Results window. This means that instead of clicking the icon, you could have typed browse. Overall, not only is the range of things you can do with menus limited, but almost everything you can do with the mouse can also be done with commands, often more efficiently. Therefore, and because it makes things much easier to automate later, we describe things mainly in terms of commands. Even so, we encourage you to explore the tasks available through menus and the toolbar and to use them when preferred.

### Changing the scrollbar buffer size

How far back you can scroll in the Results window is controlled by the command

```
set scrollbar #
```

2. For now, we consider entering only one command at a time, but in section 2.9 we show you how to run a series of commands at once using “do-files”.

where  $10,000 \leq \# \leq 500,000$ . By default, the buffer size is 32,000 bytes. When you change the size of the scroll buffer using `set scrollbar`, the change will take effect the next time you launch Stata. Unless memory is a problem, we set the buffer to its maximum.

### Changing the display of variable names in the Variables window

The Variables window displays both the names of variables in memory and their variable labels. By default, 32 columns are reserved for the name of the variable. The maximum number of characters to display for variable names is controlled by the command

```
set varlabelpos #
```

where  $8 \leq \# \leq 32$ . By default, the size is 32. In figure 2.2, none of the variable labels are shown since the 32 columns take up the width of the window. If you use short variable names, it is useful to `set varlabelpos` to a smaller number so that you can see the variable labels.

---

**Tip: Changing defaults** We prefer a larger scroll buffer and less space for variable names. We could enter the command `set varlabelpos 14` at the start of each Stata session, but it is easier to add the commands to `profile.do`, a file that is automatically run each time Stata begins. We show you how to do this in chapter 9.

---

## 2.2 Abbreviations


Commands and variable names can often be abbreviated. For variable names, the rule is easy: *any variable name can be abbreviated to the shortest string that uniquely identifies it*. For example, if there are no other variables in memory that begin with `a`, the variable `age` can be abbreviated as `a` or `ag`. If you have the variables `income` and `income2` in your data, neither of these variable names can be abbreviated.

There is no general rule for abbreviating commands, but as one would expect, typically the most common and general command names can be abbreviated. For example, four of the most often used commands are `summarize`, `tabulate`, `generate`, and `regress`, and these can be abbreviated as `su`, `ta`, `g`, and `reg`, respectively. From now on, when we introduce a Stata command that can be abbreviated, we underline the shortest abbreviation (e.g., `generate`). But, although very short abbreviations are easy to type, they can be confusing when you are getting started. Accordingly, when we use abbreviations, we stick with at least three-letter abbreviations.

## 2.3 How to get help

### 2.3.1 Online help

If you find our description of a command incomplete, or if we use a command that is not explained, you can use Stata to find more information. Use the `help` command when you know the name of the command and want to find out more about it. For example, `help regress` tells you about the `regress` command. `search` is used when you do not know the name of the command or where something is documented. `help` and `search` are typed in the Command window, with results for `help` displayed in a Viewer window and results for `search` returned to the Results window. You can also open the Viewer

by clicking on . At the top of the Viewer, there is a line labeled Command, where you can type commands, such as `help`. The Viewer is particularly useful for reading long help files.

`help` lists a shortened version of the documentation in the manual for any command. You can even type `help help` for help on using `help`. The output from `help` often makes reference to other commands, which are shown in blue. (Anything in the Results window that is in blue type is a clickable link.) Here clicking on a command name in blue type is the same as typing `help` for that command.

`search` is so useful for tracking down information that we encourage you to type `help search` and read more about this powerful command, because here we provide only a few details. By default, `search word ...` searches Stata's index on your local machine and lists the entries that match your query. For example, `search gen` lists information on `generate` but also links to many related commands. Or, if you want to run a truncated regression model but cannot remember the name of the command, you could try `search truncated` to get information on commands related to truncation. The resulting commands are listed in blue, so you can click on the name; details appear in the Viewer. If you want to extend your search to the Internet, add the option `all`; for example, `search truncated, all`. When you search the Internet, you get information from the Stata web site, including FAQs (frequently asked questions), articles that have appeared in the *Stata Journal* (often abbreviated SJ), and information about user-written commands that are not part of official Stata. For example, when you installed the `SPost` commands, you might have used `search spost, all` to find the links for installation. To get a better idea of how the `all` option works, try `search truncated, all` and compare the results with those from `search truncated`. If you always want to include a search of the Internet, you can set `searchdefault all`, permanently and `search` will automatically search the Internet each time you use the command. Or, you can leave the `search` default to search your local machine and use the `findit` command, which is equivalent to `search word ..., all`.

---

**Tip: Help with error messages** Error messages in Stata can be terse and sometimes confusing. Whereas the error message is printed in red, errors also have a *return code* (e.g., `r(199)`) listed in blue. Clicking on the return code provides a more detailed description of the error.

---

### 2.3.2 Manuals

The Stata manuals are extensive, and it is worth taking an hour to browse them to get an idea of the many features in Stata. In general, we find that learning how to read the manuals (and use the help system) is more efficient than asking someone else, and it allows you to save your questions for the really hard stuff. For those new to Stata, we recommend the *Getting Started* manual (which is specific to your platform) and the first part of the *User's Guide*. As you become more acquainted with Stata, the reference manuals will become increasingly valuable for detailed information about commands, including a discussion of the statistical theory related to the commands and references for further reading.

### 2.3.3 Other resources

The *User's Guide* also discusses more sources of information about Stata. Most importantly, the Stata web site (<http://www.stata.com>) contains many useful resources, including links to tutorials and an extensive FAQ section that discusses both introductory and advanced topics. You can also get information on the NetCourses offered by Stata, which are 4- to 7-week courses offered over the Internet. Another excellent set of online resources is provided by UCLA's Academic Technology Services at <http://www.ats.ucla.edu/stat/stata/>.

There is also a Statalist listserver that is not part of StataCorp, although many programmers and statisticians from StataCorp participate. This list is a wonderful resource for information on Stata and statistics. You can submit questions and usually receive answers very quickly. Monitoring the listserver is also a quick way to pick up insights from Stata veterans. For details on joining the list, visit <http://www.stata.com/statalist/>.

## 2.4 The working directory

The *working directory* is the default directory for any file operations such as using data, saving data, or logging output. If you type `cd` or `pwd` in the Command window, Stata displays the name of the current working directory. To load a data file stored in the working directory, you just type use *filename* (e.g., use `bin1fp2`). If a file is not in the working directory, you must specify the full path (e.g., use `d:\spostdata\examples\bin1fp2`).

At the beginning of each Stata session, we like to change our working directory to the directory where we plan to work, since this is easier than repeatedly entering the path name for the directory. For example, typing `cd d:\spostdata` changes the working directory to `d:\spostdata`. If the directory name includes spaces, you must put the path in quotation marks (e.g., `cd "d:\my work\"`).

You can list the files in your working directory by typing `dir` or `ls`, which are two names for the same command. With this command, you can use the `*` wildcard. For example, `dir *.dta` lists all files with the extension `.dta`.

## 2.5 Stata file types

Stata uses and creates many types of files, which are distinguished by extensions at the end of the filename. Some of the extensions used by Stata are

<code>.ado</code>	Programs that add commands to Stata, such as the SPost commands.
<code>.class</code>	Files that define classes in the Stata class system.
<code>.dlg</code>	Programs that define the appearance and functionality of dialog boxes.
<code>.do</code>	Batch files that execute a set of Stata commands.
<code>.dta</code>	Data files in Stata's format.
<code>.emf</code>	Graphs saved as Windows Enhanced Metafiles.
<code>.gph</code>	Graphs saved in Stata's proprietary format.
<code>.hlp</code>	The text displayed when you use the <code>help</code> command. For example, <code>fitstat.hlp</code> has help for <code>fitstat</code> .
<code>.log</code>	Output saved as plain text by the <code>log using</code> command.
<code>.mata</code>	Original source code for programs written in Mata.
<code>.mlib</code>	Libraries of programs written in Mata.
<code>.smcl</code>	Output saved in the SMCL format by the <code>log using</code> command.

The most important of these for a new user are the `.smcl`, `.log`, `.dta`, and `.do` files, which we will now discuss.

## 2.6 Saving output to log files

Stata does not automatically save the output from your commands. To save your output to print or examine later, you must open a *log file*. Once a log file is opened, both the commands and the output they generate are saved. Because the commands are recorded, you can tell exactly how the results were obtained. The syntax for the `log` command is

```
log using filename [, append replace [smcl|text] name(logname) ]
```

By default, the log file is saved to your working directory. You can save it to a different directory by typing the full path (e.g., `log using d:\project\mylog, replace`).

### 2.6.1 Closing a log file

#### Options

`append` means that if the file exists, new output should be added to the end of the existing file.

`replace` indicates that you want to replace the log file if it already exists. For example, `log using mylog` creates the file `mylog.smcl`. If this file already exists, Stata generates an error message. So, you could use `log using mylog, replace`, and the existing file would be overwritten by the new output.

`smcl` and `text` specify the format in which the log is to be recorded.

`smcl` is the default option that requests that the log be written using the Stata Markup and Control Language (SMCL) with the file suffix `.smcl`. SMCL files contain special codes that add solid horizontal and vertical lines, bold and italic typefaces, and hyperlinks to the Results window. The disadvantage of SMCL is that the special features can be viewed only within Stata. If you open a SMCL file in a text editor, your results will appear amidst a jumble of special codes.

`text` specifies that the log should be saved as plain text (ASCII), which is the preferred format for loading the log into a text editor for printing. Instead of adding the `text` option, such as `log using mywork, text`, you can specify plain text by including the `.log` extension (for example, `log using mywork.log`).

If you open multiple log files, you may choose a different format for each file.

`name(logname)` specifies an optional name for the log file to be used while it is open. This lets you have multiple log files open, each with a different name. You can then close, temporarily suspend, or resume them individually.

---

**Tip: Plain text logs by default** We prefer plain text for output rather than SMCL. Typing `set logtype text` at the beginning of a Stata session makes plain text the default for log files for the current session. Typing `set logtype text, permanently` makes plain text the default for future sessions.

If you have a question and would like to send us a log file, make sure that it is in text format rather than in SMCL. Before you send anything, please check [http://www.indiana.edu/~jslsoc/spost\\_help.htm](http://www.indiana.edu/~jslsoc/spost_help.htm) for information on what you can do to increase your odds of getting an answer!

---



### 2.6.1 Closing a log file

To close a log file, type

```
. log close
```

Also when you exit Stata, the log file closes automatically.

## 2.6.2 Viewing a log file

Regardless of whether a log file is open or closed, a log file can be viewed by selecting File→Log→View from the menu, and the log file will be displayed in the Viewer. When in the Viewer, you can print the log by selecting File→Print Viewer.... You can also view the log file by clicking on , which opens the log in the Viewer. If the Viewer window gets lost behind other windows, you can click on  to bring the Viewer to the front.

## 2.6.3 Converting from SMCL to plain text or PostScript

If you want to convert a log file in SMCL format to plain text, you can use the `translate` command. For example,

```
. translate mylog.smcl mylog.log, replace
(file mylog.log written in .log format)
```

tells Stata to convert the SMCL file `mylog.smcl` to a plain-text file called `mylog.log`. Or, you can convert a SMCL file to a PostScript file, which is useful if you are using  $\text{T}_{\text{E}}\text{X}$  or  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  or if you want to convert your output into Adobe's Portable Document Format. For example,

```
. translate mylog.smcl mylog.ps, replace
(file mylog.ps written in .ps format)
```

Converting can also be done through the menus by selecting File→Log→Translate.

## 2.7 Using and saving datasets

### 2.7.1 Data in Stata format

Stata uses its own data format with the extension `.dta`. The `use` command loads such data into memory. Pretend that we are working with the file `nomocc2.dta` in directory `d:\spostdata`. We can load the data by typing

```
. use d:\spostdata\nomocc2, clear
```

where the `.dta` extension is assumed by Stata. The `clear` option erases all data currently in memory and proceeds with loading the new data. Stata does not give an error if you include `clear` when there are no data in memory. If `d:\spostdata` was our working directory, we could use the simpler command

```
. use nomocc2, clear
```

If you have changed the data by deleting cases, merging in another file, or creating new variables, you can save the file with the `save` command. For example,

### 2.7.2 Data in other formats

```
. save d:\spostdata\nomocc3, replace
```

where again we did not need to include the `.dta` extension. Also we saved the file with a different name so that we can use the original data later. The `replace` option indicates that if the file `nomocc3.dta` already exists, Stata should overwrite it. If the file does not already exist, `replace` is ignored. If `d:\spostdata` was our working directory, we could save the file with

```
. save nomocc3, replace
```

`save` stores the data in a format that can be read only by Stata 8 or later. (Stata 8 and Stata 9 share the same dataset format.) If you use the command `saveold` instead of `save`, the dataset is written so that it can be read by Stata 7, but if your data contain multiple missing-value codes, a feature that became available in Stata 8, all the missing-value codes will be mapped to the smallest missing value (`.`).

---

**Tip: compress before saving** Before saving a file, run `compress`, which checks each variable to determine if it can be saved in a more compact form. For instance, binary variables fit into the `byte` type, which takes up only one-fourth of the space of the `float` type. If you run `compress`, it might make your data file much more compact, and at worst it will do no harm.

---

### 2.7.2 Data in other formats


To load data from another statistical package, such as SAS or SPSS, you need to convert it into Stata's format. The easiest way to do this is with a conversion program such as Stat/Transfer (<http://www.stattransfer.com>). We recommend obtaining one of these programs if you are using more than one statistical package or if you often share data with others who use different packages.

If you are moving data between Stata and SAS, you can use `fdasave`, `fdause`, and `fdadescribe` to convert datasets to and from the SAS XPORT Transport format. The commands begin with `fda` since the U.S. Food and Drug Administration has adopted the SAS XPORT format for new drug and other applications. Although a program like Stat/Transfer might be easier to use than these commands, if you move back and forth between SAS and Stata frequently, it is worth spending some time learning the `fda` commands. Type `help fdasave` or `search fda` for details.

Alternatively, but less conveniently, most statistical packages allow you to save and load data in ASCII format. You can load an ASCII file with the `infile` or `infix` commands and export it with the `outfile` command. The reference manual entry for `infile` contains an extensive discussion that is particularly helpful for reading in ASCII data, or you can type `help infile`.

### 2.7.3 Entering data by hand

Data can also be entered by hand using a spreadsheet-style editor. Although we do not recommend using the editor to change existing data (because it is too easy to make a mistake), we find that it is useful for entering small datasets. To enter the editor, click

on  or type `edit` on the command line. The *Getting Started* manual has a tutorial for the editor, but most people who have used a spreadsheet before will be immediately comfortable with the editor.

As you use the editor, every change that you make to the data is reported in the Results window and is captured by the log file, if it is open. For example, if you change `age` for the fifth observation to 32, Stata reports `replace age = 32 in 5`. This tells you that instead of using the editor, you could have changed the data with a `replace` command. When you close the editor, Stata asks if you really want to keep the changes or revert to the unaltered data.

## 2.8 Size limitations on datasets\*

If you get the error message `r(900): no room to add more observations` when trying to load a dataset or the message `r(901): no room to add more variables` when trying to add a new variable, you may need to allocate more memory. Typing `memory` shows how much memory is allocated to Stata and how much it is using. You can increase the amount of memory by typing `set memory #k` (for KB) or `#m` (for MB). For example, `set memory 32000k` or `set memory 32m` sets the memory to 32 MB.<sup>3</sup> If you have variables in memory, you must type `clear` before you can set the memory.

If you get the error `r(1000): system limit exceeded--see manual` when you try to load a dataset or add a variable, your dataset might have too many variables or the dataset might be too wide. Intercooled Stata is limited to a maximum of 2,047 variables, and the dataset can be up to 24,564 units wide (a binary variable has width 1, a double-precision variable has width 8, and a string variable has width equal to its length). Stata/SE allows 32,767 variables, and the dataset can be up to 393,192 units wide. String variables can be up to 244 characters. File transfer programs such as Stat/Transfer can drop specified variables and optimize variable storage. You can use these programs to create multiple datasets that each contain only the variables necessary for specific analyses.

## 2.9 Do-files

You can execute commands in Stata by typing one command at a time into the Command window and pressing Enter, as we have been doing. This interactive mode is

useful when you are learning Stata, exploring your data, or experimenting with alternative specifications of your regression model. You can also create a text file that contains a series of commands and then tell Stata to execute all the commands in that file, one after the other. These files, which are known as *do-files* because they use the extension `.do`, have the same function as “syntax files” in SPSS or “batch files” in other statistics packages. For more serious or complex work, we *always* use do-files because they make it easier to redo the analysis with small modifications later and because they provide an exact record of what has been done.

To get an idea of how do-files work, consider the file `example.do` saved in the working directory:

```
log using example, replace text
use binlfp2, clear
tabulate hc wc, row nolabel
log close
```

To execute a do-file, you type the command

```
do dofilename
```

from the Command window. For example, `do example` tells Stata to run each of the commands in `example.do`. (If the do-file is not in the working directory, you need to specify the directory, such as `do d:\spostdata\example`.) Executing `example.do` begins by opening the log `example.log`, and then loads `binlfp2.dta`, and finally constructs a table with `hc` and `wc`. Here is what the output looks like:

(Continued on next page)

3. Stata can use virtual memory if you need to allocate memory beyond that which is physically available on a system, but we find that virtual memory makes Stata unbearably slow.

```

-----
log: f:\spostdata\example.log
log type: text
opened on: 26 September 2005, 15:44:45
. use http://www.stata-press.com/data/lf2/binlfp2, clear
(Data from 1976 PSID-T Mroz)
. tabulate hc wc, row nolabel
+-----+
| Key |
+-----+
| frequency |
| row percentage |
+-----+

```

Husband College: 1=yes 0=no	Wife College: 0=no	College: 1=yes	Total
0	417 91.05	41 8.95	458 100.00
1	124 42.03	171 57.97	295 100.00
Total	541 71.85	212 28.15	753 100.00

```

. log close
log: f:\spostdata\example.log
log type: text
closed on: 26 September 2005, 15:44:45
-----

```

## 2.9.1 Adding comments

Stata has several different methods for denoting comments. We will make extensive use of two methods. First, on any given line, Stata treats everything that comes after // or after \* as comments that are simply echoed to the output. Second, on any given line, Stata ignores whatever comes after /// and treats the next line as a continuation of the current line. For example, the following do-file executes the same commands as the one above but includes comments:

```

//
// ==> short simple do-file
// ==> for didactic purposes
//
log using example, replace // this comment is ignored
// next we load the data
use binlfp2, clear
// tabulate husband's and wife's education
tabulate hc wc, /// the next line is treated as a continuation of this one
row nolabel
// close up
log close
// make sure there is a cr at the end!

```

## 2.9.4 Creating do-files

If you look at the do-files on our web site that reproduce the examples in this book, you will see that we use many comments. They are extremely helpful if others will be using your do-files or log files, or if there is a chance that you will use them again later.

## 2.9.2 Long lines

Sometimes you need to execute a command that is longer than the text that can fit onto a screen. If you are entering the command interactively, the Command window simply pushes the left part of the command off the screen as space is needed. Before entering a long command line in a do-file, however, you can use #delimit ; to tell Stata to interpret ";" as the end of a command. After the long command is entered, you can enter #delimit cr to return to using the carriage return as the end-of-line delimiter. For example,

```

#delimit ;
recode income91 1=500 2=1500 3=3500 4=4500 5=5500 6=6500 7=7500 8=9000
9=11250 10=13750 11=16250 12=18750 13=21250 14=23750 15=27500 16=32500
17=37500 18=45000 19=55000 20=67500 21=75000 *.=. ;
#delimit cr

```


Instead of the #delimit command, we could have used ///. For example,

```

recode income91 1=500 2=1500 3=3500 4=4500 5=5500 6=6500 7=7500 8=9000 ///
9=11250 10=13750 11=16250 12=18750 13=21250 14=23750 15=27500 16=32500 ///
17=37500 18=45000 19=55000 20=67500 21=75000 *.=.

```


## 2.9.3 Stopping a do-file while it is running


If you are running a command or a do-file that you want to stop before it completes execution, click on  or press Ctrl-Break.

## 2.9.4 Creating do-files

### Using Stata's Do-file Editor

Do-files can be created with Stata's built-in Do-file Editor. To use the editor, enter the command doedit to create a file to be named later or doedit filename to create or edit

a file named filename.do. You can also click on . The Do-file Editor is easy to use and works like most text editors (see *Getting Started* for further details). After you

finish your do-file, select Tools→Do to execute the file or click on .



### Using other editors to create do-files

Because do-files are plain text files, you can create do-files with any program that creates text files. Specialized text editors work much better than word processors such as WordPerfect or Microsoft Word. Among other things, with word processors it is easy to forget to save the file as plain text. Our own preference for creating do-files is TextPad (<http://www.textpad.com>), which runs in Windows. This program has many features that make it faster to create do-files. For example, you can create a “clip library” that contains frequently entered material, and you can obtain a syntax file from our web site that provides color coding of reserved words for Stata.

If you use an editor other than Stata’s built-in editor, you cannot run the do-file by clicking on an icon or selecting from a menu. Instead, you must switch from your editor and then enter the command `do filename`.

Two nice features of the Do-file Editor are that you can highlight a section of a file and Stata will execute only the commands that you have highlighted, and you can select Tools→Do to Bottom and Stata will execute commands starting wherever the cursor is located to the end of the file.

---

**Warning** Stata executes commands when it encounters a carriage return (i.e., the Enter key). If you do not include a carriage return after the last line in a do-file, that last line will not be executed. TextPad has a feature to enter that final, pesky carriage return automatically. To set this option in TextPad, select the option “Automatically terminate the last line of the file” in the preferences for the editor.

---

### 2.9.5 Recommended structure for do-files

This is the basic structure that we recommend for do-files:

```
// including version number ensures compatibility with later Stata releases
version 9
// if a log file is open, close it
capture log close
// don't pause when output scrolls off the page
set more off
// log results to file myfile.log
log using myfile, replace text
// * myfile.do - written 19 oct 2005 to illustrate do-files
//
// * your commands go here
//
// close the log file.
log close
```

Although the comments (which you can remove) should explain most of the file, there are a few points that we need to explain.

- The version 9 command indicates that the program was written for use in Stata 9. This command tells any future version of Stata that you want the commands that follow to work just as they did when you ran them in Stata 9. This prevents the problem of old do-files not running correctly in newer releases of the program.
- The command `capture log close` is very useful. Suppose that you have a do-file that starts with `log using mylog, replace`. You run the file and it “crashes” before reaching `log close`, which means that the log file remains open. If you revise the do-file and run it again, an error is generated when it tries to open the log file because the file is already open. The prefix `capture` tells Stata not to stop the do-file if the command that follows produces an error. Accordingly, `capture log close` closes the log file if it is open. If it is not open, the error generated by trying to close an already-closed file is ignored.

---

**Tip:** The command `cmdlog` is much like the `log` command, except that it creates a text file with extension `.txt` that saves all subsequent commands that are entered in the Command window (it does not save commands that are executed within a do-file). This is handy because it allows you to use Stata interactively and then make a do-file based on what you have done. You simply load the `cmdlog` that you saved, rename it to `newname.do`, delete commands you no longer want, and execute the new do-file. Your interactive session is now documented as a do-file. The syntax for opening and closing `cmdlog` files is the same as that for `log` (i.e., `cmdlog using` to open and `cmdlog close` to close), and you can have `log` and `cmdlog` files open simultaneously.

---

### 2.10 Using Stata for serious data analysis

Voltaire is said to have written *Candide* in three days. Creative work often rewards such inspired, seat-of-the-pants, get-the-details-later activity. *Data management does not*. Instead, effective data management rewards forethought, carefulness, double- and triple-checking of details, and meticulous, albeit tedious, documentation. Errors in data management are astonishingly (and painfully) easy to make. Moreover, tiny errors can have disastrous implications that can cost hours and even weeks of work. The extra time it takes to conduct data management carefully is rewarded many times over by the reduced risk of errors. That is, it helps prevent you from getting incorrect results that you do not know are incorrect. With this in mind, we begin with some broad, perhaps irritatingly practical, suggestions for doing data analysis efficiently and effectively.

1. *Ensure replicability by using do-files and log files for everything.* For data analysis to be credible, you must be able to reproduce *entirely and exactly* the trail from the original data to the tables in your paper. Thus any permanent changes you make to the data should be made by running do-files rather than by using the interactive mode. If you work interactively, be sure that the first thing you do is to open a log or cmdlog file. Then when you are done, you can use these files to create a do-file to reproduce your interactive results.
2. *Document your do-files.* Reasoning that is obvious today can be baffling in 6 months. We use comments extensively in our do-files, which are invaluable for remembering what we did and why we did it.
3. *Keep a research log.* For serious work, you should keep a diary that includes a description of every program you run, the research decisions that are being made (e.g., the reasons for recoding a variable in a particular way), and the files that are created. A good research log allows you to reproduce everything you have done starting with the original data. We cannot overemphasize how helpful such notes are when you return to a project that was put on hold, when you are responding to reviewers, or when you are moving on to the next stage of your research.
4. *Develop a system for naming files.* Usually it makes the most sense to have each do-file generate one log file with the same prefix (e.g., `clean_data.do`, `clean_data.log`). Names are easiest to organize when brief, but they should be long enough and logically related enough to make sense of the task the file does. Scott prefers to keep the names short and organized by major task (e.g., `recode01.do`), whereas Jeremy likes longer names (e.g., `make_income_vars.do`). Either is fine as long as it works for you.
5. *Use new names for new variables and files.* Never change a dataset and save it with the original name. If you drop three variables from `pcoms1.dta` and create two new variables, call the new file `pcoms2.dta`. When you transform a variable, give it a new name rather than simply replacing or recoding the old variable. For example, if you have a variable `workmom` with a five-point attitude scale, and you want to create a binary variable indicating positive and negative attitudes, create a new variable called `workmom2`.
6. *Use labels and notes.* When you create a new variable, give it a variable label. If it is a categorical variable, assign value labels. You can add a note about the new variable using the `notes` command (described below). When you create a new dataset, you can also use `notes` to document what it is.
7. *Double-check every new variable.* Cross-tabulating or graphing the old variable and the new variable are often effective for verifying new variables. As we describe below, using `list` with a subset of cases is similarly effective for checking transformations. Be sure to at least look carefully at the frequency distributions and summary statistics of variables in your analysis. You would not believe how many times puzzling regression results turn out to involve miscodings of variables that would have been immediately apparent by looking at the descriptive statistics.

8. *Practice good archiving.* If you want to retain hard copies of all your analyses, develop a system of binders for doing so rather than a set of intermingling piles on your desk. Back up everything. Make off-site backups or keep any on-site backups in a fireproof box. Should cataclysm strike, you will have enough other things to worry about without also having lost months or years of work.

## 2.11 Syntax of Stata commands

Think about the syntax of commands in everyday, spoken English. They usually begin with a verb telling the other person what they are supposed to do. Sometimes the verb is the entire command: “Help!” or “Stop!” Sometimes the verb needs to be followed by an object that indicates who or what the verb is to be performed on: “Help Dave!” or “Stop the car!” Sometimes the verb is followed by a qualifier that gives specific conditions under which the command should or should not be performed: “Give me a piece of pizza *if it doesn’t have mushrooms*” or “Call me *if you get home before nine*”. Verbs can also be followed by adverbs that specify that the action should be performed in some way that is different from how it might normally be, such as when a teacher commands her students to “Talk *clearly*” or “Walk *single file*”.

Stata follows an analogous logic, albeit with some other wrinkles that we will introduce later. The basic syntax of a command has four parts:

1. *Command:* What action do you want performed?
2. *Names of variables, files, or other objects:* On what things is the command to be performed?
3. *Qualifier on observations:* On which observations should the command be performed?
4. *Options:* What special things should be done in executing the command?

All commands in Stata require the first of these parts, just as it is hard in English to issue spoken commands without a verb. Each of the other three parts can be required, optional, or not allowed, depending on the particular command and circumstances. Here is an example of a command that features all four parts and uses `binlfp2.dta`, which we loaded earlier:

(Continued on next page)

```
. tabulate hc wc if age>40, row
```

Key			
frequency			
row percentage			
Husband College: 1=yes 0=no	Wife College: 1=yes 0=no		Total
	NoCol	College	
NoCol	263 91.96	23 8.04	286 100.00
College	58 38.93	91 61.07	149 100.00
Total	321 73.79	114 26.21	435 100.00

If you want to suppress the key, you can add the option `nokey`. For example, `tabulate hc wc, row nokey`.

`tabulate` is a command for making one- or two-way tables of frequencies. Here we want a two-way table of the frequencies of variables `hc` by `wc`. By putting `hc` first, we make this the row variable and `wc` the column variable. By specifying `if age>40`, we specify that the frequencies should include observations only for those older than 40. The option `row` indicates that row percentages should be printed as well as frequencies. These allow us to see that in 61% of the cases in which the husband had attended college, the wife had also done so, whereas wives had attended college only in 8% of cases in which the husbands had not. Notice the comma preceding row: *whenever options are specified, they are at the end of the command with a single comma to indicate where the list of options begins*. The precise ordering of multiple options after the comma is never important.

Next we provide more information on each of the four components.

### 2.11.1 Commands

Commands define the tasks that Stata is to perform. A great thing about Stata is that the set of commands is deliciously open ended. It expands not just with new releases of Stata but also when users add their own commands, such as our `SPost` commands. Each new command is stored in its own file, ending with the extension `.ado`. Whenever Stata encounters a command that is not in its built-in library, it searches various directories for the appropriate `ado`-file. The list of the directories it searches (and the order that it searches them) can be obtained by typing `adopath`.

### 2.11.2 Variable lists

Variable names are case sensitive. For example, you could have three different variables named `income`, `Income`, and `inCome`. Of course, this is not a good idea because it leads to confusion. To keep life simple, we stick exclusively to lowercase names. Starting with Stata 7, Stata allows variable names up to 32 characters long, compared with the eight-character maximum imposed by earlier versions of Stata and many other statistics packages. In practice, we try not to give variables names more than eight characters, as this makes it easier to share data with people who use other packages. Also we recommend using short names because longer variable names become unwieldy to type. (Although variable names can be abbreviated to whatever initial set of characters identifies the variable uniquely, we worry that too much reliance on this feature might cause one to make mistakes.)

If you list no variables, many commands assume that you want to perform the operation on every variable in the dataset. For example, the `summarize` command provides summary statistics on the listed variables:

```
. summarize age inc k5
```

Variable	Obs	Mean	Std. Dev.	Min	Max
age	753	42.53785	8.072574	30	60
inc	753	20.12897	11.6348	-.0290001	96
k5	753	.2377158	.523959	0	3

We could also get summary statistics on every variable in our dataset by just typing

```
. summarize
```

Variable	Obs	Mean	Std. Dev.	Min	Max
lfp	753	.5683931	.4956295	0	1
k5	753	.2377158	.523959	0	3
k618	753	1.353254	1.319874	0	8
age	753	42.53785	8.072574	30	60
wc	753	.2815405	.4500494	0	1
hc	753	.3917663	.4884694	0	1
lwg	753	1.097115	.5875564	-2.054124	3.218876
inc	753	20.12897	11.6348	-.0290001	96

You can also select all variables that begin or end with the same letters by using the wildcard operator `*`. For example,

```
. summarize k*
```

Variable	Obs	Mean	Std. Dev.	Min	Max
k5	753	.2377158	.523959	0	3
k618	753	1.353254	1.319874	0	8

### 2.11.3 if and in qualifiers

Stata has two qualifiers that restrict the sample that is analyzed: `if` and `in`. `in` performs operations on a range of consecutive observations. Typing `summarize in 20/100` gives summary statistics based only on the 20th through 100th observations. `in` restrictions depend on the current sort order of the data, meaning that if you re-sort your data, the 81 observations selected by the restriction `summarize in 20/100` might be different.<sup>4</sup>

In practice, `if` conditions are used much more often than `in` conditions. `if` restricts the observations to those that fulfill a specified condition. For example, `summarize if age<50` provides summary statistics for only those observations where `age` is less than 50. Here is a list of the elements that can be used to construct logical statements for selecting observations with `if`:

Operator	Definition	Example
<code>==</code>	Equal to	<code>if female==1</code>
<code>!=</code>	Not equal to	<code>if female!=1</code>
<code>&gt;</code>	Greater than	<code>if age&gt;20</code>
<code>&gt;=</code>	Greater than or equal to	<code>if age&gt;=21</code>
<code>&lt;</code>	Less than	<code>if age&lt;66</code>
<code>&lt;=</code>	Less than or equal to	<code>if age&lt;=65</code>
<code>&amp;</code>	And	<code>if age==21 &amp; female==1</code>
<code> </code>	Or	<code>if age==21   educ&gt;16</code>

There are two important things about the `if` qualifier:

1. Use a double equal sign (e.g., `summarize if female==1`) to specify a condition to test. When assigning a value to something, such as when creating a new variable, use a single equal sign (e.g., `gen newvar=1`). Putting these examples together results in `gen newvar=1 if female==1`.
2. The missing-value codes are the largest positive numbers. This implies that Stata treats missing cases as positive infinity when evaluating `if` expressions. In other words, if you type `summarize ed if age>50`, the summary statistics for `ed` are calculated on all observations where `age` is greater than 50, including cases where the value of `age` is missing. You must be careful of this when using `if` with `>` or `>=` expressions. If you type `summarize ed if age<.`, Stata gives summary statistics for cases where `age` is not missing. (Note that `.` is the smallest of the 27 missing-value codes. See section 2.12.3 for more details on missing values.) Entering `summarize ed if age>50 & age<.` provides summary statistics for those cases where `age` is greater than 50 and is not missing.

4. In Stata 6 and earlier, some official Stata commands changed the sort order of the data, but fortunately this quirk was removed in Stata 7. As of Stata 7, no properly written Stata command should change the sort order of the data, although readers should beware that user-written programs may not always follow proper Stata programming practice.

### Examples of if qualifier

If we wanted summary statistics on income for only those respondents who were between the ages of 25 and 65, we would type

```
. summarize income if age>=25 & age<=65
```

If we wanted summary statistics on income for only female respondents who were between the ages of 25 and 65, we would type

```
. summarize income if age>=25 & age<=65 & female==1
```

If we wanted summary statistics on income for the remaining female respondents—that is, those who are younger than 25 or older than 65—we would type

```
. summarize income if (age<25 | age>65) & age<. & female==1
```

We need to include `& age<.` because Stata treats missing codes as positive infinity. The condition `(age<25 | age>65)` would otherwise include those cases for which `age` is missing.

---

**Tip: Removing the separator** If you do not like the horizontal separator that appears after every five variables in the output for `summarize`, you can remove the lines with the option `sep(0)`.

---

### 2.11.4 Options



Options are set off from the rest of the command by a comma. Options can often be abbreviated, although whether and how they can be abbreviated varies across commands. In this book, we rarely cover all the available options available for any given command, but you can check the manual or use `help` for more options that might be useful for your analyses.

## 2.12 Managing data

### 2.12.1 Looking at your data

There are two easy ways to look at your data.

`browse` opens a spreadsheet in which you can scroll to look at the data, but you cannot change the data. You can look and change data with the `edit` command, but this is risky. We much prefer making changes to our data using do-files, even when we are changing the value of only one variable for one observation. The browser is also

available by clicking on , whereas the data editor is available by clicking on .

`list` creates a list of values of specified variables and observations. `if` and `in` qualifiers can be used to look at just a portion of the data, which is sometimes useful for checking that transformations of variables are correct. For example, if you want to confirm that the variable `lninc` has been correctly constructed as the natural log of `inc`, typing `list inc lninc in 1/20` lets you see the values of `inc` and `lninc` for the first 20 observations.

## 2.12.2 Getting information about variables

There are several methods for obtaining basic information about your variables. Here are five commands that we find useful. Which one you use depends mostly on the kind and level of detail you need.

`describe` provides information on the size of the dataset and the names, labels, and types of variables. For example,

```
. use http://www.stata-press.com/data/lf2/binlfp2, clear
(Data from 1976 PSID-T Mroz)
. describe
Contains data from binlfp2.dta
  obs:      753                Data from 1976 PSID-T Mroz
  vars:      8                30 Apr 2001 16:17
  size:    13,554 (98.7% of memory free)  (_dta has notes)
```

variable name	storage type	display format	value label	variable label
lfp	byte	%9.0g	lfp1bl	Paid Labor Force: 1=yes 0=no
k5	byte	%9.0g		# kids < 6
k618	byte	%9.0g		# kids 6-18
age	byte	%9.0g		Wife's age in years
wc	byte	%9.0g	collbl	Wife College: 1=yes 0=no
hc	byte	%9.0g	collbl	Husband College: 1=yes 0=no
lwg	float	%9.0g		Log of wife's estimated wages
inc	float	%9.0g		Family income excluding wife's

Sorted by: lfp

`summarize` provides summary statistics. By default, `summarize` presents the number of nonmissing observations, the mean, the standard deviation, the minimum values, and the maximum. Adding the `detail` option includes more information. For example,

## 2.12.2 Getting information about variables

```
. summarize age, detail
```

Wife's age in years			
Percentiles		Smallest	
1%	30	30	
5%	30	30	
10%	32	30	Obs 753
25%	36	30	Sum of Wgt. 753
50%	43		Mean 42.53785
		Largest	Std. Dev. 8.072574
75%	49	60	
90%	54	60	Variance 65.16645
95%	56	60	Skewness .150879
99%	59	60	Kurtosis 1.981077

`tabulate` creates the frequency distribution for a variable. For example,

```
. tabulate hc
```

Husband College: 1=yes 0=no	Freq.	Percent	Cum.
NoCol	458	60.82	60.82
College	295	39.18	100.00
Total	753	100.00	

If you do not want the value labels included, type

```
. tabulate hc, nolabel
```

Husband College: 1=yes 0=no	Freq.	Percent	Cum.
0	458	60.82	60.82
1	295	39.18	100.00
Total	753	100.00	

If you want a two-way table, type

```
. tabulate hc wc
```

Husband College: 1=yes 0=no	Wife College: 1=yes 0=no		Total
	NoCol	College	
NoCol	417	41	458
College	124	171	295
Total	541	212	753

By default, `tabulate` does not tell you the number of missing values for either variable. Specifying the `missing` option includes missing values. We recommend this option whenever you are generating a frequency distribution to check that some transformation

was done correctly. The options row, col, and cell request row, column, and cell percentages along with the frequency counts. The option chi2 reports the  $\chi^2$  for a test that the rows and columns are independent.

tab1 presents univariate frequency distributions for each variable listed. For example,

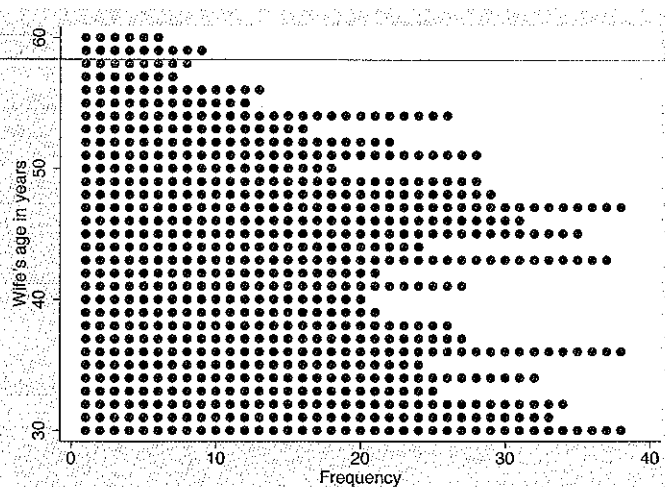
```
. tab1 hc wc
-> tabulation of hc
```

Husband College: 1=yes 0=no	Freq.	Percent	Cum.
NoCol	458	60.82	60.82
College	295	39.18	100.00
Total	753	100.00	

```
-> tabulation of wc
```

Wife College: 1=yes 0=no	Freq.	Percent	Cum.
NoCol	541	71.85	71.85
College	212	28.15	100.00
Total	753	100.00	

dotplot generates a quick graphical summary of a variable, which is useful for quickly checking your data. For example, the command dotplot age leads to the following graph:



This graph will appear in a new window called the Graph window. Details on saving, printing, and enhancing graphs are given in section 2.16.

codebook summarizes a variable in a format designed for printing a codebook. For example, codebook age produces

```
. codebook age
```

---

age	Wife's age in years				
	type: numeric (byte)				
	range: [30,60]				
	unique values: 31	units: 1			
		missing .. 0/753			
	mean: 42.5378				
	std. dev: 8.07257				
	percentiles: 10%	25%	50%	75%	90%
	32	36	43	49	54

### 2.12.3 Missing values

Although numeric missing values are automatically excluded when Stata fits models, they are stored as the largest positive values. Twenty-seven missing values are available, with the ordering

all numbers < . < .a < .b < ... < .z

This way of handling missing values can have unexpected consequences when determining samples. For instance, the expression `if age>65` is true when age has a value greater than 65 and when age is missing. Similarly, the expression `occupation!=1` is true if occupation is not equal to 1 or occupation is missing. When expressions such as these are required, be sure to explicitly exclude any unwanted missing values. For instance, `age>65 & age<.` would be true only for those people whose age is not missing and who are over 65. Similarly, `occupation!=1 & occupation <.` would be true only when the occupation is not missing and not equal to 1.

The different missing values can be used to record the distinct reasons why a variable is missing. For instance, consider a survey that asked people about their driving records. The variable that records whether someone received a ticket after being involved in an accident could be missing because the respondent had not been involved in any accidents or because the person refused to answer the question.

### 2.12.4 Selecting observations

As previously mentioned, you can select cases with the `if` and `in` qualifiers. For example, `summarize age if wc==1` provides summary statistics on age for only those observations where `wc` equals 1. Sometimes it is simpler to remove the cases with either the `drop` or `keep` commands. `drop` removes observations from memory (not from the `.dta` file) based on an `if` or `in` specification. The syntax is

```
drop [in] [if]
```

Only observations that do *not* meet those conditions are left in memory. For example, `drop if wc==1` keeps only those cases where `wc` is not equal to 1, including observations with missing values on `wc`.

`keep` has the same syntax as `drop` and deletes all cases *except* those that meet the condition. For example, `keep if wc==1` keeps only those cases where `wc` is 1; all other observations, including those with missing values for `wc`, are dropped from memory. After selecting the observations that you want, you can save the remaining variables to a new dataset with the `save` command.

### 2.12.5 Selecting variables

You can also select which variables you want to keep. The syntax is

```
drop variable_list
```

```
keep variable_list
```

With `drop`, all variables are kept except those that are explicitly listed. With `keep`, only those variables that are explicitly listed are kept. After selecting the variables that you want, you can save the remaining variables to a new dataset with the `save` command.

## 2.13 Creating new variables

The variables that you analyze are often constructed differently from the variables in the original dataset. Here we consider basic methods for creating new variables. Our examples always create a new variable from an old variable rather than transforming an existing variable. Even though you can simply transform an existing variable, we find that this leads to mistakes.

### 2.13.1 generate command

`generate` creates new variables. For example, to create `age2` as an exact copy of `age`, type

```
. generate age2 = age
. summarize age2 age
```

Variable	Obs	Mean	Std. Dev.	Min	Max
age2	753	42.53785	8.072574	30	60
age	753	42.53785	8.072574	30	60

The results of `summarize` show that the two variables are identical. We used a single equal sign because we are making a variable equal to some value.

Observations excluded by `if` or `in` qualifiers in the `generate` command are coded as missing. For example, to generate `age3` that equals `age` for those over 40 but is otherwise missing, type

### 2.13.1 generate command

```
. gen age3 = age if age>40
(318 missing values generated)
. summarize age3 age
```

Variable	Obs	Mean	Std. Dev.	Min	Max
age3	435	48.3977	4.936509	41	60
age	753	42.53785	8.072574	30	60

Whenever `generate` (or `gen`, as it can be abbreviated) produces missing values, it tells you how many cases are missing.

`generate` can also create variables that are mathematical functions of existing variables. For example, we can create `agesq` that is the square of `age` and `lnage` that is the natural log of `age`:

```
. gen agesq = age^2
. gen lnage = ln(age)
```

For quick reference, here is a list of the standard mathematical operators

Operator	Definition	Example
+	Addition	<code>gen y = a+b</code>
-	Subtraction	<code>gen y = a-b</code>
/	Division	<code>gen density = pop/area</code>
*	Multiplication	<code>gen y = a*b</code>
^	Take to a power	<code>gen y = a^3</code>

and some particularly useful functions:

Function	Definition	Example
<code>ln</code>	Natural log	<code>gen lnwage = ln(wage)</code>
<code>exp</code>	Exponential	<code>gen y = exp(a)</code>
<code>sqrt</code>	Square root	<code>gen agesqrt = sqrt(age)</code>

For a complete list of functions in Stata, type `help functions`.

**Tip:** Although `gen newvar=oldvar` is the most intuitive way of creating a copy of the values of `oldvar` as `newvar`, sometimes `clonevar newvar=oldvar` is a better alternative. `clonevar` copies not only the values of `oldvar` but also the variable label, value label, and other attributes.

### 2.13.2 replace command

replace has the same syntax as generate but is used to change values of a variable that already exists. For example, say we want to make a new variable, age4, that equals age if age is over 40 but equals 40 for all persons aged 40 and under. First, we create age4 equal to age. Then we replace those values we want to change:

```
. gen age4 = age
. replace age4 = 40 if age<40
(298 real changes made)
. summarize age4 age
```

Variable	Obs	Mean	Std. Dev.	Min	Max
age4	753	44.85126	5.593896	40	60
age	753	42.53785	8.072574	30	60

replace reports how many values were changed. This is useful in verifying that the command did what you intended. Also summarize confirms that the minimum value of age is 30 and that age4 now has a minimum of 40 as intended.

**Warning** Of course, we could have simply changed the original variable:

```
replace age = 40 if age<40. But, if we did this and saved the data, there would
be no way to return to the original values for age if we later needed them.
```

### 2.13.3 recode command

The values of *existing* variables can also be changed using the recode command. With recode you specify a set of correspondences between old values and new ones. For example, you might want old values of 1 and 2 to correspond to new values of 1, old values of 3 and 4 to correspond to new values of 2, and so on. This is particularly useful for combining categories. To use this command, we recommend that you start by making a copy of an existing variable. Then recode the copy. Or, to be more efficient, you can use the generate(*newvariablename*) option with recode. With this option, Stata creates a new variable instead of overwriting the old one. recode is best explained by example, several of which we include below (for more, type help recode).

To change 1 to 2 and 3 to 4 but leave all other values unchanged, type

```
. recode origvar (1=2) (3=4), generate(myvar1)
(23 differences between origvar and mvar1)
```

To change 2 to 1 and change all other values (including missing) to 0:

```
. recode origvar (2=1) (*=0), gen(myvar2)
(100 differences between origvar and myvar2)
```

### 2.13.4 Common transformations for RHS variables

where the asterisk indicates all values, including missing values, that have not been explicitly recoded.

To change 2 to 1 and change all other values *except missing* to 0:

```
. recode origvar (2=1) (nonmissing=0), gen(myvar3)
(89 differences between origvar and myvar3)
```

To change values from 1 to 4 inclusive to 2 and keep other values unchanged:

```
. recode origvar (1/4=2), gen(myvar4)
(40 differences between origvar and myvar4)
```

To change values 1, 3, 4, and 5 to 7 and keep other values unchanged:

```
. recode origvar (1 3 4 5=7), gen(myvar5)
(55 differences between origvar and myvar5)
```

To change all values from the minimum through 5 to the minimum:

```
. recode origvar (min/5=min), gen(myvar6)
(56 differences between origvar and myvar6)
```

To change missing values to 9:

```
. recode origvar (missing=9), gen(myvar7)
(11 differences between origvar and myvar7)
```

To change values of -999 to missing:

```
. recode origvar (-999=.), gen(myvar8)
(56 differences between origvar and myvar8)
```

recode can be used to recode several variables at once if they are all to be recoded the same way. Just include all the variable names before the instructions on how they are to be recoded, and include all the names for new variables (if you do not want the old variables to be overwritten) within the parentheses of the generate() option.

### 2.13.4 Common transformations for RHS variables

For the models we discuss in later chapters, you can use many of the tricks you learned for coding right-hand-side (i.e., independent) variables in the linear regression model. Here are some useful examples. Details on how to interpret such variables in regression models are given in chapter 9.

#### Breaking a categorical variable into a set of binary variables

To use a *j*-category nominal variable as an independent variable in a regression model, you need to create a set of *j* - 1 binary variables, also known as dummy variables or indicator variables. To show how to do this, we use educational attainment (*degree*),



which is coded as 0 = no diploma, 1 = high school diploma, 2 = associate's degree, 3 = bachelor's degree, and 4 = postgraduate degree, with some missing data. We want to make four binary variables with the "no diploma" category serving as our reference category. We also want observations that have missing values for degree to have missing values in each of the dummy variables that we create. The simplest way to do this is to use the `generate` option with `tabulate`:

```
. use http://www.stata-press.com/data/lf2/gsskidvalue2, clear
(1993 and 1994 General Social Survey)
. tabulate degree, generate(edlevel)
```

rs highest degree	Freq.	Percent	Cum.
lt high school	801	17.47	17.47
high school	2,426	52.92	70.40
junior college	273	5.96	76.35
bachelor	750	16.36	92.71
graduate	334	7.29	100.00
Total	4,584	100.00	

The `generate(name)` option creates a new binary variable for each category of the specified variable. Here `degree` has five categories, so five new variables are created. These variables all begin with `edlevel`, the root that we specified with the `generate(edlevel)` option. We can check the five new variables by typing `summarize edlevel*`:

```
. summarize edlevel*
```

Variable	Obs	Mean	Std. Dev.	Min	Max
edlevel1	4584	.1747382	.3797845	0	1
edlevel2	4584	.5292321	.4991992	0	1
edlevel3	4584	.059555	.2366863	0	1
edlevel4	4584	.1636126	.369964	0	1
edlevel5	4584	.0728621	.2599384	0	1

By cross-tabulating the new `edlevel1` by the original `degree`, we can see that `edlevel1` equals 1 for individuals with no high school diploma and equals 0 for everyone else except the 14 observations with missing values for `degree`:

```
. tabulate degree edlevel1, missing
```

rs highest degree	degree==lt high school			Total
	0	1	.	
lt high school	0	801	0	801
high school	2,426	0	0	2,426
junior college	273	0	0	273
bachelor	750	0	0	750
graduate	334	0	0	334
.	0	0	14	14
Total	3,783	801	14	4,598

One limitation of using the `generate(name)` option of `tabulate` is that it works only when there is a one-to-one correspondence between the original categories and the dummy variables that we wish to create. So, let's suppose that we want to combine high school graduates and those with associate's degrees when creating our new binary variables. Say also that we want to treat those without high school diplomas as the omitted category. The following is one way to create the three binary variables that we need:

```
. gen hsdeg = (degree==1 | degree==2) if degree<.
(14 missing values generated)
. gen coldeg = (degree==3) if degree<.
(14 missing values generated)
. gen graddeg = (degree==4) if degree<.
(14 missing values generated)
. tabulate degree coldeg, missing
```

rs highest degree	coldeg			Total
	0	1	.	
lt high school	801	0	0	801
high school	2,426	0	0	2,426
junior college	273	0	0	273
bachelor	0	750	0	750
graduate	334	0	0	334
.	0	0	14	14
Total	3,834	750	14	4,598

To understand how this works, you need to know that when Stata is presented with an expression (e.g., `degree==3`) where it expects a value, it evaluates the expression and assigns it a value of 1 if true and 0 if false. Consequently, `gen coldeg = (degree==3)` creates the variable `coldeg` that equals 1 whenever `degree` equals 3 and 0 otherwise. By adding `if degree<.` to the end of the command, we assign these values *only* to observations in which the value of `degree` is not missing. If an observation has a missing value for `degree`, these cases are given a missing value.

### More examples of creating binary variables

Binary variables are used so often in regression models that it is worth providing more examples of generating them. In the dataset that we use in chapter 5 (`ordwarm2.dta`), the independent variable for respondent's education (`ed`) is measured in years. We can create a dummy variable that equals 1 if the respondent has at least 12 years of education and 0 otherwise:

```
. gen ed12plus = (ed>=12) if ed<.
```

We might also want to create a set of variables that indicates whether an individual has less than 12, between 13 and 16, or 17 or more years of education. This is done as follows:

```
. gen edlt13 = (ed<=12) if ed<.
. gen ed1316 = (ed>=13 & ed<=16) if ed<.
. gen ed17plus = (ed>17) if ed<.
```

**Tip: Naming dummy variables** Whenever possible, we name dummy variables so that 1 corresponds to “yes” and 0 to “no”. With this convention, a dummy variable called `female` is coded 1 for women (i.e., yes, the person is female) and 0 for men. If the dummy variable were named `sex`, there would be no immediate way to know what 0 and 1 mean.

The `recode` command can also be used to create binary variables. The variable `warm` contains responses to the question of whether working women can have as warm a relationship with their children as women who do not work: 1 = strongly disagree, 2 = disagree, 3 = agree, and 4 = strongly agree. To create a dummy indicating agreement as opposed to disagreement, type

```
. gen wrmagree = warm
. recode wrmagree 1=0 2=0 3=1 4=1
(wrmagree: 2293 changes made)
. tabulate wrmagree warm
```

wrmagree	Mom can have warm relations with child				Total
	SD	D	A	SA	
0	297	723	0	0	1,020
1	0	0	856	417	1,273
Total	297	723	856	417	2,293

## Nonlinear transformations

Nonlinear transformations of the independent variables are commonly used in regression models. For example, researchers often include both age and age<sup>2</sup> as explanatory variables to allow the effect of a 1-year increase in age to change as one gets older. We can create a squared term as

```
. gen agesq = age*age
```

Likewise, income is often logged so that the impact of each additional dollar decreases as income increases. The new variable can be created as

```
. gen lnlncome = ln(income)
(495 missing values generated)
```

We can use the minimum and maximum values reported by `summarize` as a check on our transformations:

## 2.14.1 Variable labels

```
. summarize age agesq income lnlncome
```

Variable	Obs	Mean	Std. Dev.	Min	Max
age	4598	46.12375	17.33162	18	99
agesq	4598	2427.72	1798.477	324	9801
income	4103	34790.7	22387.45	1000	75000
lnlncome	4103	10.16331	.8852605	6.907755	11.22524

### Interaction terms

In regression models, you can include interactions by taking the product of two independent variables. For example, we might think that the effect of family income differs for men and women. If `sex` is measured as the dummy variable `female`, we can construct an interaction term as follows:

```
. gen feminc = female * income
(495 missing values generated)
```

## 2.14 Labeling variables and values

*Variable* labels provide descriptive information about what a variable measures. For example, the variable `agesq` might be given the label “age squared”, or `warm` could have the label “Mother has a warm relationship”. *Value* labels provide descriptive information about the different values of a categorical variable. For example, value labels might indicate that the values 1–4 correspond to survey responses of strongly agree, agree, disagree, and strongly disagree. Adding labels to variables and values is not much fun, but in the long run, it can save much time and prevent misunderstandings. Also many of the commands in *SPost* produce output that is more easily understood if the dependent variable has value labels.

### 2.14.1 Variable labels

The `label variable` command attaches a label of up to 80 characters to a variable. For example,

(Continued on next page)

```
. label variable agesq "Age squared"
. describe agesq
```

variable name	storage type	display format	value label	variable label
agesq	float	%9.0g		Age squared

If no label is specified, any existing variable label is removed. For example,

```
. label variable agesq
. describe agesq
```

variable name	storage type	display format	value label	variable label
agesq	float	%9.0g		

**Tip: Use short labels** Although variable labels of up to 80 characters are allowed, we recommend that you use short labels whenever possible. Output often does not show all 80 characters. For the same reason, we also find it useful to put the most important information at the beginning of the label. That way, if the label is truncated, you will still see the critical information.

**Tip: Searching variable labels** Typing `lookfor string` will search the data file and list all variables in which *string* appears in either the variable name or label.

## 2.14.2 Value labels

Beginners often find value labels in Stata confusing. Remember that Stata splits the process of labeling values into two steps: creating labels and then attaching the labels to variables.

Step 1 defines a set of labels *without* reference to a variable. Here are some examples of value labels:

```
. label define yesno 1 yes 0 no
. label define posneg4 1 veryN 2 negative 3 positive 4 veryP
. label define agree4 1 StrongA 2 Agree 3 Disagree 4 StrongD
. label define agree5 1 StrongA 2 Agree 3 Neutral 4 Disagree 5 StrongD
```

First, each set of labels is given a unique name (e.g., `yesno`, `agree4`). Second, individual labels are associated with a specific value. Third, none of our labels has spaces in them (e.g., we use `StrongA` not `Strong A`). Although you can have spaces if you place the label within quotes, some commands crash when they encounter blanks in value labels. So, it is easier not to do it. We have also found that the period, colon, and left curly bracket (`{`) in value labels can cause similar problems. Fourth, our labels are eight

letters or shorter in length because some programs have trouble with value labels longer than eight letters.

Step 2 assigns the value labels to variables. Let's say that variables `female`, `black`, and `anykids` all imply yes/no categories with 1 as yes and 0 as no. To assign labels to the values, we would use the following commands:

```
. label values female yesno
. label values black yesno
. label values anykids yesno
. describe female black anykids
```

variable name	storage type	display format	value label	variable label
female	byte	%9.0g	yesno	Female
black	byte	%9.0g	yesno	Black
anykids	byte	%9.0g	yesno	R have any children?

The output for `describe` shows which value labels were assigned to which variables. The new value labels are reflected in the output from `tabulate`:

```
. tabulate anykids
```

R have any children?	Freq.	Percent	Cum.
no	1,267	27.64	27.64
yes	3,317	72.36	100.00
Total	4,584	100.00	

For the degree variable that we looked at earlier, we assign labels with

```
. label define degree 0 "no_hs" 1 "hs" 2 "jun_col" 3 "bachelor" 4 "graduate"
. label values degree degree
. tabulate degree
```

rs highest degree	Freq.	Percent	Cum.
no_hs	801	17.47	17.47
hs	2,426	52.92	70.40
jun_col	273	5.96	76.35
bachelor	750	16.36	92.71
graduate	334	7.29	100.00
Total	4,584	100.00	

We used underscores (`_`) instead of spaces.

If you want a list of the value labels being used in your current dataset, use the command `labelbook`, which provides a detailed list of all value labels, including which labels are assigned to which variables. This can be useful both in setting up a complex dataset and for documenting your data.

### 2.14.3 notes command

The `notes` command allows you to add notes to the dataset as a whole or to specific variables. Because the notes are saved in the dataset, the information is always available when you use the data. Here we add one note describing the dataset and two that describe the income variable:

```
. notes: General Social Survey extract for Stata book
. notes income: self-reported family income, measured in dollars
. notes income: refusals coded as missing
```

We can review the notes by typing `notes`:

```
. notes
_dta:
  1. General Social Survey extract for Stata book
income:
  1. self-reported family income, measured in dollars
  2. refusals coded as missing
```

If we save the dataset after adding notes, the notes become a permanent part of the dataset.

## 2.15 Global and local macros

Although macros are most often used when writing ado-files, they are also very useful in do-files. Later in the book, and especially in chapter 9, we use macros extensively. Accordingly, we will discuss them briefly here. Readers with less familiarity with Stata might want to skip this section for now and read it later when macros are used in our examples.

In Stata, you can assign values or strings to *macros*. Whenever Stata encounters the macro name, it automatically substitutes the contents of the macro. For example, pretend that you want to generate a series of two-by-two tables where you want cell percentages, requiring the `cell` option; missing values, requiring the `missing` option; values printed instead of value labels, requiring the `nolabel` option; and the chi-squared test statistic, requiring the `chi2` option. Even if you use the shortest abbreviations, this would require typing `“, ce m nol ch”` at the end of each `tab` command. Instead, you could use the following command to define a global macro called `myopt`:

```
. global myopt = ", cell miss nolabel chi2 nokey"
```

Then whenever you type `$myopt` (the `$` tells Stata that `myopt` is a global macro), Stata substitutes `, cell miss nolabel chi2 nokey`. If you type

```
. tab lfp wc $myopt
```

Stata interprets this as if you had typed

### 2.15 Global and local macros

```
. tab lfp wc, cell miss nolabel chi2 nokey
```

Global macros are “global” because, once they are set, they can be accessed by any do (or ado) program in the current session. The flip side is that the global macros that you are using can be reset by any of the do- or ado-files that you use along the way. By contrast, “local” macros can be accessed only within the do- or ado-file in which they are defined. When the do- or ado-file program terminates, the local macro disappears. We prefer using local macros whenever possible because you do not have to worry about conflicts with other programs or do-files that try to use the same macro name for a different purpose. Local macros are defined using the `local` command, and they are referenced by placing the name of the local macro in single quotes; for example, ``myopt'`. The two single quote marks use different symbols (on many keyboards, the left single quote is in the upper left-hand corner, whereas the right single quote is next to the Enter key). If the operations we just performed were in a do-file, we could have produced the same output with the following lines:

```
. local opt = ", cell miss nolabel chi2 nokey"
. tab lfp wc `opt'
(output omitted)
```

Local and global macros can also be used as a shorthand way to refer to lists of variables. For example, you could use these commands to create lists of variables:

```
. local demogvars "age white female"
. local edvars "highsch college graddeg"
```

Then when you run regression models, you could use the command

```
. regress y `demogvars' `edvars'
```

which Stata would translate into

```
. regress y age white female highsch college graddeg
```

Or, you could use the command

```
. regress y `demogvars' `edvars' x1 x2 x3
```

which Stata would translate into

```
. regress y age white female highsch college graddeg x1 x2 x3
```

This technique has several advantages. First, it is easier to write the commands since you do not have to keep retyping a long list of variables. Second, if you change the set of demographic variables that you want to use, you have to do it only in one place, which reduces the chance of errors.

Often when you use a local macro name for a list of variables, the list becomes longer than one line. As with other Stata commands that extend over one line, you can use `///`, as in

```
local vars age age squared income education female occupation dadeduc dadocc ///
momeduc momocc
```

You can also define macros to equal the result of computations. After entering global four = 2+2, the value 4 will be substituted for \$four. Also Stata contains many *macro functions* in which items retrieved from memory are assigned to macros. For example, to display the variable label that you have assigned to the variable wc, you can type

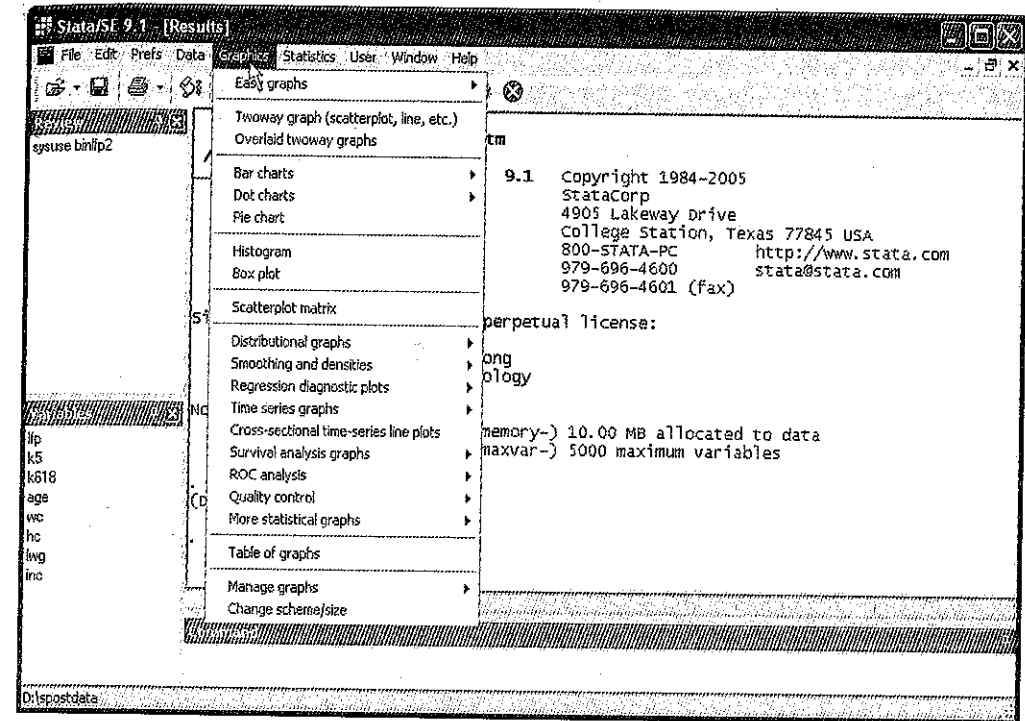
```
. global wclabel : variable label wc
. display "$wclabel"
Wife College: 1=yes 0=no
```

We have only scratched the surface of the potential of macros. Macros are immensely flexible and are indispensable for a variety of advanced tasks that Stata can perform. Perhaps most importantly, macros are essential for doing any meaningful Stata programming. If you look at the ado-files for the commands we have written for this book, you will see many instances of macros, and even of macros within macros. For users interested in advanced applications, the macro entry in the *Programming Reference Manual* should be read closely.

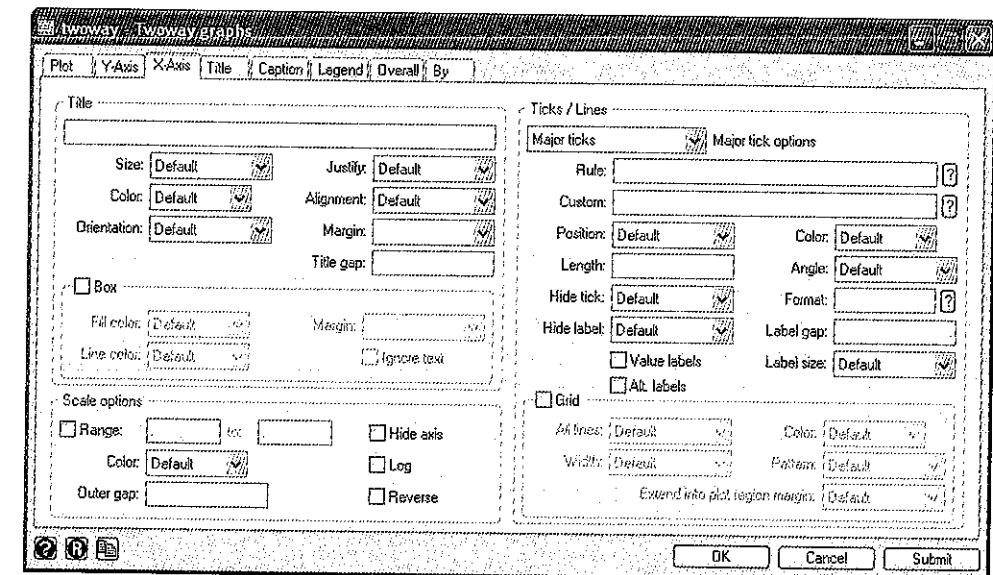
## 2.16 Graphics

Stata has a very extensive and powerful graphics system. Not only can you create many different kinds of graphs, but you have a great deal of control over almost all aspects of a graph's appearance. The cost of this is that the syntax for making a graph can get complicated. Here we provide a brief introduction to graphics in Stata, focusing on the types of graphs that we use in later chapters. Our hope is to provide a basic understanding of how the graphics system works so that you can start using it. For more information, we suggest the following. Stata has a manual dedicated to the syntax for graphics. Although we find the *Stata Graphics Reference Manual* to be an invaluable reference when you already have a good understanding of what you want to do, we find it less helpful when you want to be reminded of the way to do something, what an option is called, or to get ideas about what kinds of graph to use. In this regard, we find Mitchell's 2004 *A Visual Guide to Stata Graphics* to be extremely useful. This book shows hundreds of graphs drawn in Stata, along with the commands used to generate them. The book is organized in a way that makes it easy to scan the pictures until you see a graph that does what you want. Then you can look at the text to find out which options to use.

The way we use Stata to make graphs differs from how we use Stata to fit models or do virtually anything else. Namely, when making graphs, we make extensive use of dialog boxes. If you pull down the Graphics menu (or type Alt-g), you will see a list of the plot types and families of plot types available in Stata:



Selecting any of these will call up a dialog box. (The first time you open a dialog box there can be a considerable delay, but it is shorter the next time.) Selecting Tway graph (scatterplot, line, etc.) from the Graphics menu displays the following:




You can make selections from each tab and then click Submit or OK. (Submit leaves the dialog box open, whereas OK closes it before generating the graph.) The dialog box will translate your options into the commands Stata uses to draw the graph. These commands are echoed to the Results window, while the graph appears in a Graph window. Next we tweak the options until we have the graph the way we want it. Then we copy the command from the Results window and paste it into a do-file, so that we can reproduce the graph later. We can also edit the do-file to modify the graph.

In the rest of this section, we describe the basic syntax for Stata graphics, because it is helpful to understand how this syntax works even if you ultimately use dialog boxes to do the bulk of the work. We focus on plots of one or more outcomes against a single explanatory variable. For this, we use the commands `graph twoway scatter` and `graph twoway connected`. (In later chapters, we will introduce a few other types of graphs as they are needed.) Namely, we consider plots of one or more outcomes against an independent variable using the command `graph twoway`. The syntax of this command has the form:

`graph twoway plottype ...`

The *Stata Graphics Reference Manual* lists 38 different *plottypes* for the `graph twoway` command. Since we discuss only two (`scatter` and `connected`) *plottypes* here, interested readers are encouraged to consult the *Graphics Reference Manual* or to type `help graph` for more information.<sup>5</sup>

Graphs that you create in Stata are drawn in their own window, which should appear on top of the four windows we discussed above. If the Graph window is hidden, you

can bring it to the front by clicking on . You can make the Graph window larger or smaller by clicking and dragging the borders.

### 2.16.1 graph command

The type of graph that we use most often shows how the predicted probability of observing a given outcome changes as a continuous variable changes over a specified range. For example, in chapter 4 we show you how to compute the predicted probability of a woman being in the labor force according to the number of children she has and the family's income. In later chapters, we show you how to compute these predictions, but for now you can simply load them into memory with the command `use lfpgraph2, clear`. The variable `income` is family income measured in thousands of dollars, excluding any contribution made by the woman of the household, whereas the next three variables show the predicted probabilities of working for a woman who has no children under six (`kid0p1`), one child under six (`kid1p1`), or two children under six (`kid2p1`). Because there are only 11 values, we can easily list them:

### 2.16.1 graph command

```
. use http://www.stata-press.com/data/lf2/lfpgraph2, clear
(Sample predictions to plot.)
. list income kid0p1 kid1p1
```

	income	kid0p1	kid1p1
1.	10	.7330963	.3887608
2.	18	.6758616	.3256128
3.	26	.6128353	.2682211
4.	34	.54579	.2176799
5.	42	.477042	.1743927
6.	50	.409153	.1381929
7.	58	.3445598	.1085196
8.	66	.285241	.0845925
9.	74	.2325117	.065553
10.	82	.18698	.0505621
11.	90	.1486378	.0388569

We see that as annual income increases the predicted probability of being in the labor force decreases. Also by looking across any row, we see that for a given level of income the probability of being in the labor force decreases as the number of young children increases. We want to display these patterns graphically.

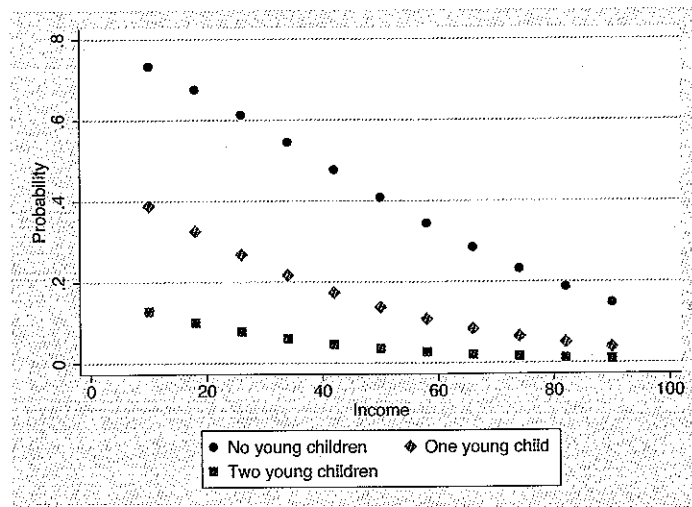
`graph twoway scatter` can be used to draw a scatterplot in which the values of one or more  $y$ -variables are plotted against values of an  $x$ -variable. Here `income` is the  $x$ -variable, and the predicted probabilities `kid0p1`, `kid1p1`, and `kid2p1` are the  $y$ -variables. Thus for each value of  $x$ , we have three values of  $y$ . In making scatterplots with `graph twoway scatter`, the  $y$ -variables are listed first, and the  $x$ -variable is listed last. If we type

```
. graph twoway scatter kid0p1 kid1p1 kid2p1 income, ytitle(Probability)
```

we obtain the following graph:

(Continued on next page)

5. We also use a third *plottype* called `rarea`, but we will postpone describing that until later.



Our simple scatterplot shows the pattern of decreasing probabilities as income or number of children increases.

This simple command produces a reasonable first graph. Indeed, Stata's default settings are usually a good place to begin. Even so, we can make a more effective graph by including more options. Below we focus only on adding titles and changing the labels for the axes. Remember that if you want to change other aspects of the graph, you will almost certainly be able to get what you want if you find the right options. A slightly more detailed syntax<sup>6</sup> for `graph twoway` is

```
graph twoway plot1 [plot2] ... [plotN] [if] [in] [, twoway-options]
```

where `ploti` is defined to be

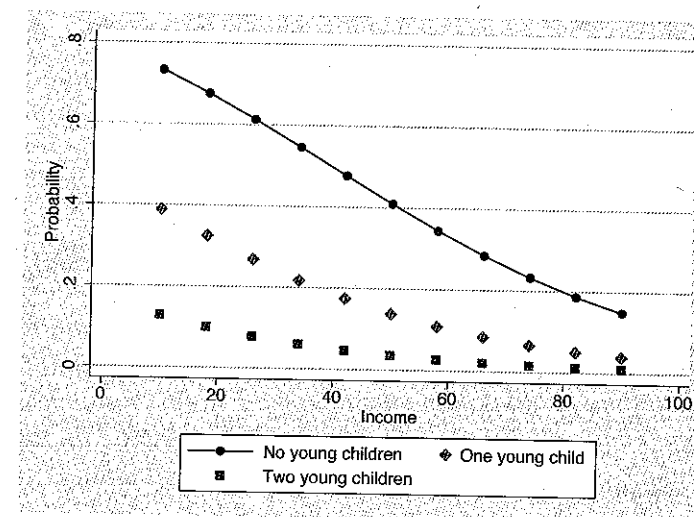
```
[ ( [ plotype varlist, [ title("string") subtitle("string") ytitle("string")
  xtitle("string") caption("string") xlabel(values) ylabel(values)
  other-options ] ) ]
```

This syntax highlights the fact that it is possible to put multiple plots on the same graph.<sup>7</sup> The plots can be of different plot types. For instance, suppose that we wanted the symbols in the plot corresponding to "No young children" to be connected. This plot type is called `connected`. For example,

6. The syntax presented here is incomplete. We wish only to explain the elements that we have found ourselves using in presenting analyses like those in this book. See the *Stata Graphics Reference Manual* for more information.

7. The parentheses are used to separate the different plots when there are multiple plots. When there is only one plot, the parentheses are not required.

```
. graph twoway (connected kid0p1 income)
> (scatter kid1p1 kid2p1 income), ytitle(Probability)
```



With the exception of the title on the *y*-axis, the default choices for the symbols, line styles, etc., all are all quite nice. Stata made these choices within the context of an overall look or scheme. For example, because our book is published in monochrome, we wanted our graphs to be drawn in monochrome. The *Graphics Reference Manual* describes how they could be changed. (Type `help schemes` in Stata for the latest information about the available schemes.) Users can choose the overall look of their graphs by setting the scheme. In writing this book, we simply included the line

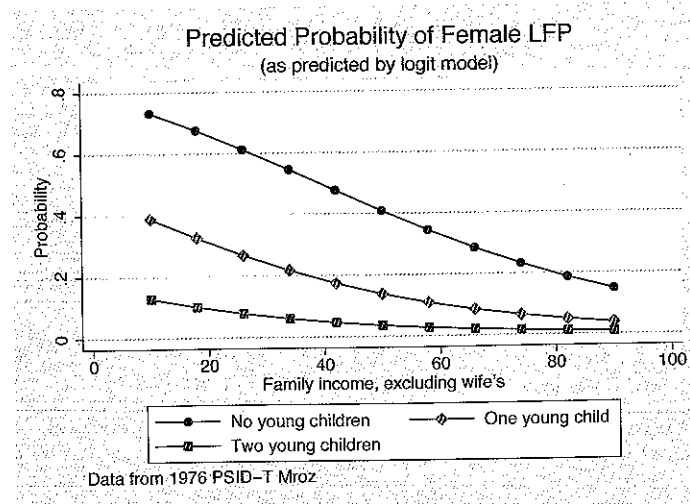
```
. set scheme sj
```

at the top of our do-files.

### Adding titles

Now we provide a quick introduction that shows how to set the five titles that we often wish to change: (1) overall title, (2) overall subtitle, (3) *y*-axis title, (4) *x*-axis title, and (5) graph caption. The options for setting each of these five titles are in the syntax diagram above. The command and graph below illustrate how we might use each of these titles.

```
. graph twoway (connected kid0p1 kid1p1 kid2p1 income),
> ytitle("Probability")
> title("Predicted Probability of Female LFP")
> subtitle("as predicted by logit model")
> xtitle("Family income, excluding wife's")
> caption("Data from 1976 PSID-T Mroz")
```



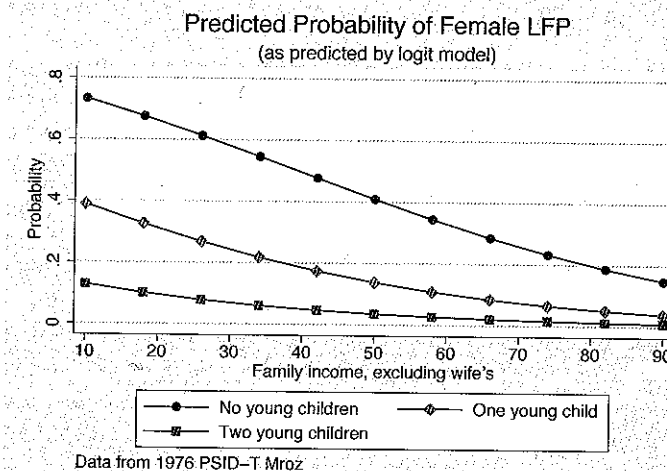
This graph is much more effective in illustrating that the probability of a woman being in the labor force declines as family income increases, and that the differences in predicted probabilities between women with no young children and those with one or two young children are greatest at the lowest levels of income.

### Labeling the axes

Even though the defaults are nice, it is common to want to change the labeling of the ticks on the  $x$ -axis or  $y$ -axis. The `ylabel()` and `xlabel()` options allow users to specify either a rule or a set of values for the tick marks. A rule is simply a compact way to specify a list of values.

Let's consider specifying a list of values first. A common change is to alter the frequency or range of tick marks. This change can also be made with the `xlabel()` and `ylabel()` options. Suppose that we liked the frequency of the ticks on the  $x$ -axis but wanted to restrict the range to [10, 90]. We make this change in the command

```
. graph twoway (connected kid0p1 kid1p1 kid2p1 income),
> ytitle("Probability")
> title("Predicted Probability of Female LFP")
> subtitle("(as predicted by logit model)")
> xtitle("Family income, excluding wife's")
> caption("Data from 1976 PSID-T Mroz")
> xlabel(10 20 30 40 50 60 70 80 90)
```



We could have obtained the same graph by specifying a rule for a new set of  $x$ -axis values. Although there are several ways to specify a rule,<sup>8</sup> we find the form `#1(#2)#3` most useful. In this form, the user specifies three numbers: `#1` specifies the beginning of the sequence of values, `#2` specifies the increment between each value, and `#3` specifies the maximum value. For instance, instead of specifying the option

```
xlabel(10 20 30 40 50 60 70 80 90)
```

in the previous graph, we could have specified

```
xlabel(10(10)90)
```

to obtain the same graph.

### Naming graphs

When you create a graph, it is displayed in a Graph window and is also saved in memory. Accordingly, when you close the Graph window, you can redisplay the graph with the command `graph display`. By default, a graph is stored in memory with the name `Graph`, and this graph is overwritten whenever you generate a new graph. If you want to store more than one graph in memory (this is not the same as storing them to disk, which is discussed in the next section), you need to use the `name()` option. For example,

```
. scatter y x, name(example1)
```

stores the scatterplot for  $y$  against  $x$  in memory with the name `example1`. Then

```
. scatter z x, name(example2)
```

8. Type `help axis_label_options` for other ways to specify a rule.



will save the scatterplot for *z* against *x* with the name `example2`. Stata displays each named graph in its own window, and multiple Graph windows can be displayed simultaneously. Even if you have closed the Graph windows, you could redisplay the graphs with the commands

```
. graph display example1
. graph display example2
```

In do-files, you might want to use `name(example1, replace)` so that the program will overwrite graph `example1` if it exists.

### Saving graphs

Graphs can be either saved in a file or stored in memory. When a graph is saved to a file, it remains there until the file is erased. When a graph is stored in memory, it remains there until you exit Stata or drop the graph from memory. Specifying `saving(filename, replace)` saves the graph to a file in Stata's proprietary format (indicated by the suffix `.gph`) in the working directory. Including `replace` tells Stata to overwrite a file with that name if it exists. Specifying `name(name, replace)` stores the graph in memory. The `replace` option tells Stata to replace any existing graphs stored under that name.

Graphs must be either saved to files or stored in memory before you can combine them. For example, if we were to later need the graph we just created, we could store it in memory under the name `graph1` with the command

```
. graph twoway (connected kid0p1 kid1p1 kid2p1 income),
> ytitle("Probability")
> title("Predicted Probability of Female LFP")
> subtitle("(as predicted by logit model)")
> xtitle("Family income, excluding wife's")
> caption("Data from 1976 PSID-T Mroz")
> xlabel(10(10)90) name(graph1, replace)
```


**Tip: Exporting graphs to other programs** If you are using Windows or Macintosh and want to export graphs to another program, such as a word processor, we find that it works best to save them as a Windows Enhanced Metafile (EMF) in Windows, or as a Macintosh PICT file in Mac OS X. In Windows with the graph currently displayed in the Graph window, you can export it in the EMF format with the command `graph export filename.emf`. In Mac OS X with the graph currently displayed in the Graph window, you can export it in the PICT format with the command `graph export filename.pct`. You cannot export a graph to the `.pct` format in Windows, nor can you export a graph to the `.emf` format in Mac OS X; the formats are exclusive to their respective operating systems. If the file is already saved in `.gph` format, you can export it to either `.emf` or `.pct` format in two steps. First, redisplay the graph with the command `graph use filename`. Then export the graph with the command `graph export filename.emf` or `graph export filename.pct`. The `replace` option can be used with `graph export` to automatically overwrite a graph of the same name, which is useful in do-files.

## 2.16.2 Displaying previously drawn graphs

There are several commands used for manipulating graphs that have been previously drawn and saved to memory or disk. `graph dir` lists graphs previously saved in memory or to a file in the current working directory. `graph use` copies a graph stored in a file into memory and displays it. `graph display` redisplay a graph stored in memory.

## 2.16.3 Printing graphs

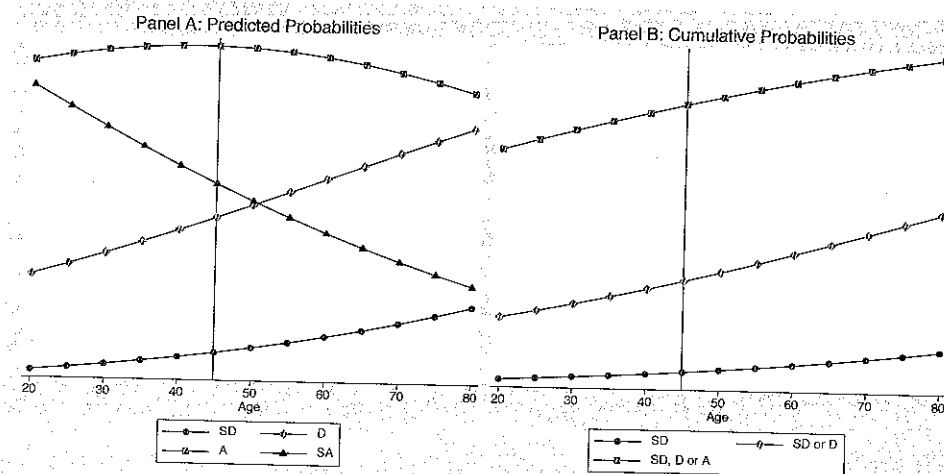
It is easiest to print a graph once it is in the Graph window. When a graph is in the Graph window, you can print it by selecting File→Print→Graph(*graphname*) from the

menus or by clicking on . You can also print a graph in the Graph window with the command `graph print`. To print a graph saved to memory or disk, first use `graph use` or `graph display` to redisplay it, and then print it with the command `graph print`.

## 2.16.4 Combining graphs

Multiple graphs that have been saved can be combined. This is useful, for example, when you want to place two graphs side by side or stack them. In chapter 5, we will find it useful to combine two graphs. Here we use two of the graphs that we discuss in detail in section 5.8.6 to illustrate `graph combine`. When we originally drew the graphs, we saved them in memory under the names `graph1` and `graph2`. Now we use `graph combine` to put the two graphs side by side in one Graph window.

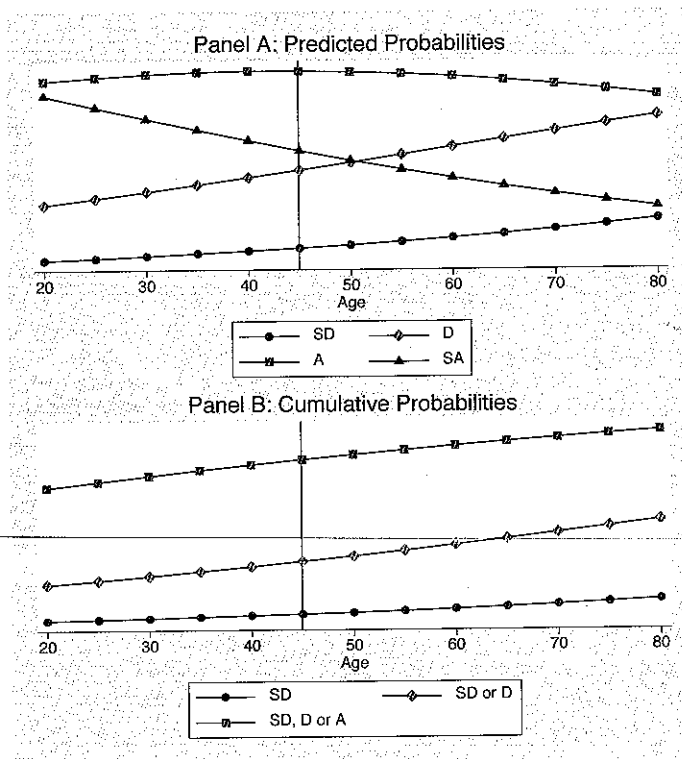
```
. graph combine graph1 graph2, imargin(small)
```



This combined graph is not as effective as one in which the graphs are stacked. The trick is to understand that when multiple graphs are combined, Stata divides the Graph window into an array. The `rows()` and `cols()` options can be used to set the number of rows and columns in the array. Of course, as with most aspects of a graph, the *Graphics Reference Manual* describes how almost any part of the combined graph can be changed.<sup>9</sup> By default, the individual graphs are allocated over the rows in the order in which the filenames are listed in the `graph combine` command.

To display the graphs stacked vertically, specify the `col()` option:

```
. graph combine graph1 graph2, iscale(*.9) imargin(small) col(1)
```



As we described earlier, `graph export` can be used to save the graph as a Windows Enhanced Metafile that can be imported to a word processor or other program. More details on combining graphs can be found in the *Stata Graphics Reference Manual*.

9. In particular, see *Advanced use* in [G] `graph combine` for a rather impressive example.

## 2.17 A brief tutorial

This tutorial uses the `science2.dta` dataset that is available from the book's web site. You can use your own dataset as you work through this tutorial, but you will need to change some of the commands to correspond to the variables in your data. In addition to our tutorial, the *User's Guide* provides a wealth of information for new users.

### Opening a log

The first step is to open a log file for recording your results. Remember that all commands are case sensitive. The commands are listed with a period in front, but you do *not* type the period:

```
. capture log close
. log using tutorial, text

-----
. log: d:\spostdata\tutorial.log
. log type: text
. opened on: 26 Sep 2005, 11:18:15
```

### Loading the data

We assume that `science2.dta` is in your working directory. `clear` tells Stata to “clear out” any existing data from memory before loading the new dataset:

```
. use http://www.stata-press.com/data/lf2/science2, clear
(Note that some of the variables have been artificially constructed.)
```

The message after loading the data reflects that this dataset was created for teaching. Although most of the variables contain real information, some variables have been artificially constructed.

(Continued on next page)

## Examining the dataset

describe gives information about the dataset.

```
. describe
Contains data from science2.dta
obs:          308
Note that some of the variables
have been artificially
constructed.
10 Mar 2001 05:51
vars:         35
size:         17,556 (98.3% of memory free)
(_dta has notes)
```

variable name	storage type	display format	value label	variable label
id	float	%9.0g		ID Number.
cit1	int	%9.0g		Citations: PhD yr -1 to 1.
cit3	int	%9.0g		Citations: PhD yr 1 to 3.
cit6	int	%9.0g		Citations: PhD yr 4 to 6.
cit9	int	%9.0g		Citations: PhD yr 7 to 9.
enrol	byte	%9.0g		Years from BA to PhD.
fel	float	%9.0g		Fellow or PhD prestige.
felclass	byte	%9.0g	prstlb	* Fellow or PhD prestige class.
fellow	byte	%9.0g	fellbl	Postdoctoral fellow: 1=y,0=n.
female	byte	%9.0g	femlbl	Female: 1=female,0=male.
job	float	%9.0g		Prestige of 1st univ job.
jobclass	byte	%9.0g	prstlb	* Prestige class of 1st job.
mcit3	int	%9.0g		Mentor's 3 yr citation.
mcitt	int	%9.0g		Mentor's total citations.
mmale	byte	%9.0g	malelb	Mentor male: 1=male,0=female.
mnas	byte	%9.0g	naslb	Mentor NAS: 1=yes,0=no.
mpub3	byte	%9.0g		Mentor's 3 year publications.
nopub1	byte	%9.0g	nopublb	1=No pubs PhD yr -1 to 1.
nopub3	byte	%9.0g	nopublb	1=No pubs PhD yr 1 to 3.
nopub6	byte	%9.0g	nopublb	1=No pubs PhD yr 4 to 6.
nopub9	byte	%9.0g	nopublb	1=No pubs PhD yr 7 to 9.
phd	float	%9.0g		Prestige of Ph.D. department.
phdclass	byte	%9.0g	prstlb	* Prestige class of Ph.D. dept.
pub1	byte	%9.0g		Publications: PhD yr -1 to 1.
pub3	byte	%9.0g		Publications: PhD yr 1 to 3.
pub6	byte	%9.0g		Publications: PhD yr 4 to 6.
pub9	byte	%9.0g		Publications: PhD yr 7 to 9.
work	byte	%9.0g	worklbl	Type of first job.
workadm	byte	%9.0g	wadmnlb	Admin: 1=yes; 0=no.
worktch	byte	%9.0g	wtchlb	* Teaching: 1=yes; 0=no.
workuniv	byte	%9.0g	wunivlb	* Univ Work: 1=yes; 0=no.
wt	byte	%9.0g		
faculty	byte	%9.0g	fac1bl	1=Faculty in University
jobrank	byte	%9.0g	job1bl	Rankings of University Job.
totpub	byte	%9.0g		Total Pubs in 9 Yrs post-Ph.D.

\* indicated variables have notes

Sorted by:

## Examining individual variables

A series of commands gives us information about individual variables. You can use whichever command you prefer, or all of them.

```
. summarize work
Variable | Obs Mean Std. Dev. Min Max
-----+-----
work | 302 2.062914 1.37829 1 5

. tabulate work, missing
Type of first job. | Freq. Percent Cum.
-----+-----
FacUniv | 160 51.95 51.95
ResUniv | 53 17.21 69.16
ColTch | 26 8.44 77.60
IndRes | 36 11.69 89.29
Admin | 27 8.77 98.05
. | 6 1.95 100.00
Total | 308 100.00
```

```
. codebook work
```

```
work Type of first job.
```

```
type: numeric (byte)
label: worklbl
range: [1,5]
unique values: 5
units: 1
missing : 6/308

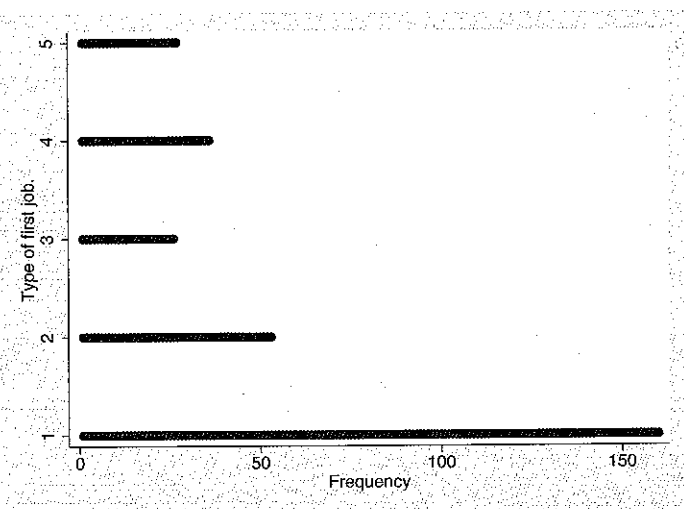
tabulation: Freq. Numeric Label
160 1 FacUniv
53 2 ResUniv
26 3 ColTch
36 4 IndRes
27 5 Admin
6
```

## Graphing variables

Graphs are also useful for examining data. The command

```
. dotplot work
```

creates the following graph:



### Saving graphs

To save the above graph as a Windows Enhanced Metafile, type

```
. graph export myname.emf, replace
(file d:\spostdata\myname.emf written in Windows Enhanced Metafile format)
```

### Adding comments

To add comments to your output, which allows you to document your command files, type \* at the beginning of each comment. The comments are listed in the log file:

```
. * saved graph as work.emf
```

### Creating a dummy variable

Now let's make a dummy variable with faculty in universities coded 1 and all others coded 0. The command `gen isfac = (work==1) if work<.` generates `isfac` as a dummy variable where `isfac` equals 1 if `work` is 1, else 0. The statement `if work<.` makes sure that missing values are kept as missing in the new variable.

```
. generate isfac = (work==1) if work<.
(6 missing values generated)
```

Six missing values were generated because `work` contained six missing observations.

### Checking transformations

One way to check transformations is with a table. In general, it is best to look at the missing values, which requires the `missing` option:

```
. tabulate isfac work, missing
```

isfac	Type of first job.					Total
	FacUniv	ResUniv	ColTch	IndRes	Admin	
0	0	53	26	36	27	142
1	160	0	0	0	0	160
.	0	0	0	0	0	6
Total	160	53	26	36	27	308

isfac	Type of first job.	
	.	Total
0	0	142
1	0	160
.	6	6
Total	6	308

### Labeling variables and values

For many of the regression commands, value labels for the dependent variable are essential. We start by creating a variable label, then create `isfac` to store the value labels, and finally assign the value labels to the variable `isfac`:

```
. label variable isfac "1=Faculty in University"
. label define isfac 0 "NotFac" 1 "Faculty"
. label values isfac isfac
```

Then we can get labeled output:

```
. tabulate isfac
```

1=Faculty in University	Freq.	Percent	Cum.
NotFac	142	47.02	47.02
Faculty	160	52.98	100.00
Total	302	100.00	

### Creating an ordinal variable

The prestige of graduate programs is often referred to using the categories of adequate, good, strong, and distinguished. Here we create such an ordinal variable from

the continuous variable for the prestige of the first job. missing tells Stata to show cases with missing values.

```
. tab job, missing
```

Prestige of 1st univ job.	Freq.	Percent	Cum.
1.01	1	0.32	0.32
1.2	1	0.32	0.65
1.22	1	0.32	0.97
1.32	1	0.32	1.30
1.37	1	0.32	1.62
<i>(output omitted)</i>			
3.97	6	1.95	48.38
4.18	2	0.65	49.03
4.42	1	0.32	49.35
4.5	6	1.95	51.30
4.69	5	1.62	52.92
.	145	47.08	100.00
Total	308	100.00	

The recode command makes it easy to group the categories from job. Of course, we then label the variable:

```
. generate jobprst = job
(145 missing values generated)
. recode jobprst . = 1/1.99=1 2/2.99=2 3/3.99=3 4/5=4
(jobprst: 162 changes made)
. label variable jobprst "Rankings of University Job"
. label define prstlbl 1 "Adeq" 2 "Good" 3 "Strong" 4 "Dist"
. label values jobprst prstlbl
```

Here is the new variable (we use the missing option so that missing values are included in the tabulation):

```
. tabulate jobprst, missing
```

Rankings of University Job	Freq.	Percent	Cum.
Adeq	31	10.06	10.06
Good	47	15.26	25.32
Strong	71	23.05	48.38
Dist	14	4.55	52.92
.	145	47.08	100.00
Total	308	100.00	

## Combining variables

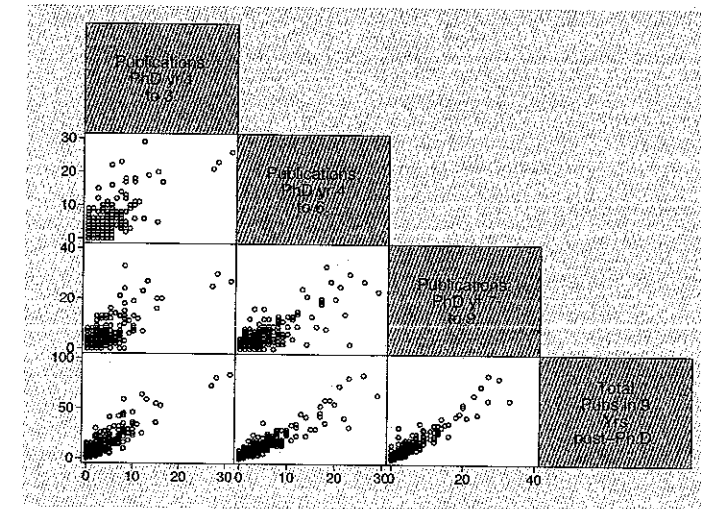
Now we create a new variable by summing existing variables. If we add pub3, pub6, and pub9, we can obtain the scientist's total number of publications over the 9 years following receipt of the Ph.D.

```
. generate pubsum = pub3 + pub6 + pub9
. label variable pubsum "Total Pubs in 9 Yrs post-Ph.D."
. summarize pub3 pub6 pub9 pubsum
```

Variable	Obs	Mean	Std. Dev.	Min	Max
pub3	308	3.185065	3.908752	0	31
pub6	308	4.165584	4.780714	0	29
pub9	308	4.512987	5.315134	0	33
pubsum	308	11.86364	12.77623	0	84

A scatterplot matrix graph can be used to plot all pairs of variables simultaneously:

```
. graph matrix pub3 pub6 pub9 pubsum, half msymbol(smcircle_hollow)
```



## Saving the new data

After you make changes to your dataset, save the data with a new filename:

```
. save sciwork, replace
file sciwork.dta saved
```

## Closing the log file

Last, we need to close the log file so that we can refer to it in the future.

```
. log close
  log: d:\spostdata\tutorial.log
  log type: text
  closed on: 26 Sep 2005, 11:18:27
```

## A batch version

If you have read section 2.9, you know that a better idea is to create a batch (do-) file, perhaps called `tutorial.do`.<sup>10</sup>

```
// batch version of tutorial do-file
version 9
set scheme sj
set more off
capture log close
log using ch2tutorial, replace

// loading the data
use http://www.stata-press.com/data/lf2/science2, clear

// examining the dataset
describe

// examining individual variables
summarize work
tabulate work, missing
codebook work

// graphing variables
dotplot work

// saving graph
graph export O2dotplot2.emf, replace

// creating a dummy variable
gen isfac = (work==1) if work<.

// checking transformations
tabulate isfac work, missing

// labeling variables and values
label variable isfac "1=Faculty in University"
label define isfac 0 "NotFac" 1 "Faculty"
label values isfac isfac
tabulate isfac

// creating an ordinal variable
tabulate job, missing
generate jobprst=job
recode jobprst .=. 1/1.99=1 2/2.99=2 3/3.99=3 4/5=4
label variable jobprst "Rankings of University Job"
label define prstlbl 1 "Adeq" 2 "Good" 3 "Strong" 4 "Dist"
label values jobprst prstlbl
tabulate jobprst, missing
```

10. If you download this file from our web site, it is called `st9ch2tutorial.do`.

```
// combining variables
generate pubsum = pub3 + pub6 + pub9
label variable pubsum "Total Pubs in 9 Yrs post-Ph.D."
summarize pub3 pub6 pub9 pubsum

// graphing variables
graph matrix pub3 pub6 pub9 pubsum, half msymbol(smcircle_hollow)

// saving graph
graph export O2matrix.emf, replace

// saving the new data
note: temporary dataset for st9ch2tutorial.do
save sciwork, replace

// close the log
log close
```

Then type `do tutorial` in the Command window or select `File→Do...` from the menu.