Microeconometrics Using Stata

Microeconometrics Using Stata

A. COLIN CAMERON
Department of Economics
University of California
Davis, CA

PRAVIN K. TRIVEDI
Department of Economics
Indiana University
Bloomington, IN

A Stata Press Publication
StataCorp LP
College Station, Texas

Published by Stata Press, 4905 Lakeway Drive, College Station, Texas 77845
Typeset in LaTeX 2ε
Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN-10: 1-59718-048-3
ISBN-13: 978-1-59718-048-1

Contents

Tables

Figures

Preface

This book explains how an econometrics computer package, Stata, can be used to perform regression analysis of cross-section and panel data. The term microeconometrics is used in the book title because the applications are to economics-related data and because the coverage includes methods such as instrumental-variables regression that are emphasized more in economics than in some other areas of applied statistics. However, many issues, models, and methodologies discussed in this book are also relevant to other social sciences.

The main audience is graduate students and researchers. For them, this book can be used as an adjunct to our own *Microeconometrics: Methods and Applications* (Cameron and Trivedi 2005), as well as to other graduate-level texts such as Greene (2008) and Wooldridge (2002). By comparison to these books, we present little theory and instead emphasize practical aspects of implementation using Stata. More advanced topics we cover include quantile regression, weak instruments, nonlinear optimization, bootstrap methods, nonlinear panel-data methods, and Stata's matrix programming language, Mata.

At the same time, the book provides introductions to topics such as ordinary least-squares regression, instrumental-variables estimation, and logit and probit models so that it is suitable for use in an undergraduate econometrics class, as a complement to an appropriate undergraduate-level text. The following table suggests sections of the book for an introductory class, with the caveat that in places formulas are provided using matrix algebra.

Stata basics	Chapter 1.1–1.4
Data management	Chapter 2.1–2.4, 2.6
OLS	Chapter 3.1–3.6
Simulation	Chapter 4.6–4.7
GLS (heteroskedasticity)	Chapter 5.3
Instrumental variables	Chapter 6.2–6.3
Linear panel data	Chapter 8
Logit and probit models	Chapter 14.1–14.4
Tobit model	Chapter 16.1–16.3

Although we provide considerable detail on Stata, the treatment is by no means complete. In particular, we introduce various Stata commands but avoid detailed listing and description of commands as they are already well documented in the Stata manuals

and online help. Typically, we provide a pointer and a brief discussion and often an example.

As much as possible, we provide template code that can be adapted to other problems. Keep in mind that to shorten output for this book, our examples use many fewer regressors than necessary for serious research. Our code often suppresses intermediate output that is important in actual research, because of extensive use of command quietly and options nolog, nodots, and noheader. And we minimize the use of graphs compared with typical use in exploratory data analysis.

We have used Stata 10, including Stata updates.[1] Instructions on how to obtain the datasets and the do-files used in this book are available on the Stata Press web site at http://www.stata-press.com/data/mus.html. Any corrections to the book will be documented at http://www.stata-press.com/books/mus.html.

We have learned a lot of econometrics, in addition to learning Stata, during this project. Indeed, we feel strongly that an effective learning tool for econometrics is hands-on learning by opening a Stata dataset and seeing the effect of using different methods and variations on the methods, such as using robust standard errors rather than default standard errors. This method is beneficial at all levels of ability in econometrics. Indeed, an efficient way of familiarizing yourself with Stata's leading features might be to execute the commands in a relevant chapter on your own dataset.

We thank the many people who have assisted us in preparing this book. The project grew out of our 2005 book, and we thank Scott Parris for his expert handling of that book. Juan Du, Qian Li, and Abhijit Ramalingam carefully read many of the book chapters. Discussions with John Daniels, Oscar Jorda, Guido Kuersteiner, and Doug Miller were particularly helpful. We thank Deirdre Patterson for her excellent editing and Lisa Gilmore for managing the LaTeX formatting and production of this book. Most especially, we thank David Drukker for his extensive input and encouragement at all stages of this project, including a thorough reading and critique of the final draft, which led to many improvements in both the econometrics and Stata components of this book. Finally, we thank our respective families for making the inevitable sacrifices as we worked to bring this multiyear project to completion.

Davis, CA
Bloomington, IN
October 2008

A. Colin Cameron
Pravin K. Trivedi

1. To see whether you have the latest update, type update query. For those with earlier versions of Stata, some key changes are the following: Stata 9 introduced the matrix programming language, Mata. The syntax for Stata 10 uses the vce(robust) option rather than the robust option to obtain robust standard errors. A mid-2008 update of version 10 introduced new random-number functions, such as runiform() and rnormal().

1 Stata basics

This chapter provides some of the basic information about issuing commands in Stata. Sections 1.1–1.3 enable a first-time user to begin using Stata interactively. In this book, we instead emphasize storing these commands in a text file, called a Stata do-file, that is then executed. This is presented in section 1.4. Sections 1.5–1.7 present more-advanced Stata material that might be skipped on a first reading.

The chapter concludes with a summary of some commonly used Stata commands and with a template do-file that demonstrates many of the tools introduced in this chapter. Chapters 2 and 3 then demonstrate many of the Stata commands and tools used in applied microeconometrics. Additional features of Stata are introduced throughout the book and in appendices A and B.

1.1 Interactive use

Interactive use means that Stata commands are initiated from within Stata.

A graphical user interface (GUI) for Stata is available. This enables almost all Stata commands to be selected from drop-down menus. Interactive use is then especially easy, as there is no need to know in advance the Stata command.

All implementations of Stata allow commands to be directly typed in; for example, entering `summarize` yields summary statistics for the current dataset. This is the primary way that Stata is used, as it is considerably faster than working through drop-down menus. Furthermore, for most analyses, the standard procedure is to aggregate the various commands needed into one file called a do-file (see section 1.4) that can be run with or without interactive use. We therefore provide little detail on the Stata GUI.

For new Stata users, we suggest entering Stata, usually by clicking on the Stata icon, opening one of the Stata example datasets, and doing some basic statistical analysis. To obtain example data, select **File > Example Datasets...**, meaning from the **File** menu, select the entry **Example Datasets...**. Then click on the link to **Example datasets installed with Stata**. Work with the dataset `auto.dta`; this is used in many of the introductory examples presented in the Stata documentation. First, select **describe** to obtain descriptions of the variables in the dataset. Second, select **use** to read the dataset into Stata. You can then obtain summary statistics either by typing `summarize` in the Command window or by selecting **Statistics > Summaries, tables, and tests > Summary and descriptive statistics > Summary statistics**. You can run a simple regression by typing `regress mpg weight` or by selecting **Statistics**

> Linear models and related > Linear regression and then using the drop-down lists in the **Model** tab to choose `mpg` as the dependent variable and `weight` as the independent variable.

The Stata manual [GS] *Getting Started with Stata* is very helpful, especially [GS] **3 A sample session**, which uses typed-in commands, and [GS] **4 The Stata user interface**.

The extent to which you use Stata in interactive mode is really a personal preference. There are several reasons for at least occasionally using interactive mode. First, it can be useful for learning how to use Stata. Second, it can be useful for exploratory analysis of datasets because you can see in real time the effect of, for example, adding or dropping regressors. If you do this, however, be sure to first start a session log file (see section 1.4) that saves the commands and resulting output. Third, you can use `help` and related commands to obtain online information about Stata commands. Fourth, one way to implement the preferred method of running do-files is to use the Stata Do-file Editor in interactive mode.

Finally, components of a given version of Stata, such as version 10, are periodically updated. Entering `update query` determines the current update level and provides the option to install official updates to Stata. You can also install user-written commands in interactive mode once the relevant software is located using, for example, the `findit` command.

1.2 Documentation

Stata documentation is extensive; you can find it in hard copy, in Stata (online), or on the web.

1.2.1 Stata manuals

For first-time users, see [GS] *Getting Started with Stata*. The most useful manual is [U] *User's Guide*. Entries within manuals are referred to using shorthand such as [U] **11.1.4 in range**, which denotes section 11.1.4 of [U] *User's Guide* on the topic **in range**.

Many commands are described in [R] *Base Reference Manual*, which spans three volumes. For version 10, these are A–H, I–P, and Q–Z. Not all Stata commands appear here, however, because some appear instead in the appropriate topical manual. These topical manuals are [D] *Data Management Reference Manual*, [G] *Graphics Reference Manual*, [M] *Mata Reference Manual* (two volumes), [MV] *Multivariate Statistics Reference Manual*, [P] *Programming Reference Manual*, [ST] *Survival Analysis and Epidemiological Tables Reference Manual*, [SVY] *Survey Data Reference Manual*, [TS] *Time-Series Reference Manual*, and [XT] *Longitudinal/Panel-Data Reference Manual*. For example, the `generate` command appears in [D] **generate** rather than in [R].

For a complete list of documentation, see [U] **1 Read this—it will help** and also [I] *Quick Reference and Index*.

1.2.2 Additional Stata resources

The *Stata Journal* (SJ) and its predecessor, the *Stata Technical Bulletin* (STB), present examples and code that go beyond the current installation of Stata. SJ articles over three years old and all STB articles are available online from the Stata web site at no charge. You can find this material by using various Stata help commands given later in this section, and you can often install code as a free user-written command.

The Stata web site has a lot of information. This includes a summary of what Stata does. A good place to begin is http://www.stata.com/support/. In particular, see the answers to frequently asked questions (FAQs).

The University of California–Los Angeles web site
http://www.ats.ucla.edu/STAT/stata/ provides many Stata tutorials.

1.2.3 The help command

Stata has extensive help available once you are in the program.

The `help` command is most useful if you already know the name of the command for which you need help. For example, for help on the `regress` command, type

```
. help regress
(output omitted)
```

Note that here and elsewhere the dot (`.`) is not typed in but is provided to enable distinction between Stata commands (preceded by a dot) and subsequent Stata output, which appears with no dot.

The `help` command is also useful if you know the class of commands for which you need help. For example, for help on functions, type

```
. help function
(output omitted)
```

(*Continued on next page*)

Often, however, you need to start with the basic `help` command, which will open the Viewer window shown in figure 1.1.

```
. help
```

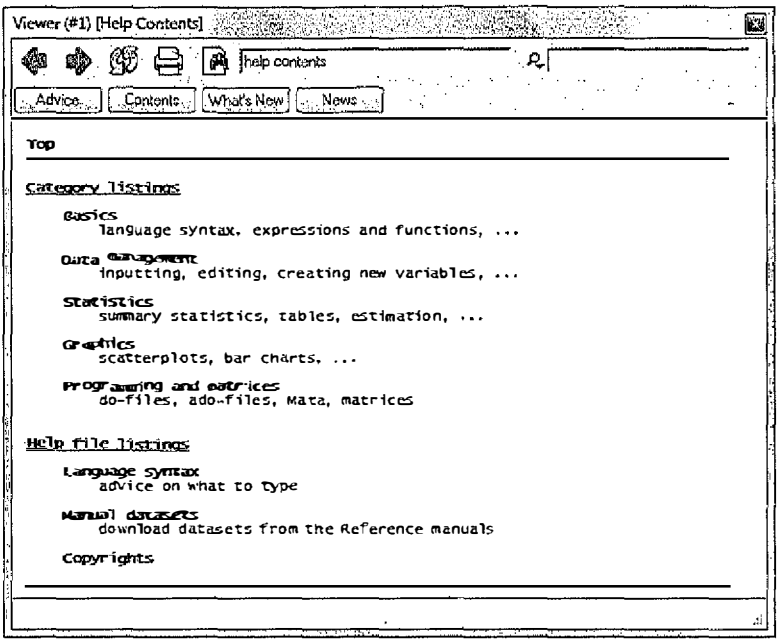

Figure 1.1. Basic `help` contents

For further details, click on a category and subsequent subcategories.

For help with the Stata matrix programming language, Mata, add the term `mata` after `help`. Often, for Mata, it is necessary to start with the very broad command

```
. help mata
(output omitted)
```

and then narrow the results by selecting the appropriate categories and subcategories.

1.2.4 The search, findit, and hsearch commands

There are several search-related commands that do not require knowledge of command names.

For example, the `search` command does a keyword search. It is especially useful if you do not know the Stata command name or if you want to find the many places that

a command or method might be used. The default for search is to obtain information from official help files, FAQs, examples, the SJ, and the STB but not from Internet sources. For example, for ordinary least squares (OLS) the command

```
. search ols
  (output omitted)
```

finds references in the manuals [R], [MV], [SVY], and [XT]; in FAQs; in examples; and in the SJ and the STB. It also gives help commands that you can click on to get further information without the need to consult the manuals. The net search command searches the Internet for installable packages, including code from the SJ and the STB.

The findit command provides the broadest possible keyword search for Stata-related information. You can obtain details on this command by typing help findit. To find information on weak instruments, for example, type

```
. findit weak instr
  (output omitted)
```

This finds joint occurrences of keywords beginning with the letters "weak" and the letters "instr".

The search and findit commands lead to keyword searches only. A more detailed search is not restricted to keywords. For example, the hsearch command searches all words in the help files (extension .sthlp or .hlp) on your computer, for both official Stata commands and user-written commands. Unlike the findit command, hsearch uses a whole word search. For example,

```
. hsearch weak instrument
  (output omitted)
```

actually leads to more results than hsearch weak instr.

The hsearch command is especially useful if you are unsure whether Stata can perform a particular task. In that case, use hsearch first, and if the task is not found, then use findit to see if someone else has developed Stata code for the task.

1.3 Command syntax and operators

Stata command syntax describes the rules of the Stata programming language.

1.3.1 Basic command syntax

The basic command syntax is almost always some subset of

[*prefix*:] *command* [*varlist*] [= *exp*] [*if*] [*in*] [*weight*]
[using *filename*] [, *options*]

The square brackets denote qualifiers that in most instances are optional. Words in the typewriter font are to be typed into Stata like they appear on the page. Italicized words are to be substituted by the user, where

- *prefix* denotes a command that repeats execution of *command* or modifies the input or output of *command*,
- *command* denotes a Stata command,
- *varlist* denotes a list of variable names,
- *exp* is a mathematical expression,
- *weight* denotes a weighting expression,
- *filename* is a filename, and
- *options* denotes one or more options that apply to *command*.

The greatest variation across commands is in the available options. Commands can have many options, and these options can also have options, which are given in parentheses.

Stata is case sensitive. We generally use lowercase throughout, though occasionally we use uppercase for model names.

Commands and output are displayed following the style for Stata manuals. For Stata commands given in the text, the typewriter font is used. For example, for OLS, we use the `regress` command. For displayed commands and output, the commands have the prefix `. ` (a period followed by a space), whereas output has no prefix. For Mata commands, the prefix is a colon (`:`) rather than a period. Output from commands that span more than one line has the continuation prefix `>` (greater-than sign). For a Stata or Mata `program`, the lines within the program do not have a prefix.

1.3.2 Example: The summarize command

The `summarize` command provides descriptive statistics (e.g., mean, standard deviation) for one or more variables.

You can obtain the syntax of `summarize` by typing `help summarize`. This yields output including

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

It follows that, at the minimum, we can give the command without any qualifiers. Unlike some commands, `summarize` does not use [= *exp*] or [`using` *filename*].

As an example, we use a commonly used, illustrative dataset installed with Stata called `auto.dta`, which has information on various attributes of 74 new automobiles. You can read this dataset into memory by using the `sysuse` command, which accesses Stata-installed datasets. To read in the data and obtain descriptive statistics, we type

```
. sysuse auto.dta
(1978 Automobile Data)
. summarize
```

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

The dataset comprises 12 variables for 74 automobiles. The average price of the automobiles is \$6,165, and the standard deviation is \$2,949. The column Obs gives the number of observations for which data are available for each variable. The make variable has zero observations because it is a string (or text) variable giving the make of the automobile, and summary statistics are not applicable to a nonnumeric variable. The rep78 variable is available for only 69 of the 74 observations.

A more focused use of summarize restricts attention to selected variables and uses one or more of the available options. For example,

```
. summarize mpg price weight, separator(1)
```

Variable	Obs	Mean	Std. Dev.	Min	Max
mpg	74	21.2973	5.785503	12	41
price	74	6165.257	2949.496	3291	15906
weight	74	3019.459	777.1936	1760	4840

provides descriptive statistics for the mpg, price, and weight variables. The option separator(1) inserts a line between the output for each variable.

1.3.3 Example: The regress command

The regress command implements OLS regression.

You can obtain the syntax of regress by typing help regress. This yields output including

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

It follows that, at the minimum, we need to include the variable name for the dependent variable (in that case, the regression is on an intercept only). Although not explicitly stated, prefixes can be used. Many estimation commands have similar syntax.

Suppose that we want to run an OLS regression of the `mpg` variable (fuel economy in miles per gallon) on `price` (auto price in dollars) and `weight` (weight in pounds). The basic command is simply

```
. regress mpg price weight
```

Source	SS	df	MS	Number of obs =	74
				F(2, 71) =	66.85
Model	1595.93249	2	797.966246	Prob > F =	0.0000
Residual	847.526967	71	11.9369995	R-squared =	0.6531
				Adj R-squared =	0.6434
Total	2443.45946	73	33.4720474	Root MSE =	3.455

mpg	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
price	-.0000935	.0001627	-0.57	0.567	-.000418	.0002309
weight	-.0058175	.0006175	-9.42	0.000	-.0070489	-.0045862
_cons	39.43966	1.621563	24.32	0.000	36.20635	42.67296

The coefficient of `-.0058175` for `weight` implies that fuel economy falls by 5.8 miles per gallon when the car's weight increases by 1,000 pounds.

A more complicated version of `regress` that demonstrates much of the command syntax is the following:

```
. by foreign: regress mpg price weight if weight < 4000, vce(robust)
  (output omitted)
```

For each value of the `foreign` variable, here either 0 or 1, this command fits distinct OLS regressions of `mpg` on `price` and `weight`. The `if` qualifier limits the sample to cars with `weight` less than 4,000 pounds. The `vce(robust)` option leads to heteroskedasticity-robust standard errors being used.

Output from commands is not always desired. We can suppress output by using the `quietly` prefix. For example,

```
. quietly regress mpg price weight
```

The `quietly` prefix does not require a colon, for historical reasons, even though it is a command prefix. In this book, we use this prefix extensively to suppress extraneous output.

The preceding examples used one of the available options for `regress`. From `help regress`, we find that the `regress` command has the following options: `noconstant`, `hascons`, `tsscons`, `vce(`*vcetype*`)`, `level(#)`, `beta`, `eform(`*string*`)`, `noheader`, `plus`, `depname(`*varname*`)`, and `mse1`.

1.3.4 Abbreviations, case sensitivity, and wildcards

Commands and parts of commands can be abbreviated to the shortest string of characters that uniquely identify them, often just two or three characters. For example, we can shorten `summarize` to `su`. For expositional clarity, we do not use such abbreviations in this book; a notable exception is that we may use abbreviations in the options to graphics commands because these commands can get very lengthy. Not using abbreviations makes it much easier to read your do-files.

Variable names can be up to 32 characters long, where the characters can be A–Z, a–z, 0–9, and _ (underscore). Some names, such as `in`, are reserved. Stata is case sensitive, and the norm is to use lowercase.

We can use the wildcard * (asterisk) for variable names in commands, provided there is no ambiguity such as two potential variables for a one-variable command. For example,

```
. summarize t*
```

Variable	Obs	Mean	Std. Dev.	Min	Max
trunk	74	13.75676	4.277404	5	23
turn	74	39.64865	4.399354	31	51

provides summary statistics for all variables with names beginning with the letter `t`. Where ambiguity may arise, wildcards are not permitted.

1.3.5 Arithmetic, relational, and logical operators

The arithmetic operators in Stata are + (addition), - (subtraction), * (multiplication), / (division), ^ (raised to a power), and the prefix - (negation). For example, to compute and display $-2 \times \{9/(8+2-7)\}^2$, which simplifies to -2×3^2, we type

```
. display -2*(9/(8+2-7))^2
-18
```

If the arithmetic operation is not possible, or data are not available to perform the operation, then a missing value denote by . is displayed. For example,

```
. display 2/0
.
```

The relational operators are > (greater than), < (less than), >= (greater than or equal), <= (less than or equal), == (equal), and != (not equal). These are the obvious symbols, except that a pair of equal-signs is used for equality, and != denotes not equal. Relational operators are often used in `if` qualifiers that define the sample for analysis.

Logical operators return 1 for true and 0 for false. The logical operators are & (and), | (or), and ! (not). The operator ~ can be used in place of !. Logical operators are also used to define the sample for analysis. For example, to restrict regression analysis to smaller less expensive cars, type

```
regress mpg price weight if weight <= 4000 & price <= 10000
(output omitted)
```

The string operator + is used to concatenate two strings into a single, longer string.

The order of evaluation of all operators is ! (or ~), ^, - (negation), /, *, - (subtraction), +, != (or ~=), >, <, <=, >=, ==, &, and |.

1.3.6 Error messages

Stata produces error messages when a command fails. These messages are brief, but a fuller explanation can be obtained from the manual or directly from Stata.

For example, if we regress mpg on notthere but the notthere variable does not exist, we get

```
. regress mpg notthere
variable notthere not found
r(111);
```

Here r(111) denotes return code 111. You can obtain further details by clicking on r(111); if in interactive mode or by typing

```
. search rc 111
(output omitted)
```

1.4 Do-files and log files

For Stata analysis requiring many commands, or requiring lengthy commands, it is best to collect all the commands into a program (or script) that is stored in a text file called a do-file.

In this book, we perform data analysis using a do-file. We assume that the do-file and, if relevant, any input and output files are in a common directory and that Stata is executed from that directory. Then we only need to provide the filename rather than the complete directory structure. For example, we can refer to a file as mus02data.dta rather than c:\mus\chapter2\mus02data.dta.

1.4.1 Writing a do-file

A do-file is a text file with extension .do that contains a series of Stata commands.

As an example, we write a two-line program that reads in the Stata example dataset auto.dta and then presents summary statistics for the mpg variable that we already know is in the dataset. The commands are sysuse auto.dta, clear, where the clear option is added to remove the current dataset from memory, and summarize mpg. The two commands are to be collected into a command file called a do-file. The filename should include no spaces, and the file extension is .do. In this example, we suppose this file is given the name example.do and is stored in the current working directory.

To see the current directory, type cd without any arguments. To change to another directory, cd is used with an argument. For example, in Windows, to change to the directory c:\Program Files\Stata10\, we type

```
. cd "c:\Program Files\Stata10"
c:\Program Files\Stata10
```

The directory name is given in double quotes because it includes spaces. Otherwise, the double quotes are unnecessary.

One way to create the do-file is to start Stata and use the Do-file Editor. Within Stata, we select **Window > Do-file Editor > New Do-file**, type in the commands, and save the do-file.

Alternatively, type in the commands outside Stata by using a preferred text editor. Ideally, this text editor supports multiple windows, reads large files (datasets or output), and gives line numbers and column numbers.

The type command lists the contents of the file. We have

```
. type example.do
sysuse auto.dta, clear
summarize mpg
```

1.4.2 Running do-files

You can run (or execute) an already-written do-file by using the Command window. Start Stata and, in the Command window, change directory (cd) to the directory that has the do-file, and then issue the do command. We obtain

```
. do example.do
. sysuse auto.dta, clear
(1978 Automobile Data)
. summarize mpg
    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
         mpg |         74    21.2973    5.785503         12         41
.
end of do-file
```

where we assume that example.do is in directory c:\Program Files\Stata10\.

An alternative method is to run the do-file from the Do-file Editor. Select **Window > Do-file Editor > New Do-file**, and then select **File > Open...** and the appropriate file, and finally select **Tools > Do**. An advantage to using the Do-file Editor is that you can highlight or select just part of the do-file and then execute this part by selecting **Tools > Do Selection**.

You can also run do-files noninteractively, using batch mode. This initiates Stata, executes the commands in the do-file, and (optionally) exits Stata. The term batch

mode is a throwback to earlier times when each line of a program was entered on a separate computer card, so that a program was a collection or "batch" of computer cards. For example, to run example.do in batch mode, double-click on example.do in Windows Explorer. This initiates Stata and executes the file's Stata commands. You can also use the do command. (In Unix, you would use the stata -b example.do command.)

It can be useful to include the set more off command at the start of a do-file so that output scrolls continuously rather than pausing after each page of output.

1.4.3 Log files

By default, Stata output is sent to the screen. For reproducibility, you should save this output in a separate file. Another advantage to saving output is that lengthy output can be difficult to read on the screen; it can be easier to review results by viewing an output file using a text editor.

A Stata output file is called a log file. It stores the commands in addition to the output from these commands. The default Stata extension for the file is .log, but you can choose an alternative extension, such as .txt. An extension name change may be worthwhile because several other programs, such as LaTeX compilers, also create files with the .log extension. Log files can be read as either standard text or in a special Stata code called smcl (Stata Markup and Control Language). We use text throughout this book, because it is easier to read in a text editor. A useful convention can be to give the log the same filename as that for the do-file. For example, for example.do, we save the output as example.txt.

A log file is created by using the log command. In a typical analysis, the do-file will change over time, in which case the output file will also change. The Stata default is to protect against an existing log being accidentally overwritten. To create a log file in text form named example.txt, the usual command is

```
. log using example.txt, text replace
```

The replace option permits the existing version of example.txt, if there is one, to be overwritten. Without replace, Stata will refuse to open the log file if there is already a file called example.txt.

In some cases, we may not want to overwrite the existing log, in which case we would not specify the replace option. The most likely reason for preserving a log is that it contains important results, such as those from final analysis. Then it can be good practice to rename the log after analysis is complete. Thus example.txt might be renamed example07052008.txt.

When a program is finished, you should close the log file by typing log close.

The log can be very lengthy. If you need a hard copy, you can edit the log to include only essential results. The text editor you use should use a monospace font such

as Courier New, where each character takes up the same space, so that output table columns will be properly aligned.

The log file includes the Stata commands, with a dot (.) prefix, and the output. You can use a log file to create a do-file, if a do-file does not already exist, by deleting the dot and all lines that are command results (no dot). By this means, you can do initial work using the Stata GUI and generate a do-file from the session, provided that you created a log file at the beginning of the session.

1.4.4 A three-step process

Data analysis using Stata can repeatedly use the following three-step process:

1. Create or change the do-file.
2. Execute the do-file in Stata.
3. Read the resulting log with a text editor.

The initial do-file can be written by editing a previously written do-file that is a useful template or starting point, especially if it uses the same dataset or the same commands as the current analysis. The resulting log may include Stata errors or estimation results that lead to changes in the original do-file and so on.

Suppose we have fitted several models and now want to fit an additional model. In interactive mode, we would type in the new command, execute it, and see the results. Using the three-step process, we add the new command to the do-file, execute the do-file, and read the new output. Because many Stata programs execute in seconds, this adds little extra time compared with using interactive mode, and it has the benefit of having a do-file that can be modified for later use.

1.4.5 Comments and long lines

Stata do-files can include comments. This can greatly increase understanding of a program, which is especially useful if you return to a program and its output a year or two later. Lengthy single-line comments can be allowed to span several lines, ensuring readability. There are several ways to include comments:

- For single-line comments, begin the line with an asterisk (*); Stata ignores such lines.
- For a comment on the same line as a Stata command, use two slashes (//) after the Stata command.
- For multiple-line comments, place the commented text between slash-star (/*) and star-slash (*/).

The Stata default is to view each line as a separate Stata command, where a line continues until a carriage return (end-of-line or *Enter* key) is encountered. Some commands, such as those for nicely formatted graphs, can be very long. For readability, these commands need to span more than one line. The easiest way to break a line at, say, the 70th column is by using three slashes (`///`) and then continuing the command on the next line.

The following do-file code includes several comments to explain the program and demonstrates how to allow a command to span more than one line.

```
* Demonstrate use of comments
* This program reads in system file auto.dta and gets summary statistics
clear // Remove data from memory
* The next code shows how to allow a single command to span two lines
sysuse ///
auto.dta
summarize
```

For long commands, you can alternatively use the command `#delimit` command. This changes the delimiter from the Stata default, which is a carriage return (i.e., end-of-line), to a semicolon. This also permits more than one command on a single line. The following code changes the delimiter from the default to a semicolon and back to the default:

```
* Change delimiter from cr to semicolon and back to cr
#delimit ;
* More than one command per line and command spans more than one line;
clear; sysuse
auto.dta; summarize;
#delimit cr
```

We recommend using `///` instead of changing the delimiter because the comment method produces more readable code.

1.4.6 Different implementations of Stata

The different platforms for Stata share the same command syntax; however, commands can change across versions of Stata. For this book, we use Stata 10. To ensure that later versions of Stata will continue to work with our code, we include the `version 10` command near the beginning of the do-file.

Different implementations of Stata have different limits. A common limit encountered is the memory allocated to Stata, which restricts the size of dataset that can be handled by Stata. The default is small, e.g., 1 megabyte, so that Stata does not occupy too much memory, permitting other tasks to run while Stata is used. Another common limit is the size of matrix, which limits the number of variables in the dataset.

You can increase or decrease the limits with the `set` command. For example,

```
. set matsize 300
```

sets the maximum number of variables in an estimation command to 300.

The maximum possible values vary with the version of Stata: Small Stata, Stata/IC, Stata/SE, or Stata/MP. The `help limits` command provides details on the limits for the current implementation of Stata. The `query` and `creturn list` commands detail the current settings.

1.5 Scalars and matrices

Scalars can store a single number or a single string, and matrices can store several numbers or strings as an array. We provide a very brief introduction here, sufficient for use of the scalars and matrices in section 1.6.

1.5.1 Scalars

A scalar can store a single number or string. You can display the contents of a scalar by using the `display` command.

For example, to store the number 2×3 as the scalar a and then display the scalar, we type

```
. * Scalars: Example
. scalar a = 2*3
. scalar b = "2 times 3 = "
. display b a
2 times 3 = 6
```

One common use of scalars, detailed in section 1.6, is to store the scalar results of estimation commands that can then be accessed for use in subsequent analysis. In section 1.7, we discuss the relative merits of using a scalar or a macro to store a scalar quantity.

1.5.2 Matrices

Stata provides two distinct ways to use matrices, both of which store several numbers or strings as an array. One way is through Stata commands that have the `matrix` prefix. More recently, beginning with version 9, Stata includes a matrix programming language, Mata. These two methods are presented in, respectively, appendices A and B.

The following Stata code illustrates the definition of a specific 2×3 matrix, the listing of the matrix, and the extraction and display of a specific element of the matrix.

```
. * Matrix commands: Example
. matrix define A = (1,2,3 \ 4,5,6)
. matrix list A
A[2,3]
    c1  c2  c3
r1   1   2   3
r2   4   5   6
. scalar c = A[2,3]
. display c
6
```

1.6 Using results from Stata commands

One goal of this book is to enable analysis that uses more than just Stata built-in commands and printed output. Much of this additional analysis entails further computations after using Stata commands.

1.6.1 Using results from the r-class command summarize

The Stata commands that analyze the data but do not estimate parameters are r-class commands. All r-class commands save their results in `r()`. The contents of `r()` vary with the command and are listed by typing `return list`.

As an example, we list the results stored after using `summarize`:

```
. * Illustrate use of return list for r-class command summarize
. summarize mpg
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         mpg |        74     21.2973    5.785503        12         41
. return list
scalars:
                  r(N) =  74
              r(sum_w) =  74
               r(mean) =  21.2972972972973
                r(Var) =  33.47204738985561
                 r(sd) =  5.785503209735141
                r(min) =  12
                r(max) =  41
                r(sum) =  1576
```

There are eight separate results stored as Stata scalars with the names `r(N)`, `r(sum_w)`, ..., `r(sum)`. These are fairly obvious aside from `r(sum_w)`, which gives the sum of the weights. Several additional results are returned if the `detail` option to `summarize` is used; see [R] **summarize**.

The following code calculates and displays the range of the data:

```
  * Illustrate use of r()
  quietly summarize mpg
  scalar range = r(max) - r(min)
  display "Sample range = " range
Sample range = 29
```

The results in r() disappear when a subsequent r-class or e-class command is executed. We can always save the value as a scalar. It can be particularly useful to save the sample mean.

```
  * Save a result in r() as a scalar
. scalar mpgmean = r(mean)
```

1.6.2 Using results from the e-class command regress

Estimation commands are e-class commands (or estimation-class commands), such as regress. The results are stored in e(), the contents of which you can view by typing ereturn list.

A leading example is regress for OLS regression. For example, after typing

```
. regress mpg price weight
```

Source	SS	df	MS	Number of obs =	74
				F(2, 71) =	66.85
Model	1595.93249	2	797.966246	Prob > F =	0.0000
Residual	847.526967	71	11.9369995	R-squared =	0.6531
				Adj R-squared =	0.6434
Total	2443.45946	73	33.4720474	Root MSE =	3.455

mpg	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
price	-.0000935	.0001627	-0.57	0.567	-.000418	.0002309
weight	-.0058175	.0006175	-9.42	0.000	-.0070489	-.0045862
_cons	39.43966	1.621563	24.32	0.000	36.20635	42.67296

ereturn list yields

```
. * ereturn list after e-class command regress
. ereturn list
scalars:
                  e(N) =  74
              e(df_m) =  2
              e(df_r) =  71
                  e(F) =  66.84814256414501
                 e(r2) =  .6531446579233134
               e(rmse) =  3.454996314099513
                e(mss) =  1595.932492798133
                e(rss) =  847.5269666613265
             e(r2_a) =  .6433740849070687
                 e(ll) =  -195.2169813478502
              e(ll_0) =  -234.3943376482347
```

```
macros:
            e(cmdline) : "regress mpg price weight"
              e(title) : "Linear regression"
                e(vce) : "ols"
              e(depvar) : "mpg"
                e(cmd) : "regress"
         e(properties) : "b V"
            e(predict) : "regres_p"
              e(model) : "ols"
           e(estat_cmd) : "regress_estat"
matrices:
                  e(b) :  1 x 3
                  e(V) :  3 x 3
functions:
              e(sample)
```

The key numeric output in the analysis-of-variance table is stored as scalars. As an example of using scalar results, consider the calculation of R^2. The model sum of squares is stored in `e(mss)`, and the residual sum of squares is stored in `e(rss)`, so that

```
. * Use of e() where scalar
. scalar r2 = e(mss)/(e(mss)+e(rss))
. display "r-squared = " r2
r-squared = .65314466
```

The result is the same as the 0.6531 given in the original regression output.

The remaining numeric output is stored as matrices. Here we present methods to extract scalars from these matrices and manipulate them. Specifically, we obtain the OLS coefficient of `price` from the 1×3 matrix `e(b)`, the estimated variance of this estimate from the 3×3 matrix `e(V)`, and then we form the t statistic for testing whether the coefficient of `price` is zero:

```
. * Use of e() where matrix
. matrix best = e(b)
. scalar bprice = best[1,1]
. matrix Vest = e(V)
. scalar Vprice = Vest[1,1]
. scalar tprice = bprice/sqrt(Vprice)
. display "t statistic for H0: b_price = 0 is " tprice
t statistic for H0: b_price = 0 is -.57468079
```

The result is the same as the -0.57 given in the original regression output.

The results in `e()` disappear when a subsequent e-class command is executed. However, you can save the results by using `estimates store`, detailed in section 3.4.4.

1.7 Global and local macros

A macro is a string of characters that stands for another string of characters. For example, you can use the macro `xlist` in place of `"price weight"`. This substitution can lead to code that is shorter, easier to read, and that can be easily adapted to similar problems.

Macros can be global or local. A global macro is accessible across Stata do-files or throughout a Stata session. A local macro can be accessed only within a given do-file or in the interactive session.

1.7.1 Global macros

Global macros are the simplest macro and are adequate for many purposes. We use global macros extensively throughout this book.

Global macros are defined with the `global` command. To access what was stored in a global macro, put the character `$` immediately before the macro name. For example, consider a regression of the dependent variable `mpg` on several regressors, where the global macro `xlist` is used to store the regressor list.

```
* Global macro definition and use
global xlist price weight
regress mpg $xlist, noheader        // $ prefix is necessary
```

mpg	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
price	-.0000935	.0001627	-0.57	0.567	-.000418	.0002309
weight	-.0058175	.0006175	-9.42	0.000	-.0070489	-.0045862
_cons	39.43966	1.621563	24.32	0.000	36.20635	42.67296

Global macros are frequently used when fitting several different models with the same regressor list because they ensure that the regressor list is the same in all instances and they make it easy to change the regressor list. A single change to the global macro changes the regressor list in all instances.

A second example might be where several different models are fitted, but we want to hold a key parameter constant throughout. For example, suppose we obtain standard errors by using the bootstrap. Then we might define the global macro `nbreps` for the number of bootstrap replications. Exploratory data analysis might set `nbreps` to a small value such as 50 to save computational time, whereas final results set `nbreps` to an appropriately higher value such as 400.

A third example is to highlight key program parameters, such as the variable used to define the cluster if cluster–robust standard errors are obtained. By gathering all such global macros at the start of the program, it can be clear what the settings are for key program parameters.

1.7.2 Local macros

Local macros are defined with the `local` command. To access what was stored in the local macro, enclose the macro name in single quotes. These quotes differ from how they appear on this printed page. On most keyboards, the left quote is located at the upper left, under the tilde, and the right quote is located at the middle right, under the double quote.

As an example of a local macro, consider a regression of the `mpg` variable on several regressors. We define the local macro `xlist` and subsequently access its contents by enclosing the name in single quotes as `` `xlist' ``.

```
. * Local macro definition and use
. local xlist "price weight"
. regress mpg `xlist', noheader     // single quotes are necessary
```

mpg	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
price	-.0000935	.0001627	-0.57	0.567	-.000418	.0002309
weight	-.0058175	.0006175	-9.42	0.000	-.0070489	-.0045862
_cons	39.43966	1.621563	24.32	0.000	36.20635	42.67296

The double quotes used in defining the local macro as a string are unnecessary, which is why we did not use them in the earlier global macro example. Using the double quotes does emphasize that a text substitution has been made. The single quotes in subsequent references to `xlist` are necessary.

We could also use a macro to define the dependent variable. For example,

```
. * Local macro definition without double quotes
. local y mpg
. regress `y' `xlist', noheador
```

mpg	Coef.	Std. Err.	t	P>|t|	[95% Conf.	Interval]
price	-.0000935	.0001627	-0.57	0.567	-.000418	.0002309
weight	-.0058175	.0006175	-9.42	0.000	-.0070489	-.0045862
_cons	39.43966	1.621563	24.32	0.000	36.20635	42.67296

Note that here `` `y' `` is not a variable with N observations. Instead, it is the string `mpg`. The `regress` command simply replaces `` `y' `` with the text `mpg`, which in turn denotes a variable that has N observations.

We can also define a local macro through evaluation of a function. For example,

```
* Local macro definition through function evaluation
local z = 2+2
display `z'
4
```

leads to 'z' being the string 4. Using the equality sign when defining a macro causes the macro to be evaluated as an expression. For numerical expressions, using the equality sign stores the result of the expression and not the characters in the expression itself in the macro. For string assignments, it is best not to use the equality sign. This is especially true when storing lists of variables in macros. Strings in Stata expressions can contain only 244 characters, fewer characters than many variable lists. Macros assigned without an equality sign can hold 165,200 characters in Stata/IC and 1,081,511 characters in Stata/MP and Stata/SE.

Local macros are especially useful for programming in Stata; see appendix A. Then, for example, you can use `y' and `x' as generic notation for the dependent variable and regressors, making the code easier to read.

Local macros apply only to the current program and have the advantage of no potential conflict with other programs. They are preferred to global macros, unless there is a compelling reason to use global macros.

1.7.3 Scalar or macro?

A macro can be used in place of a scalar, but a scalar is simpler. Furthermore, [P] **scalar** points out that using a scalar will usually be faster than using a macro, because a macro requires conversion into and out of internal binary representation. This reference also gives an example where macros lead to a loss of accuracy because of these conversions.

One drawback of a scalar, however, is that the scalar is dropped whenever `clear all` is used. By contrast, a macro is still retained. Consider the following example:

```
. * Scalars disappear after clear all but macro does not
. global b 3
. local c 4
. scalar d = 5
. clear
. display $b _skip(3) `c'   // display macros
3   4
. display d                 // display the scalar
5
. clear all
. display $b _skip(3) `c'   // display macros
3   4
. display d                 // display the scalar
d not found
r(111);
```

Here the scalar d has been dropped after `clear all`, though not after `clear`.

We use global macros in this text because there are cases in which we want the contents of our macros to be accessible across do-files. A second reason for using global macros is that the required $ prefix makes it clear that a global parameter is being used.

1.8 Looping commands

Loops provide a way to repeat the same command many times. We use loops in a variety of contexts throughout the book.

Stata has three looping constructs: `foreach`, `forvalues`, and `while`. The `foreach` construct loops over items in a list, where the list can be a list of variable names (possibly given in a macro) or a list of numbers. The `forvalues` construct loops over consecutive values of numbers. A `while` loop continues until a user-specified condition is not met.

We illustrate how to use these three looping constructs in creating the sum of four variables, where each variable is created from the uniform distribution. There are many variations in the way you can use these loop commands; see [P] **foreach**, [P] **forvalues**, and [P] **while**.

The `generate` command is used to create a new variable. The `runiform()` function provides a draw from the uniform distribution. Whenever random numbers are generated, we set the seed to a specific value with the `set seed` command so that subsequent runs of the same program lead to the same random numbers being drawn. We have, for example,

```
. * Make artificial dataset of 100 observations on 4 uniform variables
. clear
. set obs 100
obs was 0, now 100
. set seed 10101
. generate x1var = runiform()
. generate x2var = runiform()
. generate x3var = runiform()
. generate x4var = runiform()
```

We want to sum the four variables. The obvious way to do this is

```
. * Manually obtain the sum of four variables
. generate sum = x1var + x2var + x3var + x4var
. summarize sum
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sum |       100    2.093172     .594672   .5337163   3.204005
```

We now present several ways to use loops to progressively sum these variables. Although only four variables are considered here, the same methods can potentially be applied to hundreds of variables.

1.8.1 The foreach loop

We begin by using `foreach` to loop over items in a list of variable names. Here the list is `x1var`, `x2var`, `x3var`, and `x4var`.

The variable ultimately created will be called `sum`. Because `sum` already exists, we need to first drop `sum` and then generate `sum=0`. The `replace sum=0` command collapses these two steps into one step, and the `quietly` prefix suppresses output stating that 100 observations have been replaced. Following this initial line, we use a `foreach` loop and additionally use `quietly` within the loop to suppress output following `replace`. The program is

```
.
.       * foreach loop with a variable list
.       quietly replace sum = 0
        foreach var of varlist x1var x2var x3var x4var {
        2.      quietly replace sum = sum + `var'
        3. }
.       summarize sum
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sum |       100    2.093172     .594672   .5337163   3.204005
```

The result is the same as that obtained manually.

The preceding code is an example of a program (see appendix A) with the { brace appearing at the end of the first line and the } brace appearing on its own at the last line of the program. The numbers `2.` and `3.` do not actually appear in the program but are produced as output. In the `foreach` loop, we refer to each variable in the variable list `varlist` by the local macro named `var`, so that `` `var' `` with single quotes is needed in subsequent uses of `var`. The choice of `var` as the local macro name is arbitrary and other names can be used. The word `varlist` is necessary, though types of lists other than variable lists are possible, in which case we use `numlist`, `newlist`, `global`, or `local`; see [P] **foreach**.

An attraction of using a variable list is that the method can be applied when variable names are not sequential. For example, the variable names could have been `incomehusband`, `incomewife`, `incomechild1`, and `incomechild2`.

1.8.2 The forvalues loop

A `forvalues` loop iterates over consecutive values. In the following code, we let the index be the local macro `i`, and `` `i' `` with single quotes is needed in subsequent uses of `i`. The program

```
.
.       * forvalues loop to create a sum of variables
.       quietly replace sum = 0
        forvalues i = 1/4 {
        2.      quietly replace sum = sum + x`i'var
        3. }
```

```
summarize sum
    Variable |  sum |  Obs        Mean    Std. Dev.       Min        Max
-------------+------+--------------------------------------------------------
             |      |  100    2.093172     .594672   .5337163   3.204005
```

produces the same result.

The choice of the name `i` for the local macro was arbitrary. In this example, the increment is one, but you can use other increments. For example, if we use `forvalues i = 1(2)11`, then the index goes from 1 to 11 in increments of 2.

1.8.3 The while loop

A `while` loop continues until a condition is no longer met. This method is used when `foreach` and `forvalues` cannot be used. For completeness, we apply it to the summing example.

In the following code, the local macro `i` is initialized to 1 and then incremented by 1 in each loop; looping continues, provided that $i \leq 4$.

```
. * While loop and local macros to create a sum of variables
. quietly replace sum = 0
. local i 1
. while `i' <= 4 {
  2.     quietly replace sum = sum + x`i'var
  3.     local i = `i' + 1
  4. }
. summarize sum
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sum |       100    2.093172     .594672   .5337163   3.204005
```

1.8.4 The continue command

The `continue` command provides a way to prematurely cease execution of the current loop iteration. This may be useful if, for example, the loop includes taking the log of a number and we want to skip this iteration if the number is negative. Execution then resumes at the start of the next loop iteration, unless the `break` option is used. For details, see `help continue`.

1.9 Some useful commands

We have mentioned only a few Stata commands. See [U] **27.1 43 commands** for a list of 43 commands that everyone will find useful.

1.10 Template do-file

The following do-file provides a template. It captures most of the features of Stata presented in this chapter, aside from looping commands.

```
* 1. Program name
* mus01p2template.do written 2/15/2008 is a template do-file
* 2. Write output to a log file
log using mus01p2template.txt, text replace
* 3. Stata version
version 10.1          // so will still run in a later version of Stata
* 4. Program explanation
* This illustrative program creates 100 uniform variates
* 5. Change Stata default settings - two examples are given
set more off        // scroll screen output by at full speed
set mem 20m         // set aside 20 mb for memory space
* 6. Set program parameters using global macros
global numobs 100
local seed 10101
local xlist xvar
* 7. Generate data and summarize
set obs $numobs
set seed `seed'
generate xvar = runiform()
generate yvar = xvar^2
summarize
* 8. Demonstrate use of results stored in r()
summarize xvar
display "Sample range = " r(max)-r(min)
regress yvar `xlist'
scalar r2 = e(mss)/(e(mss)+e(rss))
display "r-squared = " r2
* 9. Close output file and exit Stata
log close
exit, clear
```

1.11 User-written commands

We make extensive use of user-written commands. These are freely available ado-files (see section A.2.8) that are easy to install, provided you are connected to the Internet and, for computer lab users, that the computer lab places no restriction on adding components to Stata. They are then executed in the same way as Stata commands.

As an example, consider instrumental-variables (IV) estimation. In some cases, we know which user-written commands we want. For example, a leading user-written command for IV is `ivreg2`, and we type `findit ivreg2` to get it. More generally, we can type the broader command

```
. findit instrumental variables
(output omitted)
```

This gives information on IV commands available both within Stata and packages available on the web, provided you are connected to the Internet.

Many entries are provided, often with several potential user-written commands and several versions of a given user-written command. The best place to begin can be a recent *Stata Journal* article because this code is more likely to have been closely vetted for accuracy and written in a way suited to a range of applications. The listing from the `findit` command includes

```
SJ-7-4  st0030_3  . . . . .  Enhanced routines for IV/GMM estimation and testing
. . . . . . . . . . . . . . .  C. F. Baum, M. E. Schaffer, and S. Stillman
(help ivactest, ivendog, ivhettest, ivreg2, ivreset,
overid, ranktest if installed)
Q4/07   SJ 7(4):465--506
extension of IV and GMM estimation addressing hetero-
skedasticity- and autocorrelation-consistent standard
errors, weak instruments, LIML and k-class estimation,
tests for endogeneity and Ramsey's regression
specification-error test, and autocorrelation tests
for IV estimates and panel-data IV estimates
```

The entry means that it is the third revision of the package (`st0030_3`), and the package is discussed in detail in *Stata Journal*, volume 7, number 4 (`SJ-7-4`).

By left-clicking on the highlighted text `st0030_3` on the first line of the entry, you will see a new window with title, description/author(s), and installation files for the package. By left-clicking on the help files, you can obtain information on the commands. By left-clicking on the (`click here to install`), you will install the files into an ado-directory.

1.12 Stata resources

For first-time users, [GS] *Getting Started with Stata* is very helpful, along with analyzing an example dataset such as `auto.dta` interactively in Stata. The next source is [U] *Users Guide*, especially the early chapters.

1.13 Exercises

1. Find information on the estimation method `clogit` using `help`, `search`, `findit`, and `hsearch`. Comment on the relative usefulness of these search commands.
2. Download the Stata example dataset `auto.dta`. Obtain summary statistics for `mpg` and `weight` according to whether the car type is foreign (use the `by foreign:` prefix). Comment on any differences between foreign and domestic cars. Then `regress mpg` on `weight` and `foreign`. Comment on any difference for foreign cars.
3. Write a do-file to repeat the previous question. This do-file should include a log file. Run the do-file and then use a text editor to view the log file.
4. Using `auto.dta`, obtain summary statistics for the `price` variable. Then use the results stored in `r()` to compute a scalar, `cv`, equal to the coefficient of variation (the standard deviation divided by the mean) of `price`.

5. Using `auto.dta`, `regress mpg` on `price` and `weight`. Then use the results stored in `e()` to compute a scalar, `r2adj`, equal to $\overline{R}^2$. The adjusted R^2 equals $R^2 - (1 - R^2)(k-1)/(N-k)$, where N is the number of observations and k is the number of regressors including the intercept. Also use the results stored in `e()` to calculate a scalar, `tweight`, equal to the t statistic to test that the coefficient of `weight` is zero.
6. Using `auto.dta`, define a global macro named `varlist` for a variable list with `mpg`, `price`, and `weight`, and then obtain summary statistics for `varlist`. Repeat this exercise for a local macro named `varlist`.
7. Using `auto.dta`, use a `foreach` loop to create a variable, `total`, equal to the sum of `headroom` and `length`. Confirm by using `summarize` that `total` has a mean equal to the sum of the means of `headroom` and `length`.
8. Create a simulated dataset with 100 observations on two random variables that are each drawn from the uniform distribution. Use a seed of 12345. In theory, these random variables have a mean of 0.5 and a variance of 1/12. Does this appear to be the case here?