Stata Tutorial

Updated for Version 18
https://grodri.github.io/stata

Germán Rodríguez
Princeton University

June 2023

1 Introduction

Stata is a powerful statistical package with smart data-management facilities, a wide array of up-to-date statistical techniques, and an excellent system for producing publication-quality tables and graphs. Stata is fast and easy to use. In this tutorial I start with a quick introduction and overview, and then discuss data management, tables of various types, statistical graphs, and Stata programming.

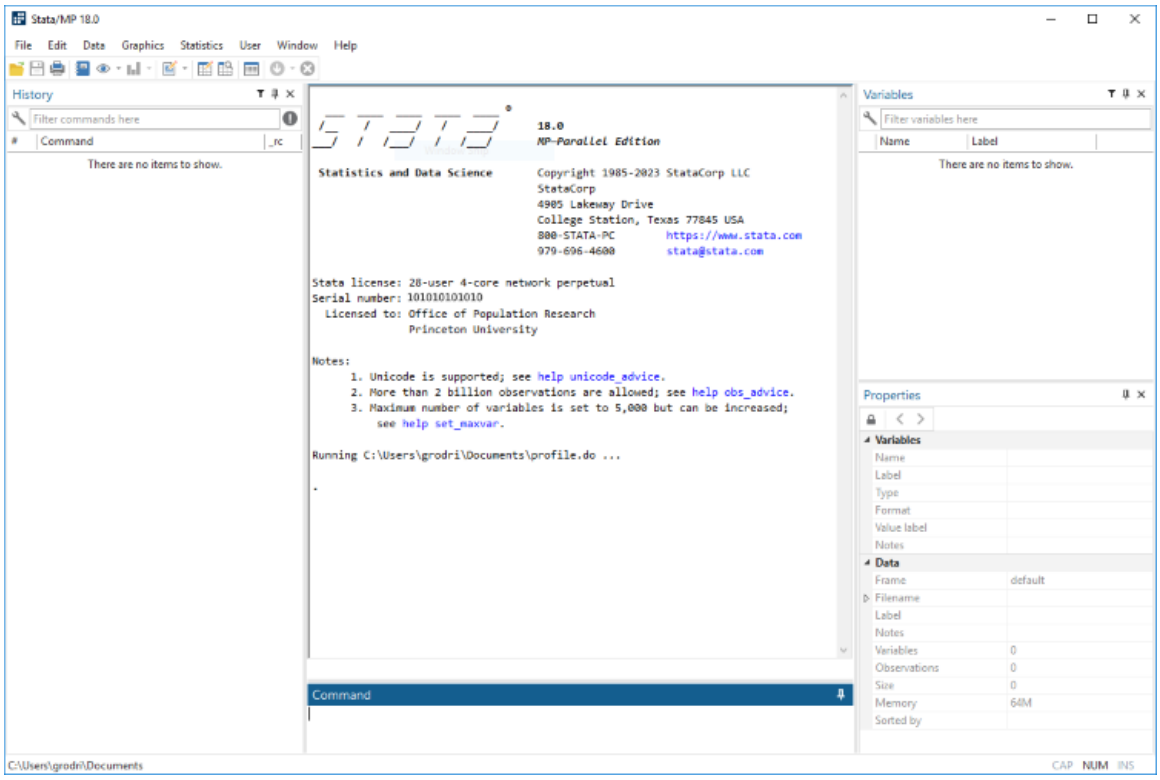
The tutorial has been updated for version 18, but most of the discussion applies to older versions as well. An important exception is Section 3 on Stata Tables, which describes a set a commands for producing customizable tables introduced in version 17, with a new command in version 18. A few other features added in recent versions will be noted along the way.

1.1 A Quick Tour of Stata

Stata is available for Windows, Mac, and Unix computers. This tutorial was created using the Windows version, but all the code shown here runs in all three platforms. There are three editions: (1) Stata/BE, the basic edition (formerly intercooled), suitable for mid-size datasets with up to 2048 variables, (2) Stata/SE, the standard edition (formerly special edition), it can handle up to 32,766 variables as well as longer strings and matrices, and (3) Stata/MP, an edition for multicore/multiprocessor computers that allows processing very large datasets and is substantially faster. (The first two designations changed with Stata 17.) The number of observations is limited by your computer's memory, as long as it doesn't exceed about two billion in Stata/SE and about a trillion in Stata/MP. Starting with version 16 Stata can be installed on 64-bit computers only; previous versions were available for both older 32-bit and newer 64-bit computers. All of these versions can read each other's files within their size limits.

1.1.1 The Stata Interface

When Stata starts up you see five docked windows, initially arranged as shown in the figure below.



The window labeled *Command* is where you type your commands. Stata then shows the results in the larger window immediately above, called appropriately enough *Results*. Your command is added to a list in the window labeled *History* on the left (called *Review* in earlier versions), so you can keep track of the commands you have used. The window labeled *Variables*, on the top right, lists the variables in your dataset. The *Properties* window immediately below that, introduced in version 12, displays properties of your variables and dataset.

You can resize or even close some of these windows. Stata remembers its settings the next time it runs. You can also save (and then load) named preference sets using the menu Edit|Preferences. I happen to like the Compact Window Layout. You can also choose the font used in each window, just right click and select font from the context menu. Finally, it is possible to change the color scheme under General Preferences. You can select one of four overall color schemes: light, light gray, blue or dark. You can also choose one of several preset or customizable styles for the Results and Viewer windows.

There are other windows that we will discuss as needed, namely the *Graph*, *Viewer*, *Variables Manager*, *Data Editor,* and *Do file Editor*.

Starting with version 8 Stata's graphical user interface (GUI) allows selecting commands and options from a menu and dialog system. However, I strongly recommend using the

command language as a way to ensure reproducibility of your results. In fact, I recommend that you type your commands on a separate file, called a do file, as explained in Section 1.2 below, but for now we will just type in the command window. The GUI can be helpful when you are starting to learn Stata, particularly because after you point and click on the menus and dialogs, Stata types the corresponding command for you.

1.1.2 Typing Commands

Stata can work as a calculator using the `display` command. Try typing the following (you may skip the dot at the start of a line, which is how Stata marks the lines you type):

```
. display 2+2
4
. display 2 * ttail(20, 2.1)
.04861759
```

Stata commands are case-sensitive, `display` is not the same as `Display` and the latter will *not* work. Commands can also be abbreviated; the documentation and online help underlines the shortest legal abbreviation of each command, and we will do the same here.

The second command shows the use of a built-in function to compute a p-value, in this case twice the probability that a Student's t with 20 d.f. exceeds 2.1. This result would just make the 5% cutoff. To find the two-tailed 5% critical value try `display invttail(20, 0.025)`. We list a few other functions you can use in Section 2.

If you issue a command and discover that it doesn't work, press the Page Up key to recall it (you can cycle through your command history using the Page Up and Page Down keys) and then edit it using the arrow, insert and delete keys, which work exactly as you would expect. For example the Arrow keys advance a character at a time, combined with the Shift key they select a character at a time, and combined with the Ctlr or Option key they advance or select a word at a time, which you can then delete or replace. A command can be as long as needed (up to some 64k characters); in an interactive session you just keep on typing and the command window will wrap and scroll as needed.

1.1.3 Getting Help

Stata has excellent online help. To obtain help on a command (or function) type `help command_name`, which displays the help on a separate window called the *Viewer*. Try `help ttail`. Each help file appears in a separate viewer tab (a separate window before Stata 12) unless you use the option `, nonew`.

If you don't know the name of the command you need, you can search for it. Stata has a `search` command that will search the documentation and other resources, type `help search` to learn more. By default this command searches the net in Stata 13 and later. If you are using an earlier version, learn about the `findit` command. Also, the `help` command reverts to a search if the argument is not recognized as a command. Try `help Student's t`. This will list all Stata commands and functions related to the t distribution. Among the list of "Stat functions" you will see `t()` for the distribution function and `ttail()` for

right-tail probabilities. Stata can also compute tail probabilities for the normal, chi-squared and F distributions, among others.

One of the nicest features of Stata is that, starting with version 11, all the documentation is available in PDF files. (In fact, since version 13 you can no longer get printed manuals.) Moreover, these files are linked from the online help, so you can jump directly to the relevant section of the manual. To learn more about the help system type `help help`.

1.1.4 Loading a Sample Data File

Stata comes with a few sample data files. You will learn how to read your own data into Stata in Section 2, but for now we will load one of the sample files, namely `lifeexp.dta`, which has data on life expectancy and gross national product (GNP) per capita in 1998 for 68 countries. To see a list of the files shipped with Stata type `sysuse dir`. To load the file we want type `sysuse lifeexp` (the file extension is optional, so I left it out). To see what's in the file type `describe`,. (This can be abbreviated to a single letter, but I prefer `desc`.)

```
. sysuse lifeexp, clear
(Life expectancy, 1998)

. desc

Contains data from C:\Program Files\Stata18\ado\base/l/lifeexp.dta
 Observations:            68                  Life expectancy, 1998
    Variables:             6                  26 Mar 2022 09:40
                                              (_dta has notes)
-------------------------------------------------------------------------------
Variable      Storage   Display    Value
    name         type    format    label      Variable label
-------------------------------------------------------------------------------
region          byte    %16.0g     region     Region
country         str28   %28s                  Country
popgrowth       float   %9.0g                * Avg. annual % growth
lexp            byte    %9.0g                * Life expectancy at birth
gnppc           float   %9.0g                * GNP per capita
safewater       byte    %9.0g                * Safe water
                                             * indicated variables have notes
-------------------------------------------------------------------------------
Sorted by:
```

We see that we have six variables. The dataset has notes that you can see by typing `notes`. Four of the variables have annotations that you can see by typing `notes varname`. You'll learn how to add notes in Section 2.

1.1.5 Descriptive Statistics

Let us run simple descriptive statistics for the two variables we are interested in, using the `summarize` command followed by the names of the variables:

```
. summarize lexp gnppc
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
        lexp |         68    72.27941    4.715315        54         79
       gnppc |         63    8674.857    10634.68       370      39980
```

We see that live expectancy averages 72.3 years and GNP per capita ranges from $370 to $39,980 with an average of $8,675. We also see that Stata reports only 63 observations on

GNP per capita, so we must have some missing values. Let us `list` the countries for which we are missing GNP per capita:

```
. list country gnppc if missing(gnppc)
```

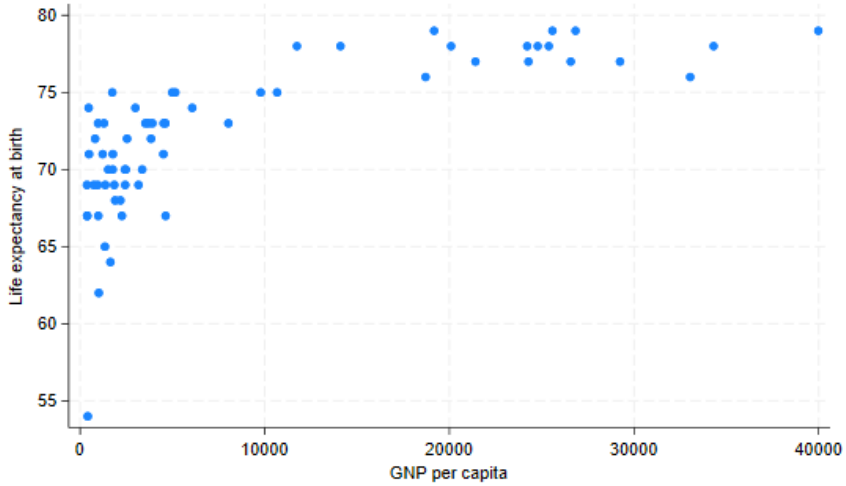
	country	gnppc
7.	Bosnia and Herzegovina	.
40.	Turkmenistan	.
44.	Yugoslavia, FR (Serb./Mont.)	.
46.	Cuba	.
56.	Puerto Rico	.

We see that we have indeed five missing values. This example illustrates a powerful feature of Stata: the action of any command can be restricted to a subset of the data. If we had typed `list country gnppc` we would have listed these variables for all 68 countries. Adding the *condition* `if missing(gnppc)` restricts the list to cases where `gnppc` is missing. Note that Stata lists missing values using a dot. We'll learn more about missing values in Section 2.

1.1.6 Drawing a Scatterplot

To see how life expectancy varies with GNP per capita we will draw a scatter plot using the `graph` command, which has a myriad of subcommands and options, some of which we describe in Section 4.

```
. graph twoway scatter lexp gnppc
. graph export scatter.png, width(550) replace
file scatter.png saved as PNG format
```



The plot shows a curvilinear relationship between GNP per capita and life expectancy. We will see if the relationship can be linearized by taking the log of GNP per capita.

1.1.7 Computing New Variables

We compute a new variable using the `generate` command with a new variable name and an arithmetic expression. Choosing good variable names is important. When computing logs I usually just prefix the old variable name with `log` or `l`, but compound names can easily become cryptic and hard-to-read. Some programmers separate words using an underscore, as in `log_gnp_pc`, and others prefer the camel-casing convention which capitalizes each word after the first: `logGnpPc`. I suggest you develop a consistent style and stick to it. Variable labels can also help, as described in Section 2.

To compute natural logs we use the built-in function `log`:

```
. gen loggnppc = log(gnppc)
(5 missing values generated)
```

Stata says it has generated five missing values. These correspond to the five countries for which we were missing GNP per capita. Try to confirm this statement using the list command. We will learn more about generating new variables in Section 2.

1.1.8 Simple Linear Regression

We are now ready to run a linear regression of life expectancy on log GNP per capita. We will use the `regress` command, which lists the outcome followed by the predictors (here just one, `loggnppc`)

```
. regress lexp loggnppc
```

Source	SS	df	MS	Number of obs	=	63
				F(1, 61)	=	97.09
Model	873.264865	1	873.264865	Prob > F	=	0.0000
Residual	548.671643	61	8.99461709	R-squared	=	0.6141
				Adj R-squared	=	0.6078
Total	1421.93651	62	22.9344598	Root MSE	=	2.9991

lexp	Coefficient	Std. err.	t	P>|t|	[95% conf.	interval]
loggnppc	2.768349	.2809566	9.85	0.000	2.206542	3.330157
_cons	49.41502	2.348494	21.04	0.000	44.71892	54.11113

Note that the regression is based on only 63 observations. Stata omits observations that are missing the outcome or one of the predictors. The log of GNP per capita accounts for 61% of the variation in life expectancy in these countries. We also see that a one percent increase in GNP per capita is associated with an increase of 0.0277 years in life expectancy. (To see this point note that if GNP increases by one percent its log increases by 0.01.)

Following a regression (or in fact any estimation command) you can retype the command with no arguments to see the results again. Try typing `reg`.

1.1.9 Post-Estimation Commands

Stata has a number of post-estimation commands that build on the results of a model fit. A useful command is `predict`, which can be used to generate fitted values or residuals following a regression. The command

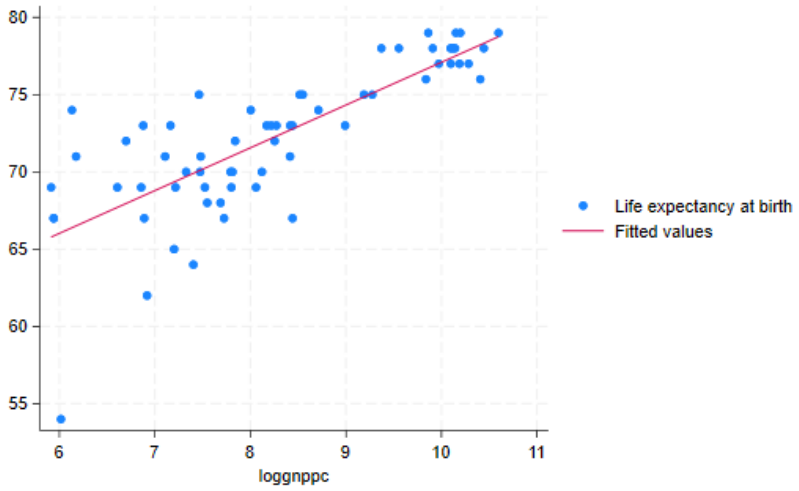
```
. predict plexp
(option xb assumed; fitted values)
(5 missing values generated)
```

generates a new variable, `plexp`, that has the life expectancy predicted from our regression equation. No predictions are made for the five countries without GNP per capita. (If life expectancy was missing for a country it would be excluded from the regression, but a prediction would be made for it. This technique can be used to fill-in missing values.)

1.1.10 Plotting the Data and a Linear Fit

A common task is to superimpose a regression line on a scatter plot to inspect the quality of the fit. We could do this using the predictions we stored in `plexp`, but Stata's `graph` command knows how to do linear fits on the fly, using the `lfit` plot type, and can superimpose different types of `twoway` plots, as explained in more detail in Section 4. Try the command

```
. graph twoway (scatter lexp loggnppc) (lfit lexp loggnppc)
. graph export fit.png, width(550) replace
file fit.png saved as PNG format
```



In this command each expression in parenthesis is a separate two-way plot to be overlayed in the same graph. The fit looks reasonably good, except for a possible outlier.

1.1.11 Listing Selected Observations

It's hard not to notice the country on the bottom left of the graph, which has much lower life expectancy than one would expect, even given its low GNP per capita. To find which country it is we list the (names of the) countries where life expectancy is less than 55:

```
. list country lexp plexp if lexp < 55, clean
      country   lexp      plexp
50.     Haiti     54   66.06985
```

We find that the outlier is Haiti, with a life expectancy 12 years less than one would expect given its GNP per capita. (The keyword `clean` after the comma is an *option* which omits the

borders on the listing. Many Stata commands have options, and these are always specified after a comma.) If you are curious where the United States is try

```
. list gnppc loggnppc lexp plexp if country == "United States", clean
      gnppc   loggnppc   lexp      plexp
58.   29240   10.28329     77   77.88277
```

Here we restricted the listing to cases where the value of the variable `country` was "United States". Note the use of a double equal sign in a logical expression. In Stata `x = 2` assigns the value 2 to the variable `x`, whereas `x == 2` checks to see if the value of `x` is 2.

1.1.12 Saving your Work and Exiting Stata

To exit Stata you use the `exit` command. (You can also use a shortcut or the menu system, type `help exit` for details for your operating system.) If you have been following along this tutorial by typing the commands and try to exit Stata will refuse, saying "no; data in memory would be lost". This happens because we have added a new variable that is not part of the original dataset, and it hasn't been saved. As you can see, Stata is very careful to ensure we don't loose our work.

If you don't care about saving anything you can type `exit, clear`, which tells Stata to quit no matter what. Alternatively, you can save the data to disk using the `save filename` command, and then exit. A cautious programmer will *always* save a modified file using a new name.

1.2 Using Stata Effectively

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

1.2.1 Create a Project Directory

Stata reads and saves data from the current working directory, which you can display using the command `pwd`, short for print working directory. You can change directory using the command `cd` *`directory_name`*, where the directory name follows the conventions of your operating system, including an optional drive letter on Windows. Type `help cd` to learn more. I recommend that you create a separate directory for each course or research project you are involved in, and start your Stata session by changing to that directory.

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

1.2.2 Open a Log File

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

```
log using filename, text replace
```

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

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

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

```
capture log close
```

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

1.2.3 Always Use a Do File

A do file is just a set of Stata commands typed in a plain text file. You can use Stata's own built-in *do-file editor*, which has the great advantage that you can run your code directly from the editor using the shortcut `Ctrl-D` or `Cmd-D` or the run icon, which do their job smartly: if you have selected some text Stata will extend the selection to include complete lines and will then run them; if there is no selection Stata will run the entire script. The editor can be activated with the `doedit` command, the shortcut `Ctrl-9` or `Cmd-9`, or an icon on the GUI.

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

You could even use a word processor such as Word, but you would have to remember to save the file in plain text format, not in Word document format. Also, you may find Word's insistence on capitalizing the first word on each line annoying when you are trying to type

Stata commands that must be in lowercase. You can, of course, turn auto-correct off. But it's a lot easier to just use a plain-text editor.

1.2.4 Use Comments and Annotations

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

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

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

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

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

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

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

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

1.2.5 Continuation Lines

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

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

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

Old hands might write

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

which "comments out" the end of the line.

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

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

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

```
#delimit cr
```

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

1.2.6 A Sample Do File

Here's a simple do file that can reproduce all the results in our Quick Tour. The file doesn't have many comments because this page has all the details. Following the listing we comment on a couple of lines that require explanation.

```
// A Quick Tour of Stata
// Germán Rodríguez - June 2023

version 18
clear
capture log close
log using QuickTour, text replace

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

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

graph twoway scatter lexp gnppc, ///
  title(Life Expectancy and GNP ) xtitle(GNP per capita)
// save the graph in PNG format
graph export scatter.png, width(550) replace
gen loggnppc = log(gnppc)
regress lexp loggnppc

predict plexp

graph twoway (scatter lexp loggnppc) (lfit lexp loggnppc) ///
    ,  title(Life Expectancy and GNP) xtitle(log GNP per capita)
graph export fit.png, width(550) replace

list country lexp plexp if lexp < 55, clean
list gnppc loggnppc lexp plexp if country == "United States", clean
log close
```

We start the do file by specifying the version of Stata that we are using, in this case 18. This

helps ensure that future versions of Stata will continue to interpret the commands correctly, even if Stata has changed, see `help version` for details. (The previous version of this file read version 17, and I could have left that in place to run under version control; the results would be the same because none of the commands used in this quick tour has changed.)

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

Note also that we use a `graph export` command to convert the graph in memory to Portable Network Graphics (PNG) format, ready for inclusion in a web page. We discuss other graph formats in Section 4.

1.2.7 Stata Command Syntax

Having used a few Stata commands it may be time to comment briefly on their structure, which usually follows the following syntax, where bold indicates keywords and square brackets indicate optional elements:

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

We now describe each syntax element:

***command*:** The only required element is the command itself, which is usually (but not always) an action verb, and is often followed by the names of one or more variables. Stata commands are case-sensitive. The commands `describe` and `Describe` are different, and only the former will work. Commands can usually be abbreviated as noted earlier. When we introduce a command we underline the letters that are required. For example `regress` indicates that the `regress` command can be abbreviated to `reg`.

***varlist*:** The command is often followed by the names of one or more variables, for example `describe lexp` or `regress lexp loggnppc`. Variable names are case sensitive; `lexp` and `LEXP` are different variables. A variable name can be abbreviated to the minimum number of letters that makes it unique in a dataset. For example in our quick tour we could refer to `loggnppc` as `log` because it is the only variable that begins with those three letters, but this is a really bad idea. Abbreviations that are unique may become ambiguous as you create new variables, so you have to be very careful. You can also use wildcards such as `v*` or name ranges, such as `v101-v105` to refer to several variables. Type `help varlist` to learn more about variable lists.

***=exp*:** Commands used to generate new variables, such as `generate log_gnp = log(gnp)`, include an arithmetic expression, basically a formula using the standard operators (+ - * and / for the four basic operations and ^ for exponentiation, so 3^2 is three squared), functions, and parentheses. We discuss expressions in Section 2.

`if` *`exp`* `and in` *`range`*: As we have seen, a command's action can be restricted to a subset of the data by specifying a logical condition that evaluates to true or false, such as `lexp < 55`. Relational operators are <, <=, ==, >= and >, and logical negation is expressed using `!` or `~`, as we will see in Section 2. Alternatively, you can specify a range of the data, for example `in 1/10` will restrict the command's action to the first 10 observations. Type `help numlist` to learn more about lists of numbers.

***`weight`*:** Some commands allow the use of weights, type `help weights` to learn more.

`using` *`filename`*: The keyword `using` introduces a file name; this can be a file in your computer, on the network, or on the internet, as you will see when we discuss data input in Section 2.

***`options`*:** Most commands have *`options`* that are specified following a comma. To obtain a list of the options available with a command type `help command`. where `command` is the actual command name.

`by` *`varlist`*: A very powerful feature, it instructs Stata to *repeat* the command for each group of observations defined by distinct values of the variables in the list. For this to work the command must be "byable" (as noted on the online help) and the data must be sorted by the grouping variable(s) (or use `bysort` instead).

1.3 Stata Resources

There are many resources available to learn more about Stata, both online and in print.

1.3.1 Online Resources

Stata has an excellent website at https://www.stata.com. Among other things you will find that they make available online all datasets used in the official documentation, that they publish a journal called The *Stata Journal*, and that they have an excellent bookstore with texts on Stata and related statistical subjects. Stata also offers email and web-based training courses called NetCourses, see https://www.stata.com/netcourse/.

There is a Stata forum where you can post questions and receive prompt and knowledgeable answers from other users, quite often from the indefatigable and extremely knowledgeable Nicholas Cox, who deserves special recognition for his service to the user community. The list was started by Marcello Pagano at the Harvard School of Public Health, and is now maintained by StataCorp, see https://www.statalist.org for more information, including how to participate. Stata also maintains a list of frequently asked questions (FAQ) classified by topic, see https://www.stata.com/support/faqs/.

UCLA maintains an excellent Stata portal at https://stats.idre.ucla.edu/stata/, with many useful links, including a list of resources to help you learn and stay up-to-date with Stata, including classes and seminars, learning modules and useful links, not to mention comparisons with other packages such as SAS and SPSS.

1.3.2 Manuals and Books

The Stata documentation has been growing with each version and now consists of 33 volumes with more than 18,000 pages, all available in PDF format with your copy of Stata. Here

is a list, with italics indicating two new and one renamed manual in Stata 18. The basic documentation consists of a Base Reference Manual, separate volumes on Data Management, Graphics, Customizable Tables and Collected Results, Reporting, and Functions; a User's Guide, an Index, and Getting Started with Stata, which has platform-specific versions for Windows, Mac and Unix. Some statistical subjects that may be important to you are described in 21 separate manuals: *Adaptive Designs: Group Sequential Trials*, Bayesian Analysis, *Bayesian Model Averaging*, *Causal Inference and Treatment-Effects Estimation*, Choice Models, Dynamic Stochastic General Equilibrium Models, Extended Regression Models, Finite Mixture Models, Item Response Theory, Lasso, Longitudinal Data/Panel Data, Meta Analysis, Multilevel Mixed Effects, Multiple Imputation, Multivariate Statistics; Power, Precision and Sample Size; Spatial Autoregressive Models, Structural Equation Modeling, Survey Data, Survival Analysis, and Time Series. Additional volumes of interest to programmers, particularly those seeking to extend Stata's capabilities, are manuals on Programming and on Mata, Stata's matrix programming language. You can access all of these manuals from the help system or at https://www.stata.com/features/documentation/.

A good introduction to Stata is Alan C. Acock's *A Gentle Introduction to Stata*, now in a revised 6th edition. One of my favorite statistical modeling books is Scott Long and Jeremy Freese's *Regression Models for Categorical Dependent Variables Using Stata* (3rd edition); Section 2.10 of this book is a set of recommended practices that should be read and followed faithfully by every aspiring Stata data analyst. Another book I like is Michael Mitchell's excellent *A Visual Guide to Stata Graphics*, which was written specially to introduce the new graphs in version 8 and is now in its 4th edition. Two useful (but more specialized) references written by the developers of Stata are *An Introduction to Survival Analysis Using Stata* (revised 3rd edition), by Mario Cleves, William Gould and Julia Marchenko, and *Maximum Likelihood Estimation with Stata* (4th edition) by William Gould, Jeffrey Pitblado, and Brian Poi. Readers interested in programming Stata will find Christopher F. Baum's *An Introduction to Stata Programming* (2nd edition), and William Gould's *The Mata Book: A Book for Serious Programmers and Those Who Want to Be*, both invaluable.

2 Data Management

In this section I describe Stata data files, discuss how to read raw data into Stata in free and fixed formats, how to create new variables, how to document a dataset labeling the variables and their values, and how to manage Stata system files.

Stata 11 introduced a variables manager that allows editing variable names, labels, types, formats, and notes, as well as value labels, using an intuitive graphical user interface available under Data|Variables Manager in the menu system. While the manager is certainly convenient, I still prefer writing all commands in a do file to ensure research reproducibility. A nice feature of the manager, however, is that it generates the Stata commands needed to accomplish the changes, so it can be used as a learning tool and, as long as you are logging the session, leaves a record behind.

2.1 Stata Files

Stata datasets are rectangular arrays with n observations on m variables. Unlike packages that read one observation at a time, Stata keeps all data in memory, which is one reason why it is so fast. There's a limit of 2,047 variables in Stata/BE, 32,767 in Stata/SE, and 120,000 in Stata/MP. You can have as many observations as your computer's memory will allow, provided you don't go too far above 2 billion cases with Stata/SE and 1 trillion with Stata/MP. (To find these limits type `help limits`.)

2.1.1 Variable Names

Variable names can have up to 32 characters, but many commands print only 12, and shorter names are easier to type. Stata names are *case sensitive*, `Age` and `age` are different variables! It pays to develop a convention for naming variables and sticking to it. I prefer short lowercase names and tend to use single words or abbreviations rather than multi-word names, for example I prefer `effort` or `fpe` to `family_planning_effort or familyPlanningEffort,` although all four names are legal. Note the use of underscores or camel casing to separate words.

2.1.2 Variable Types

Variables can contain numbers or strings. Numeric variables can be stored as integers (bytes, integers, or longs) or floating point (float or double). These types differ in the range or precision of the values they can hold, type `help datatype` for details.

You usually don't need to be concerned about the storage mode; Stata does all calculations using doubles, and the `compress` command will find the most economical way to store each variable in your dataset, type `help compress` to learn more.

You *do* have to be careful with logical comparisons involving floating point types. If you store 0.1 in a float called x, you may be surprised to learn that x == 0.1 is never true. The reason is that 0.1 is "rounded" to different binary numbers when stored as a float (the variable `x`) or as a double (the constant 0.1). This problem does not occur with integers or strings.

String variables can have varying lengths up to 244 characters in Stata 12, or up to two billion characters in Stata 13 or higher, where you can use `str1...str2045` to define fixed-length strings of up to 2045 characters, and `strL` to define a long string, suitable for storing plain text or even binary large objects such as images or word processing documents, type `help strings` to learn more. Strings are ideally suited for id variables because they can be compared without problems.

Sometimes you may need to convert between numeric and string variables. If a variable has been read as a string but really contains numbers you will want to use the command `destring` or the function `real()`. Otherwise, you can use `encode` to convert string data into a numeric variable or `decode` to convert numeric variables to strings. These commands rely on value labels, which are described below.

2.1.3 Missing Values

Like other statistical packages, Stata distinguishes *missing* values. The basic missing value for numeric variables is represented by a dot `.` Starting with version 8 there are 26 additional missing-value codes denoted by `.a` to `.z`. These values are represented internally as very large numbers, so *valid_numbers* `< . < .a < ... < .z`.

To check for missing you need to write `var >= .` (not `var == .`). Stata has a function that can do this comparison, `missing(varname)` and I recommend it because it leads to more readable code, e.g. I prefer `list id if missing(age)` to `list id if age >= .`

Missing values for string variables are denoted by ““, the empty string; not to be confused with a string that is all blanks, such as” “.

Demographic survey data often use codes such as 88 for *not applicable* and 99 for *not ascertained*. For example age at marriage may be coded 88 for single women and 99 for women who are known to be married but did not report their age at marriage. You will often want to distinguish these two cases using different kinds of missing value codes. If you wanted to recode 88's to `.n` (for “na” or not applicable) and 99's to `.m` (for “missing”) you could use the code

```
replace ageAtMar = .n if ageAtMar == 88
replace ageAtMar = .m if ageAtMar == 99
```

Sometimes you want to tabulate a variable including missing values but excluding not applicable cases. If you will be doing this often you may prefer to leave 99 as a regular code and define only 88 as missing. Just be careful if you then run a regression!

Stata ships with a number of small datasets, type `sysuse dir` to get a list. You can use any of these by typing `sysuse name`. The Stata website is also a repository for datasets used in the Stata manuals and in a number of statistical books.

2.2 Reading Data Into Stata

In this section we discuss how to read *raw* data files. If your data come from another statistical package, such as SAS or SPSS, you will be glad to know that starting with version 16 Stata can `import sas` and `import spss`. Older versions could read SAS transport or export files, using the command `fdause` (so-named because this is the format required by the Food and Drug Administration), later renamed to `import sasxport`. Stata can also import and export Excel spreadsheets, type `help import excel` for details, and can read data from relational databases, type `help odbc` for an introduction. For more alternatives consider using a tool such as Stat/Transfer (stattransfer.com).

2.2.1 Free Format

If your data are in free format, with variables separated by blanks, commas, or tabs, you can use the `infile` command.

For an example of a free format file consider the family planning effort data available online as `effort.raw` as shown below. This is essentially a text file with four columns, one with

country names and three with numeric variables, separated by white space. We can read the data into Stata using the command

```
. clear
. infile str14 country setting effort change using ///
>     https://grodri.github.io/datasets/effort.raw
(20 observations read)
```

The `infile` command is followed by the names of the variables. Because the country name is a string rather than a numeric variable we precede the name with `str14`, which sets the type of the variable as a string of up to 14 characters. All other variables are numeric, which is the default type.

The keyword `using` is followed by the name of the file, which can be a file on your computer, a local network, or the internet. In this example we are reading the file directly off the internet. And that's all there is to it. For more information on this command type `help infile1`. To see what we got we can `list` a few cases

```
. list in 1/3
```

	country	setting	effort	change
1.	Bolivia	46	0	1
2.	Brazil	74	0	10
3.	Chile	89	16	29

Spreadsheet packages such as Excel often export data separated by tabs or commas, with one observation per line. Sometimes the first line has the names of the variables. If your data are in this format you can read them using the `import delimited` command. This command superseeded the `insheet` command as of Stata 13. Type `help import delimited` to learn more.

2.2.2 Fixed Format

Survey data often come in fixed format, with one or more records per case and each variable in a fixed position in each record.

The simplest way to read fixed-format data is using the `infix` command to specify the columns where each variable is located. As it happens, the effort data are neatly lined up in columns, so we could read them as follows:

```
. infix str country 4-17 setting 23-24 effort 31-32 change 40-41 using ///
>     https://grodri.github.io/datasets/effort.raw, clear
(20 observations read)
```

This says to read the `country` name from columns 4-17, `setting` from columns 23-24, and so on. It is, of course, essential to read the correct columns. We specified that country was a string variable but didn't have to specify the width, which was clear from the fact that the data are in columns 4-17. The `clear` option is used to overwrite the existing dataset in memory.

If you have a large number of variables you should consider typing the names and locations on a separate file, called a *dictionary*, which you can then call from the `infix` command.

Try typing the following dictionary into a file called `effort.dct`:

```
infix dictionary using https://grodri.github.io/datasets/effort.raw {
  str country  4-17
      setting 23-24
      effort  31-32
      change  40-41
}
```

Dictionaries accept only `/* */` comments, and these must appear *after* the first line. After you save this file you can read the data using the command

```
infix using effort.dct, clear
```

Note that you now 'use' the dictionary, which in turn 'uses' the data file. Instead of specifying the name of the data file in the dictionary you could specify it as an option to the infix command, using the form `infix using dictionaryfile, using(datafile).` The first 'using' specifies the dictionary and the second 'using' is an option specifying the data file. This is particularly useful if you want to use one dictionary to read several data files stored in the same format.

If your observations span multiple records or lines, you can still read them using `infix` as long as all observations have the same number of records (not necessarily all of the same width). For more information see `help infix`.

The `infile` command can also be used with fixed-format data and a dictionary. This is a very powerful command that gives you a number of options not available with `infix`; for example it lets you define variable labels right in the dictionary, but the syntax is a bit more complicated. See `help infile2`.

In most cases you will find that you can read free-format data using `infile` and fixed-format data using `infix`. For more information on various ways to import data into Stata see `help import`.

Data can also be typed directly into Stata using the `input` command, see `help input`, or using the built-in Stata data editor available through Data|Data editor on the menu system.

2.3 Data Documentation

After you read your data into Stata it is important to prepare some documentation. In this section we will see how to create labels for your dataset, the variables, and their values, and how to create notes for the dataset and the variables.

2.3.1 Data Label and Notes

Stata lets you label your dataset using the `label data` command followed by a label of up to 80 characters. You can also add notes of up to ~64K characters each using the `notes` command followed by a colon and then the text:

```
. label data "Family Planning Effort Data"
. notes:  Source P.W. Mauldin and B. Berelson (1978). ///
>   Conditions of fertility decline in developing countries, 1965-75. ///
```

```
>   Studies in Family Planning, 9:89-147
```

Users of the data can type `notes` to see your annotation. Documenting your data carefully always pays off.

2.3.2 Variable Labels and Notes

You can (and should) label your variables using the `label variable` command followed by the name of the variable and a label of up to 80 characters enclosed in quotes. With the `infile` command you can add these labels to the dictionary, which is a natural home for them. Otherwise you should prepare a do file with all the labels. Here's how to define labels for the three variables in our dataset:

```
. label variable setting "Social Setting"
. label variable effort  "Family Planning Effort"
. label variable change  "Fertility Change"
```

Stata also lets you add notes to specific variables using the command `notes varname: text`. Note that the command is followed by a variable name and *then* a colon:

```
. notes change: Percent decline in the crude birth rate (CBR) ///
>   -the number of births per thousand population- between 1965 and 1975.
```

Type `describe` and then `notes` to check our work so far.

2.3.3 Value Labels

You can also label the values of categorical variables. Our dataset doesn't have any categorical variables, but let's create one. We will make a copy of the family planning effort variable and then group it into three categories, 0-4, 5-14 and 15+, which represent weak, moderate and strong programs (the `generate` and `recode` used in the first two lines are described in the next section, where we also show how to accomplish all these steps with just one command):

```
. generate effortg = effort
. recode effortg 0/4=1 5/14=2 15/max=3
(20 changes made to effortg)
. label define effortg 1 "Weak" 2 "Moderate" 3 "Strong", replace
. label values effortg effortg
. label variable effortg "Family Planning Effort (Grouped)"
```

Stata has a two-step approach to defining labels. First you define a *named label set* which associates integer codes with labels of up to 80 characters, using the label define command. Then you associate the set of labels with a variable, using the `label values` command. Often you use the same name for the label set and the variable, as we did in our example.

One advantage of this approach is that you can use the same set of labels for several variables. The canonical example is `label define yesno 1 "yes" 0 "no"`, which can then be associated with all 0-1 variables in your dataset, using a command of the form `label values variablename yesno` for each one. When defining labels you can omit the quotes if the label is a single word, but I prefer to use them always for clarity.

Label sets can be modified using the options `add` or `modify`, listed using `label dir` (lists

only names) or `label list` (lists names and labels), and saved to a do file using `label save`. Type `help label` to learn more about these options and commands. You can also have labels in different languages as explained below.

2.3.4 Multilingual Labels*

(This sub-section can be skipped without loss of continuity.) A Stata file can store labels in several languages and you can move freely from one set to another. One limitation of multi-language support in version 13 and earlier is that labels were restricted to 7-bit ascii characters, so you couldn't include letters with diacritical marks such as accents. This limitation was removed with the introduction of Unicode support in Stata 14, so you can use diacritical marks and other non-ascii characters, not just in labels but throughout Stata.

I'll illustrate the idea by creating Spanish labels for our dataset. Following Stata recommendations we will use the ISO standard two-letter language codes, **en** for English and **es** for Spanish.

First we use `label language` to rename the current language to `en`, and to create a new language set `es`:

```
. label language en, rename
(language default renamed en)
. label language es, new
(language es now current language)
```

If you type `desc` now you will discover that our variables have no labels! We could have copied the English ones by using the option `copy`, but that wouldn't save us any work in this case. Here are Spanish versions of the data and variable labels:

```
. label data "Datos de Mauldin y Berelson sobre Planificación Familiar"
. label variable country "País"
. label variable setting "Indice de Desarrollo Social"
. label variable effort  "Esfuerzo en Planificación Familiar"
. label variable effortg "Esfuerzo en Planificación Familiar (Agrupado)"
. label variable change  "Cambio en la Tasa Bruta de Natalidad (%)"
```

These definitions do not overwrite the corresponding English labels, but coexist with them in a parallel Spanish universe. With value labels you have to be a bit more careful, however; you can't just redefine the label set called `effortg` because it is only the association between a variable and a set of labels, not the labels themselves, that is stored in a language set. What you need to do is define a new label set; we'll call it `effortg_es`, combining the old name and the new language code, and then associate it with the variable `effortg`:

```
. label define effortg_es 1 "Débil" 2 "Moderado" 3 "Fuerte"
. label values effortg effortg_es
```

You may want to try the `describe` command now. Try tabulating effort (output not shown).

```
table effortg
```

Next we change the language back to English and run the table again:

```
label language en
table effortg
```

For more information type `help label_language`.

2.4 Creating New Variables

The most important Stata commands for creating new variables are `generate/replace` and `recode`, and they are often used together.

2.4.1 Generate and Replace

The `generate` command creates a new variable using an expression that may combine constants, variables, functions, and arithmetic and logical operators. Let's start with a simple example: here is how to create setting squared:

```
. gen settingsq = setting^2.
```

If you are going to use this term in a regression you know that linear and quadratic terms are highly correlated. It may be a good idea to center the variable (by subtracting the mean) before squaring it. Here we run `summarize` using `quietly` to suppress the output and retrieve the mean from the stored result `r(mean)`:

```
. quietly summarize setting
. gen settingcsq = (setting - r(mean))^2
```

Note that I used a different name for this variable. Stata will not let you overwrite an existing variable using generate. If you really mean to replace the values of the old variable use `replace` instead. You can also use `drop var_names` to drop one or more variables from the dataset.

2.4.2 Operators and Expressions

The following table shows the standard arithmetic, logical and relational operators you may use in expressions:

Arithmetic	**Logical**	**Relational**
+ add	! not (also ~)	== equal
- subtract	| or	!= not equal (also ~=)
* multiply	& and	< less than
/ divide		<= less than or equal
^ raise to power		> greater than
+ string concatenation		>= greater than or equal

Here's how to create an indicator variable for countries with high-effort programs:

```
generate hieffort1 = effort > 14
```

This is a common Stata idiom, taking advantage of the fact that logical expressions take the

value 1 if true and 0 if false. A common alternative is to write

```
generate hieffort2 = 0
replace  hieffort2 = 1 if effort > 14
```

The two strategies yield exactly the same answer. Both will be wrong if there are missing values, which will be coded as high effort because missing value codes are very large values, as noted in Section 2.1 above. You should develop a good habit of avoiding open ended comparisons. My preferred approach is to use

```
generate hieffort = effort > 14 if !missing(effort)
```

which gives true for effort above 14, false for effort less than or equal to 14, and missing when effort is missing. Logical expressions may be combined using & for "and" or | for "or". Here's how to create an indicator variable for effort between 5 and 14:

```
gen effort5to14 = (effort >=5 & effort <= 14)
```

Here we don't need to worry about missing values, they are excluded by the clause `effort <= 14`.

2.4.3 Functions

Stata has a large number of functions, here are a few frequently-used mathematical functions, type `help mathfun` to see a complete list:

abs(x)	the absolute value of x
exp(x)	the exponential function of x
int(x)	the integer obtained by truncating x towards zero
ln(x) or log(x)	the natural logarithm of x if x>0
log10(x)	the log base 10 of x (for x>0)
logit(x)	the log of the odds for probability x: logit(x) = ln(x/(1-x))
max(x1,x2,...,xn)	the maximum of x1, x2, ..., xn, ignoring missing values
min(x1,x2,...,xn)	the minimum of x1, x2, ..., xn, ignoring missing values
round(x)	x rounded to the nearest whole number
sqrt(x)	the square root of x if x >= 0

These functions are automatically applied to all observations when the argument is a variable in your dataset.

Stata also has a function to generate random numbers (useful in simulation), namely `uniform()`. It also has an extensive set of functions to compute probability distributions (needed for p-values) and their inverses (needed for critical values), including `normal()` for the normal cdf and `invnormal()` for its inverse, see `help density functions` for more information. To simulate normally distributed observations you can use

```
rnormal()   // or invnormal(uniform())
```

There are also some specialized functions for working with strings, see `help string functions`, and with dates, see `help date functions`.

2.4.4 Recoding Variables

The `recode` command is used to group a numeric variable into categories. Suppose for example a fertility survey has age in single years for women aged 15 to 49, and you would like to code it into 5-year age groups. You could, of course, use something like

```
gen age5 = int((age-15)/5)+1 if !missing(age)
```

but this only works for regularly spaced intervals (and is a bit cryptic). The same result can be obtained using

```
recode age (15/19=1) (20/24=2) (25/29=3) (30/34=4) ///
           (35/39=5) (40/44=6) (45/49=7), gen(age5)
```

Each expression in parenthesis is a recoding rule, and consist of a list or range of values, followed by an equal sign and a new value. A range, specified using a slash, includes the two boundaries, so 15/19 is 15 to 19, which could also be specified as 15 16 17 18 19 or even 15 16 17/19. You can use `min` to refer to the smallest value and `max` to refer to the largest value, as in `min/19` and `44/max`. The parentheses may be omitted when the rule has the form range=value, but they usually help make the command more readable.

Values are assigned to the first category where they fall. Values that are never assigned to a category are kept as they are. You can use `else` (or *) as the last clause to refer to any value not yet assigned. Alternatively, you can use `missing` and `nonmissing` to refer to unassigned missing and nonmissing values; these must be the last two clauses and cannot be combined with else.

In our example we also used the `gen()` option to generate a new variable, in this case `age5`; the default is to replace the values of the existing variable. I strongly recommend that you always use the `gen` option, or make a copy of the original variable before recoding it.

You can also specify *value labels* in each recoding rule. This is simpler and less error prone that creating the labels in a separate statement. The option `label(label_name)` lets you assign a name to the labels created (the default is the same as the variable name). Here's an example showing how to recode and label family planning effort in one step (compare with the four commands used in Section 2.4.2 above).

```
recode effort (0/4=1 Weak) (5/14=2 Moderate) (15/max=3 Strong) ///
     , generate(efffortg) label(effortg)
```

It is often a good idea to cross-tabulate original and recoded variables to check that the transformation has worked as intended. (Of course this can only be done if you have generated a new variable!)

2.5 Managing Stata Files

Once you have created a Stata system file you will want to save it on disk using `save filename, replace`, where the replace option, as usual, is needed only if the file already exists. To load a Stata file you have saved in a previous session you issue the command `use filename`.

If there are temporary variables you do not need in the saved file you can drop them (before saving) using `drop varnames`. Alternatively, you may specify the variables you want to keep, using `keep varnames`. With large files you may want to `compress` them before saving; this command looks at the data and stores each variable in the smallest possible data type that will not result in loss of precision.

It is possible to add variables or observations to a Stata file. To add *variables* you use the `merge` command, which requires two (or more) Stata files, usually with a common id so observations can be paired correctly. A typical application is to add household information to an individual data file. Type `help merge` to learn more.

To add *observations* to a file you use the `append` command, which requires the data to be appended to be on a Stata file, usually containing the same variables as the dataset in memory. You may, for example, have data for patients in one clinic and may want to append similar data from another clinic. Type `help append` to learn more.

A related but more specialized command is `joinby`, which forms all pairwise combinations of observations in memory with observations in an external dataset (see also `cross`).

2.6 Data Frames

Stata 16 introduced *frames*, which allow it to keep more than one dataset in memory at the same time. Consider a situation where you have household and individual data on separate files, both with a common household id, and need to combine them. In previous versions of Stata you would have needed to `merge` the files. Starting with Stata 16 you can store both datasets as frames, and *link* the household data to each individual. Stata 18 lets you save a set of frames in a single file with extension `.dtas`, the plural of `.dta`. You can then `use` that file to load the set of frames into memory. There are many more applications of frames, type `help frames` to learn more.

3 Stata Tables

Stata 17 introduced a new system for producing highly-customizable tables. At the heart of the system is a new `collect` command that can be used to collect the results left behind by various Stata commands and present them in tables. It also introduced a new `table` command that simplifies the process for many kinds of tabulations, and later an `etable` command that specializes in tables of estimates. Stata 18 added a `dtable` command to easily produce tables of descriptive statistics. In this tutorial we will touch briefly on all four commands. Stata 16 and earlier had a different `table` command with its own syntax and features, still available under version control.

3.1 Frequency Tables

Frequency tables include marginals or one-way distributions, crosstabs or two-way tabulations, and multi-way tables involving three or more variables.

3.1.1 One-Way Tables

The simplest table we can consider is just a one-way frequency table, where we often want to show percents as well as counts. The example below uses an extract from the 1975 Dominican Republic Fertility Survey and tabulates the distribution of respondent's education

```
. use https://grodri.github.io/datasets/drsr03x, clear
(DRSR03 extract)
. table educg, statistic(frequency) statistic(percent)
```

	Frequency	Percent
Education level		
0-2	941	30.21
3-4	771	24.75
5-7	744	23.88
8-18	659	21.16
Total	3,115	100.00

If you just type `table educg` you will see the frequencies, which is the default. If you want percents instead you use the option `statistic(percent)`. If you want both frequencies and percents you use the `statistic` option twice, as we did here.

You could, of course, obtain the same results using `tabulate educg`, which also gives you cumulative frequencies. However, the new `table` command is much more powerful, letting you customize the table and export the result in various formats.

To give you just one example, suppose you wanted to label the columns `N` and `%`. Although we view this as a one-way table, it has two *dimensions*, the education groups that go in the rows, and the two results that go in the columns, a dimension Stata calls `result` with *levels* `frequency` and `percent`. We can use `collect` to replace the labels of the levels of result and then preview our change. Try the next two commands

```
collect label levels result frequency "N" percent "%", modify
collect preview
```

The table above can be transposed, putting the results in the rows and the categories of education in the columns using the command `collect layout (result) (educg)`. (Alternatively, we could specify `table () (educg)` from the outset.)

The `collect` commands act on the current collection, which was produced by the `table` command and is actually called `Table`. We'll see how to generate our own collections in Section 3.4. To learn more about one-way tables type `help table oneway`.

3.1.2 Two-Way Tables

To obtain a two-way table we specify a row and a column variable. The example below looks at contraceptive use by education groups.

```
. table educg cuse, statistic(percent, across(cuse))
```

	Contraceptive use			
	Not using	Inefficient	Efficient	Total
Education level				
0-2	69.38	5.37	25.25	100.00
3-4	59.65	5.45	34.90	100.00
5-7	50.00	8.50	41.50	100.00
8-18	31.84	14.43	53.73	100.00
Total	57.07	7.36	35.57	100.00

If you just type `table educg cuse` you will get the frequencies. Here we are more interested in row percents, which we obtain using the `percent` statistic with the `across(cuse)` option. We see that use of both efficient and inefficient methods increases substantially with educational level.

This survey defined contraceptive use only for currently married fecund women, and `table` by default excludes missing values. To include missing values use the `missing` option. To see the frequencies add the `statistic(frequency)` option. To learn more about two-way tables type `help table twoway`.

3.1.3 Multi-way Tables

It is also possible to do three-way tables, which is as far as we'll go because tables get rather unwieldy as the number of dimensions increases. Let us look at contraceptive use by area and education:

```
. table (area educg) (cuse), statistic(percent, across(cuse))
```

	Contraceptive use			
	Not using	Inefficient	Efficient	Total
Type of area				
Urban				
Education level				
0-2	54.17	5.95	39.88	100.00
3-4	49.70	2.42	47.88	100.00
5-7	47.92	7.81	44.27	100.00
8-18	31.40	14.53	54.07	100.00
Total	45.77	7.75	46.48	100.00
Rural				
Education level				
0-2	77.01	5.07	17.91	100.00
3-4	66.53	7.53	25.94	100.00
5-7	53.51	9.65	36.84	100.00
8-18	34.48	13.79	51.72	100.00
Total	68.06	6.97	24.97	100.00
Total				
Education level				
0-2	69.38	5.37	25.25	100.00
3-4	59.65	5.45	34.90	100.00
5-7	50.00	8.50	41.50	100.00
8-18	31.84	14.43	53.73	100.00
Total	57.07	7.36	35.57	100.00

This command combines categories of residence and education in the rows and shows contraceptive use in the columns. I used parentheses for clarity, but they can be omitted.

We see that use of contraception increases with education in both areas, and is generally much higher in urban than rural areas.

We could also produce separate tables for urban and rural areas. Try the following command

```
table (educg) (cuse) (area), statistic(percent, across(cuse))
```

Here parentheses are required, and the order is rows, columns, panels, so `area` comes last. The results are the same as before, but to compare urban and rural you have to look across panels.

You can supress marginal totals using the `nototals` option, or specify which margins to include with `totals()`, using `#` to interact variables. For example we could supress the total panel but keep the row totals, so it is clear that the percents add to 100% in each row, by using `totals(educg#area)`. To learn more type `help table multiway`.

3.2 Tables of Statistics

These are just like the frequency tables we have seen, except that the cells show summary statistics of yet another variable. The table can have rows, columns and panels, each with one or more variables. We illustrate with two classification variables.

3.2.1 A Two-Way Table of Statistics

Here is a table showing the mean number of years of education by age groups and area of residence.

```
. table ageg area, statistic(mean educ) nformat(%5.2f)
```

	Type of area		
	Urban	Rural	Total
Age groups			
15-19	6.21	4.20	5.31
20-29	6.67	3.75	5.40
30-39	4.98	2.64	3.88
40-49	4.32	1.63	2.90
Total	5.87	3.25	4.66

We use the `nformat` option to set the format for numeric output, so we get just two decimal points. We notice that younger women have achieved more education than their older counterparts in both areas, and that average education is higher in urban than in rural areas.

This table could use a title. As it happens the `table` command does not have a title option, but there is a `collect title` command that adds a title to the current collection, and a `collect preview` command to display the collection. Try

```
collect title "Mean years of education by age and area"
collect preview
```

Alternatively, you could add a note at the foot of the table with `collect note "Cells show mean years of education"`.

Tables of statistics can include not just means, but many other statistics, such as the median, quartiles, standard deviation or variance. For a full list of the statistics available type `help table_summary##stat`. An interesting "statistic" is `fvproportion`, which gives relative frequencies for a factor or categorical variable.

It is possible to include two (or more) statistics in the same table. Here is an example showing the mean and standard deviation of years of education by age groups and area of residence.

```
. table ageg area, statistic(mean educ) statistic(sd educ) ///
>   nformat(%5.2f) sformat((%s) sd) style(table-tab2)
```

	Type of area		
	Urban	Rural	Total
Age groups			
15-19	6.21	4.20	5.31
	(2.97)	(2.67)	(3.01)
20-29	6.67	3.75	5.40
	(3.95)	(2.87)	(3.81)
30-39	4.98	2.64	3.88
	(3.87)	(2.40)	(3.47)
40-49	4.32	1.63	2.90
	(3.93)	(1.71)	(3.26)
Total	5.87	3.25	4.66
	(3.79)	(2.71)	(3.58)

Type just the first line first to see all the defaults. The second line adds some customization. We use our old friend `nformat` to display the statistics with just two decimals. We also use `sformat` to print the standard deviation in parentheses, specifying `sd` to ensure that this format applies only to that statistic.

Why two kinds of formats? All numeric output is first converted to a string, using an `nformat` if any. Then that string is displayed using an `sformat` if any. So a standard deviation of 9.4148 becomes "9.41" using the numeric format `%5.2f`, and is displayed as "(9.41)" using the string format `(%s)`.

Finally we use a built-in style called `table-tab2` to hide the labels for the statistics and add some space between the age groups. To learn more about the available styles type `help Predefined styles`.

To learn more about the `table` command, and its many options, including the `command` option that lets you run any Stata command and collect its results, type `help table`.

3.2.2 Descriptive Statistics: Table 1

Research reports often include a table showing descriptive statistics for a number of variables, using the mean and standard deviation for numeric or continuous variables, and relative frequencies for categorical or factor variables, frequently within categories of another variable of interest. Sometimes this is called "Table 1". The `table` command can produce this type

of table, but the `dtable` command added in version 18 makes it very easy.

Here is a table showing means and standard deviations for age and years of education, our two continuous variables, and the frequency and percent distribution of contraceptive use, all separately for urban and rural areas.

```
. dtable age educ i.cuse, by(area, test)
note: using test regress across levels of area for age and educ.
note: using test pearson across levels of area for cuse.
```

	Type of area			
	Urban	Rural	Total	Test
N	1,683 (54.0%)	1,432 (46.0%)	3,115 (100.0%)	
Age in years	27.435 (9.415)	28.515 (10.083)	27.931 (9.741)	0.002
Education in years	5.866 (3.786)	3.249 (2.708)	4.663 (3.580)	<0.001
Contraceptive use				
Not using	319 (45.8%)	488 (68.1%)	807 (57.1%)	<0.001
Inefficient	54 (7.7%)	50 (7.0%)	104 (7.4%)	
Efficient	324 (46.5%)	179 (25.0%)	503 (35.6%)	

As you can see, all we need to do is list the variables to be described, using the `i.` prefix for factor variables. The `by()` option specifies a classification variable, with the suboption `test` to request a test of differences across that variable, based on regression or Pearson's statistic as indicated in the notes. That's quite a bit of work with little effort on our part.

We see that the sample has a few more urban than rural women, and that urban women are younger, more educated, and more likely to use contraception (particularly efficient methods) than rural women. Moreover, all three differences are highly significant.

The sample statistics showing the urban/rural split can be omitted using the `nosample` option. You can also select which statistics to calculate and where to place them using the `sample` option, type `help dtable##sample` for details.

There is a `continuous` option to specify the statistics and/or tests to use for one or more continuous variables. For example if you wanted to use the median and interquartile range as descriptive statistics and the Kruskal-Wallis rank test for education you could use the option `continuous(educ, stat(median iqr) test(kwallis))`. Omitting the variable name would apply these choices to all continuous variables. To see a list of all the statistics and tests available for continuous variables type `help dtable##cstats` and `help dtable##ctests`.

There is an equivalent `factor` option to specify the statistics and tests to be used for factor variables. For example you can use Fisher's exact test, or a test based on ordinal association, such as Kendall's tau or Goodman and Kruskal's gamma. Type `help dtable##fstats` and `help dtable##ftests` for a full list of statistics and tests available for factor variables.

The `dtable` command has a large number of options, including several that control table styles. The command creates its own collection called `DTable`, which allows further customization using `collect` commands. To learn more type `help dtable`.

3.2.3 An Alternative Table 1

The code below shows an alternative "table 1" that can be obtained with the `table` command in both Stata 17 and 18. It shows sample sizes, mean and standard deviations on separate lines for continuous variables, and just percents for factor variables, but no significance tests.

```
. gen N = 1
. table (var) (area) ,  ///
>   stat(count N)                          /// sample
>   stat(mean age educ) stat(sd age educ) /// continuous
>   stat(fvpercent cuse)                   /// factor
>   nformat(%5.2f mean sd) nformat(%5.1f fvpercent) ///
>   sformat((%s) sd) sformat(%s%% fvpercent)  style(table-1)
```

	Type of area		
	Urban	Rural	Total
N	1,683	1,432	3,115
Age in years	27.43	28.51	27.93
	(9.41)	(10.08)	(9.74)
Education in years	5.87	3.25	4.66
	(3.79)	(2.71)	(3.58)
Contraceptive use			
Not using	45.8%	68.1%	57.1%
Inefficient	7.7%	7.0%	7.4%
Efficient	46.5%	25.0%	35.6%

We first create a new variable called `N` to obtain sample sizes. We specify the table rows using `var`, which refers to the variables in the `statistics` option, and the columns using `area`. We then request the `count` for the sample size, the `mean` and `sd` for our continuous variables, and the `fvpercent` for our factor variable.

To control the number of decimals printed we use our old friend `nformat`, specifying 2 decimals for the mean and standard deviation, but just one for percents. To enclose the standard deviations in parentheses and append a `%` sign to the percents we use `sformat`. (If you are puzzled by the `%s%%` format, note that `%s` is the placeholder for the string and that to append a `%` symbol we need to escape it using `%%`.)

Finally we use the built-in style `table-1`, which provides a more compact layout for factor variables and a few other tweaks. Try running the table without the style to see what it does.

3.3 Tables of Estimates

We now turn our attention to tables presenting the results of one or more estimation commands. We will use as an example simple linear regression with the `regress` command, but the same ideas apply to other models. We could collect the results ourselves using `collect` as a prefix of the `regress` command, or even the `command` option of `table`, but the `etable` command makes things easier.

3.3.1 A Single Regression

If you type `etable` after a `regress` command you get a table showing coefficients with standard errors in parentheses, and the number of observations at the bottom. Let us add just a couple of options.

```
. sysuse auto, clear
(1978 automobile data)
. quietly regress mpg i.foreign
. etable, showstars showstarsnote
```

	mpg
Car origin	
Foreign	4.946 **
	(1.362)
Intercept	19.827 **
	(0.743)
Number of observations	74

** p<.01, * p<.05

So foreign cars travel almost 5 more miles per gallon than domestic cars. The option `showstars` shows the usual significance stars, and `showstarsnote` adds an explanatory note. The stars may be customized using the `stars()` option, type `help table##starspec` to see how.

3.3.2 Comparing Two Regressions

To compare two or more regressions all we have to do is save the results of each one using `estimates store` (before they are overwriten by the next regression) and then pass the list of stored estimates to `etable`.

```
. gen gphm = 100/mpg
. quietly regress gphm i.foreign
. estimates store unadjusted
. quietly regress gphm i.foreign weight
. estimates store adjusted
. etable, estimates(unadjusted adjusted) column(estimates) ///
>     cstat(_r_b) cstat(_r_z, sformat((%s))) ///
>     note(test statistic in parentheses) showstars showstarsnote
```

	unadjusted	adjusted
Car origin		
Foreign	-1.005 **	0.622 **
	(-3.29)	(3.11)
Weight (lbs.)		0.002 **
		(13.74)
Intercept	5.318 **	-0.073
	(31.92)	(-0.18)
Number of observations	74	74

** p<.01, * p<.05
test statistic in parentheses

Here we compare the efficiency of foreign and domestic cars before and after adjusting for weight. Our measure of efficiency is gallons per 100 miles or `gphm` rather than the usual

`mpg`, because it has a more linear relationship with weight. To get the defaults try `etable estimates(unadjusted adjusted)`. Here we added a couple of options.

The option `column(estimates)` specifies that we want the columns to be labeled with the name of the estimates rather than the name of the dependent variable, which is the default.

The `cstat` option (short for coefficient statistics), lets you select which statistics to display. Type `help etable##cstat` to see a complete list. Here we selected the coefficient (`_r_b`) and the test statistic (`_r_z`). To make sure the test statistic is in parentheses we use the `sformat` option of `cstat` to specify `(%s)`, where `%s` is a placeholder for the string, just as we did earlier in Section 3.2.1. We also use the `note` option of `etable` to indicate exactly what's shown.

There is also a `mstat` option (short for model statistics) that lets you select model statistics to display, such as the number of cases, R-squared, Akaike's information criterion, and others. Type `help etable##mstat` to see a list. Try adding R-squared to the previous table.

3.3.3 Regressions with Different Outcomes

Our last example compared regressions with the same outcome and different predictors. It is also possible to compare regressions with different outcomes and the same predictors (or at least some overlap). The table below compares regressions of `weight` and `length` using four and three predictors, respectively, with foreign cars as the reference cell for car origin:

```
. quietly regress weight ib1.foreign price rep78 headroom
. estimates store weight
. quietly regress length ib1.foreign price rep78
. estimates store length
. etable, estimates(weight length) eqrecode(weight=both length=both) ///
>     mstat(N) mstat(r2) showstars showstarsnote
```

	weight	length
Car origin		
Domestic	893.057 **	29.353 **
	(137.788)	(5.013)
Price	0.140 **	0.003 **
	(0.017)	(0.001)
Repair record 1978	-47.367	-0.211
	(61.474)	(2.347)
Headroom (in.)	222.060 **	
	(61.361)	
Intercept	1048.304 **	147.845 **
	(320.826)	(11.229)
Number of observations	69	69
R-squared	0.76	0.56

```
** p<.01, * p<.05
```

The essential new option here is `eqrecode()` which ensures that coefficients for the same predictor with different outcomes appear in the same row. Try running the command without this option to see the default. This option is also essential if you run a multivariate regression. At the bottom of the table we listed R-squared for each regression, but you already knew how to do that, right? Did you notice that to keep the number of observations you have to add `mstat(N)`?

The `etable` command creates a collection called `ETable` which becomes the current collection and can then be modified and/or exported. Type `help etable` to learn more.

3.4 Collection Tables

Let us move now to an example where we will collect the results of standard Stata commands ourselves. We want to calculate Tukey's five number summary, namely the minimum, first quartile, median, third quartile and maximum. These statistics are all computed by `summarize` with the `detail` option. We would like to do this for several variables.

The `collect` command can be used as a prefix to gather the results stored by a general command in `r()` or by an estimation command in `e()`. You can find out exactly what a command has stored by typing `return list` after a general command such as `summarize`, or typing `ereturn list` after an estimation command. But don't worry, `collect` will gather everything. So here is our table:

```
. sysuse auto, clear
(1978 automobile data)
. collect clear
. quietly collect, tags(cmdset[mpg]):    summarize mpg,    detail
. quietly collect, tags(cmdset[length]): summarize length, detail
. quietly collect, tags(cmdset[weight]): summarize weight, detail
. collect style autolevels result min p25 p50 p75 max
. collect label levels result ///
>     min "Min" p25 "Q1" p50 "Md" p75 "Q3" max "Max", modify
. collect layout (cmdset) (result)
Collection: default
      Rows: cmdset
   Columns: result
   Table 1: 3 x 5
```

	Min	Q1	Md	Q3	Max
mpg	12	18	20	25	41
length	142	170	192.5	204	233
weight	1760	2240	3190	3600	4840

This will require a bit of explanation. We start by clearing the collection system with `collect clear`.

We then collect the results of `summarize mpg, detail`, which will produce the statistics we need, using `quietly` to skip displaying them. We also ask the system to tag the results with the name of the variable being summarized, which unfortunately is not stored with the results. Fortunately Stata creates a *dimension* called `cmdset` for our commands, which are just numbered 1, 2, and 3. The `tags` option creates a more informative tag, using the name of the variable.

Next we define a style. As it happens, `summarize, detail` produces 19 results and we don't want them all, just the five-number summary. The `collect style autolevels result` command sets the levels of `result` to the five statistics we want. (Alternatively, you can specify which results to collect, type `help collect get` to learn more.)

Stata generates labels for practically all the results stored by its commands, for example the label for `p25` is "25th percentile", and by default uses these on the tables. We would like to use shorter labels, in this case "Q1", hence the `collect label levels result` command.

The final step is to specify the layout of the table with `collect layout`, which says we want the `cmdset` with the variable names in the rows, and the `result` with the five-number summaries in the columns. The row and column specifications in `collect layout` must be enclosed in parentheses.

Rather than repeat essentially the same command three times, varying only the name of the variable, we could have used a loop, a concept discussed later in Section 5.2 of this tutorial. That would make it easy to include many more variables in our table.

It is possible to produce similar results using `table`, as all five summaries are in the list of statistics available, but the idea here was to collect the results ourselves to give you a sense of the power and flexibility of the collection system.

3.5 Customizing Tables

Consider the two-way table in Section 3.1.2, showing contraceptive use by education. We would like to show just the row percents, as we did, but add a column with the total number of observations in each row. One way to do this is to get both the frequencies and percents, and then decide exactly what we want to show and how. We will also modify the header, and remove a vertical border. Try the following commands (you may want to try the first two without `quietly` to see what happens at each step):

```
. use https://grodri.github.io/datasets/drsr03x, clear
(DRSR03 extract)
. quietly table educg cuse, stat(percent, across(cuse)) stat(frequency)
. quietly collect layout (educg) ///
>     (cuse#result[percent]  cuse[.m]#result[frequency])
. collect style header result , level(hide)
. collect style cell border_block, border(right, pattern(nil))
. collect preview
```

	Contraceptive use				
	Not using	Inefficient	Efficient	Total	
Education level					
0-2	69.38	5.37	25.25	100.00	503
3-4	59.65	5.45	34.90	100.00	404
5-7	50.00	8.50	41.50	100.00	306
8-18	31.84	14.43	53.73	100.00	201
Total	57.07	7.36	35.57	100.00	1,414

After using `table` to tabulate the data, we use `collect layout` to specify rows with `educg` and columns with the percents for `cuse` (using an interaction between `cuse` and `result[percent]`) and the frequency for the total (interacting `cuse[.m]` with `result[frequency]`).

We have used *dimensions* informally to refer to the rows and columns of a table, but the concept of *dimension* here is more general, representing all features used to tag the elements

of a collection. Type `collect dims` to list all dimensions of the current collection. Type `collect levelsof` *`dimname`* to list the levels of a dimension, and `collect label list` *`dimname`* to list the labels of the levels. This is how I learned that `cuse[.m]` had the totals.

Finally we use a couple of `collect style` commands that aim for a cleaner look; one to remove the labels of the levels of result from the header, and another to omit the vertical border between the row headers and the body of the table. This, by the way, uses yet another dimension called `border_block`, used to tag cells in the row and column headers, the top-left corner, and the body of the table with the items. Type `collect levelsof border_block` to list the level names.

This example has barely touched the surface of table customization. To learn more type `help collect`.

3.6 Exporting Tables

Tables are displayed on your screen but can also be exported in various formats, including HTML, Word documents, Excel documents, LaTeX, PDF, plain text, Markdown and even Stata's own SMCL format. Type `collect export` to learn more.

4 Stata Graphics

Stata has excellent graphic facilities, accessible through the `graph` command, see `help graph` for an overview. The most common graphs in statistics are X-Y plots showing points or lines. These are available in Stata through the `twoway` subcommand, which in turn has many sub-subcommands or plot types, the most important of which are `scatter` and `line`. I will also describe briefly bar plots, available through the `bar` subcommand, and other plot types.

Stata 10 introduced a graphics editor that can be used to modify a graph interactively. I do not recomment this practice, however, because it conflicts with the goals of documenting and ensuring reproducibility of all the steps in your research.

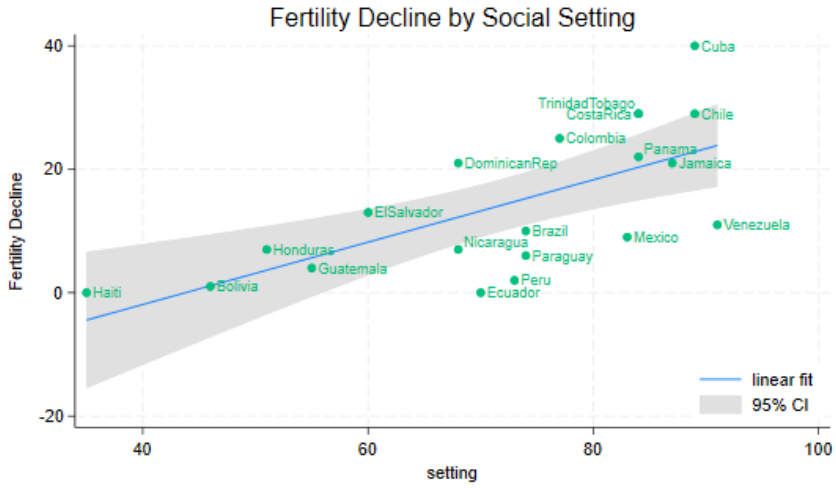
All the graphs in this section (except where noted) use the new default scheme in Stata 18, called `stcolor`. If you use an earlier version your graphs will look a bit different, but the commands shown here will still work. I discuss schemes in Section 4.2.5.

4.1 Scatterplots

In this section I will illustrate a few plots using the data on fertility decline first used in Section 2.1. To read the data from net-aware Stata type

```
. infile str14 country setting effort change ///
>     using https://grodri.github.io/datasets/effort.raw, clear
(20 observations read)
```

To whet your appetite, here's the plot that we will produce in this section:



4.1.1 A Simple Scatterplot

To produce a simple scatterplot of fertility change by social setting you use the command

```
graph twoway scatter change setting
```

Note that you specify `y` first, then `x`. Stata labels the axes using the variable labels, if they are defined, or variable names if not. The command may be abbreviated to `twoway scatter`, or just `scatter` if that is the only plot on the graph. We will now add a few bells and whistles.

4.1.2 Fitted Lines

Suppose we want to show the fitted regression line as well. In some packages you would need to run a regression, compute the fitted line, and then plot it. Stata can do all that in one step using the `lfit` plot type. (There is also a `qfit` plot for quadratic fits.) This can be combined with the scatter plot by enclosing each sub-plot in parenthesis. (One can also combine plots using two horizontal bars `||` to separate them.)

```
graph twoway (scatter setting effort) ///
             (lfit setting effort)
```

Now suppose we wanted to put confidence bands around the regression line. Stata can do this with the `lfitci` plot type, which draws the confidence region as a gray band. (There is also a `qfitci` band for quadratic fits.) Because the confidence band can obscure some points we draw the region first and the points later

```
graph twoway (lfitci setting effort) ///
             (scatter setting effort)
```

Note that this command doesn't label the y-axis but uses a legend instead. You could specify a label for the y-axis using the `ytitle()` option, and omit the (rather obvious) legend using `legend(off)`. Here we specify both as options to the `twoway` command. To make the options more obvious to the reader, I put the comma at the start of a new line:

```
graph twoway (lfitci setting effort) ///
             (scatter setting effort) ///
           , ytitle("Fertility Decline") legend(off)
```

4.1.3 Labeling Points

There are many options that allow you to control the markers used for the points, including their shape and color, see `help marker_options`. It is also possible to label the points with the values of a variable, using the `mlabel(varname)` option. In the next step we add the country names to the plot:

```
graph twoway (lfitci change setting) ///
             (scatter change setting, mlabel(country) )
```

One slight problem with the labels is the overlap of Costa Rica and Trinidad Tobago (and to a lesser extent Panama and Nicaragua). We can solve this problem by specifying the position of the label relative to the marker using a 12-hour clock (so 12 is above, 3 is to the right, 6 is below and 9 is to the left of the marker) and the `mlabv()` option. We create a variable to hold the position set by default to 3 o'clock and then move Costa Rica to 9 o'clock and Trinidad Tobago to just a bit above that at 11 o'clock (we can also move Nicaragua and Panama up a bit, say to 2 o'clock):

```
. gen pos=3
. replace pos = 11 if country == "TrinidadTobago"
(1 real change made)
. replace pos = 9 if country == "CostaRica"
(1 real change made)
. replace pos = 2 if country == "Panama" | country == "Nicaragua"
(2 real changes made)
```

The command to generate this version of the graph is as follows

```
graph twoway (lfitci change setting) ///
             (scatter change setting, mlabel(country) mlabv(pos) )
```

4.1.4 Titles, Legends and Captions

There are options that apply to all two-way graphs, including titles, labels, and legends. Stata graphs can have a `title()` and `subtitle()`, usually at the top, and a `legend()`, `note()` and `caption()`, usually at the bottom, type `help title_options` to learn more. Usually a title is all you need. Stata 11 allows text in graphs to include bold, italics, greek letters, mathematical symbols, and a choice of fonts. Stata 14 introduced Unicode, greatly expanding what can be done. Type `help graph text` to learn more.

Our final tweak to the graph will be to add a legend to specify the linear fit and 95% confidence interval, but not fertility decline itself. We do this using the `order(2 "linear fit" 1 "95% CI")` option of the legend to label the second and first items in that order. We also use `ring(0)` to move the legend inside the plotting area, and `pos(5)` to place the legend box near the 5 o'clock position. Our complete command is then

```
. graph twoway (lfitci change setting) ///
>          (scatter change setting, mlabel(country) mlabv(pos) ) ///
```

```
>        , title("Fertility Decline by Social Setting") ///
>          ytitle("Fertility Decline") ///
>          legend(ring(0) pos(5) order(2 "linear fit" 1 "95% CI"))
. graph export twoway.png, width(550) replace
file twoway.png saved as PNG format
```

The result is the graph shown at the beginning of this section.

4.1.5 Axis Scales and Labels

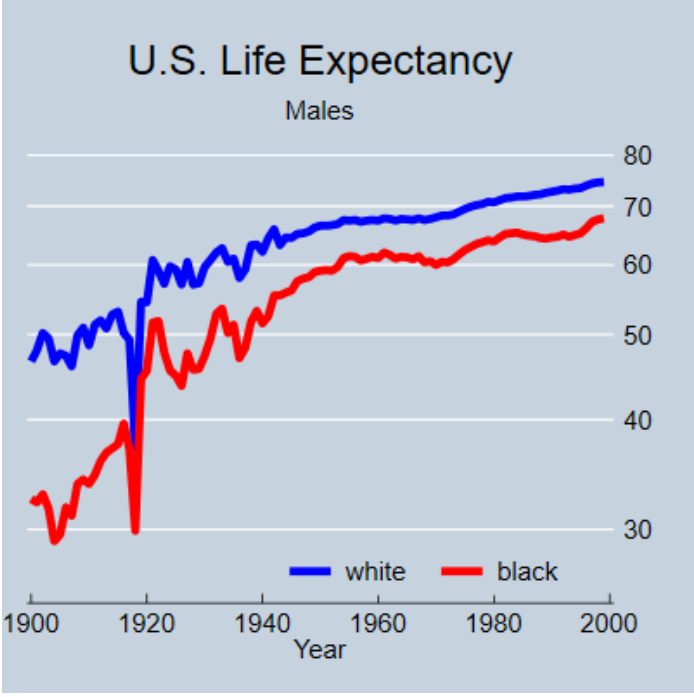
There are options that control the scaling and range of the axes, including `xscale()` and `yscale()`, which can be arithmetic, log, or reversed, type `help axis_scale_options` to learn more. Other options control the placing and labeling of major and minor ticks and labels, such as as `xlabel()`, `xtick()` and `xmtick()`, and similarly for the y-axis, see `help axis_label_options`. Usually the defaults are acceptable, but it's nice to know that you can change them.

4.2 Line Plots

I will illustrate line plots using data on U.S. life expectancy, available as one of the datasets shipped with Stata. (Try `sysuse dir` to see what else is available.)

```
. sysuse uslifeexp, clear
(U.S. life expectancy, 1900-1999)
```

The idea is to plot life expectancy for white and black males over the 20th century. Again, to whet your appetite I'll start by showing you the final product, and then we will build the graph step by step.



4.2.1 A Simple Line Plot

The simplest plot uses all the defaults:

```
graph twoway line le_wmale le_bmale year
```

If you are puzzled by the dip before 1920, Google "US life expectancy 1918". We could abbreviate the command to `twoway line`, or even `line` if that's all we are plotting. (This shortcut only works for `scatter` and `line`.)

The `line` plot allows you to specify more than one "y" variable, the order is y_1, y_2, ..., y_m, x. In our example we specified two, corresponding to white and black life expectancy. Alternatively, we could have used two line plots: `(line le_wmale year) (line le_bmale year)`.

4.2.2 Titles and Legends

The default graph is quite good, but the legend seems too wordy. We will move most of the information to the title and keep only ethnicity in the legend:

```
graph twoway line le_wmale le_bmale year ///
    , title("U.S. Life Expectancy") subtitle("Males") ///
      legend( order(1 "white" 2 "black") )
```

Here I used three options, which as usual in Stata go after a comma: `title`, `subtitle` and `legend`. The `legend` option has many sub options; I used `order` to list the keys and their labels, saying that the first line represented whites and the second blacks. To omit a key you just leave it out of the list. To add text without a matching key use a hyphen (or minus sign) for the key. There are many other legend options, see `help legend_option` to learn more.

I would like to use space a bit better by moving the legend inside the plot area, say around the 5 o'clock position, where improving life expectancy has left some spare room. As noted earlier we can move the legend inside the plotting area by using `ring(0)`, the "inner circle", and place it near the 5 o'clock position using `pos(5)`. Because these are legend sub-options they have to go *inside* `legend()`:

```
graph twoway line le_wmale le_bmale year ///
    , title("U.S. Life Expectancy") subtitle("Males") ///
      legend( order(1 "white" 2 "black") ring(0) pos(5) )
```

4.2.3 Line Styles

I don't know about you, but I find hard to distinguish the default lines on the plot. Stata lets you control the line style in different ways. The `clstyle()` option lets you use a named style, such as `foreground`, `grid`, `yxline`, or `p1-p15` for the styles used by lines 1 to 15, see `help linestyle`. This is useful if you want to pick your style elements from a *scheme*, as noted further below.

Alternatively, you can specify the three components of a style: the line pattern, width and color:

- Patterns are specified using the `clpattern()` option. The most common patterns are `solid`, `dash`, and `dot`; see `help linepatternstyle` for more information.
- Line width is specified using `clwidth()`; the available options include `thin`, `medium` and `thick`, see `help linewidthstyle` for more.
- Colors can be specified using the `clcolor()` option using color names (such as `red`, `white` and `blue`, `teal`, `sienna`, and many others) or RGB values, see `help colorstyle`.

Here's how to specify blue for whites and red for blacks:

```
graph twoway (line le_wmale le_bmale year , clcolor(blue red) ) ///
        , title("U.S. Life Expectancy") subtitle("Males") ///
        legend( order(1 "white" 2 "black") ring(0) pos(5))
```

Note that `clcolor()` is an option of the line plot, so I put parentheses round the `line` command and inserted it there.

4.2.4 Scale Options

It looks as if improvements in life expectancy slowed down a bit in the second half of the century. This can be better appreciated using a log scale, where a straight line would indicate a constant percent improvement. This is easily done using the axis options of the two-way command, see `help axis_options`, and in particular `yscale()`, which lets you choose `arithmetic`, `log`, or `reversed` scales. There's also a suboption `range()` to control the plotting range. Here I will specify the y-range as 25 to 80 to move the curves a bit up:

```
. graph twoway (line le_wmale le_bmale year , clcolor(blue red) ) ///
>     , title("U.S. Life Expectancy") subtitle("Males") ///
>     legend( order(1 "white" 2 "black") ring(0) pos(5)) ///
>     yscale(log range(25 80))
```

4.2.5 Graph Schemes

Stata uses schemes to control the appearance of graphs, see `help scheme`. You can set the default scheme to be used in all graphs with `set scheme_name`. You can also redisplay the (last) graph using a different scheme with `graph display, scheme(scheme_name)`.

To see a list of available schemes type `graph query, schemes`. Try `stgcolor` for the scheme used in the Stata manuals, `stcolor_alt` for a scheme used by some Stata commands, and `economist` for the style used in *The Economist*. Using the latter we obtain the graph shown at the start of this section.

```
. graph display, scheme(economist)
. graph export economist.png, width(400) replace
file economist.png saved as PNG format
```

4.3 Other Graphs

I conclude the graphics section discussing bar graphs, box plots, and kernel density plots using area graphs with transparency.

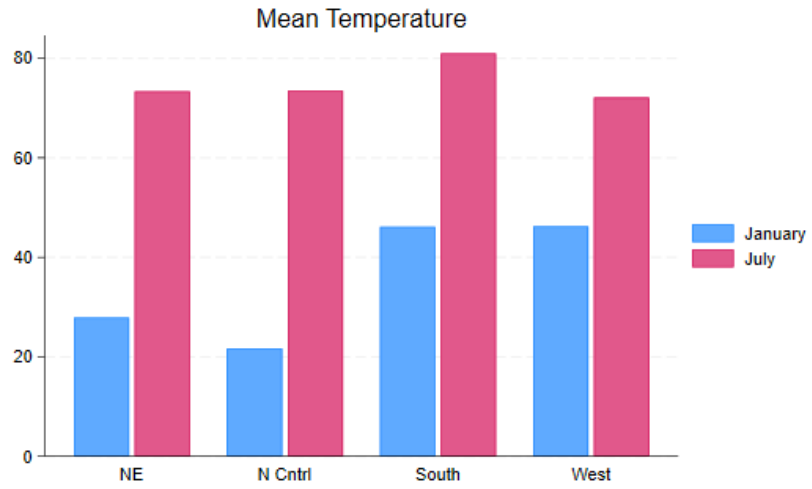
4.3.1 Bar Graphs

Bar graphs may be used to plot the frequency distribution of a categorical variable, or to plot descriptive statistics of a continuous variable within groups defined by a categorical variable. For our examples we will use the city temperature data that ships with Stata.

If I was to just type `graph bar, over(region)` I would obtain the frequency distribution of the region variable. Let us show instead the average temperatures in January and July. To do this I could specify `(mean) tempjan (mean) tempjuly`, but because the default statistic is the mean I can use the shorter version below. I think the default legend is too long, so I also specified a custom one.

I use `over()` so the regions are overlaid in the same graph; using `by()` instead, would result in a graph with a separate panel for each region. The `bargap()` option controls the gap between bars for different statistics in the same over group; here I put a small space. The `gap()` option, not used here, controls the space between bars for different over groups. I also set the intensity of the color fill to 70%, which I think looks nicer.

```
. sysuse citytemp, clear
(City temperature data)
. graph bar tempjan tempjul, over(region) bargap(10) intensity(70) ///
>     title(Mean Temperature) legend(order(1 "January" 2 "July"))
. graph export bar.png, width(550) replace
file bar.png saved as PNG format
```



Obviously the north-east and north-central regions are much colder in January than the south and west. There is less variation in July, but temperatures are higher in the south.

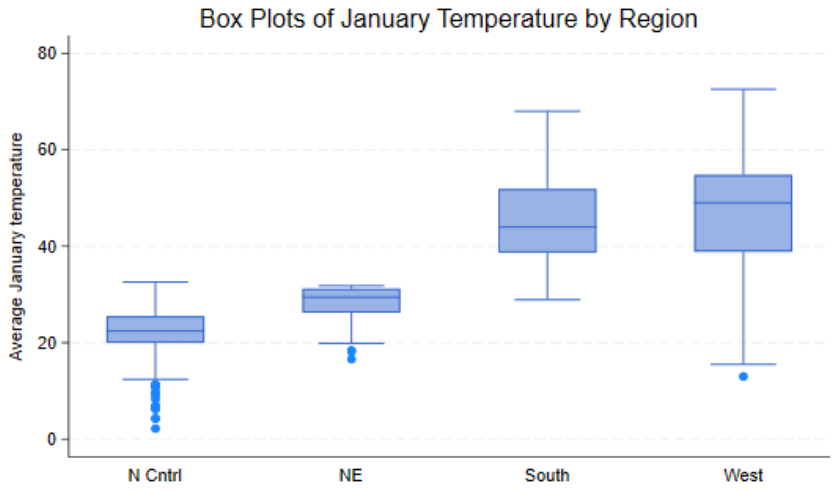
4.3.2 Box Plots

A quick summary of the distribution of a variable may be obtained using a "box-and-wiskers" plot, which draws a box ranging from the first to the third quartile, with a line at the median, and adds "wiskers" going out from the box to the adjacent values, defined as the highest and lowest values that are no farther from the median than 1.5 times the inter-quartile range.

Values further out are outliers, indicated by circles.

Let us draw a box plot of January temperatures by region. I will use the `over(region)` option, so the boxes will be overlaid in the same graph, rather than `by(region)`, which would produce a separate panel for each region. The option `sort(1)` arranges the boxes in order of the median of `tempjan`, the first (and in this case only) variable. I also set the box color to a nice blue by specifying the Red, Blue and Green (RGB) color components in a scale of 0 to 255:

```
. graph box tempjan, over(region, sort(1)) box(1, color("51 102 204")) ///
>     title(Box Plots of January Temperature by Region)
. graph export boxplot.png, width(550) replace
file boxplot.png saved as PNG format
```



We see that January temperatures are lower and less variable in the north-east and north-central regions, with quite a few cities with unusually cold averages.

4.3.3 Kernel Density Estimates

A more detailed view of the distribution of a variable may be obtained using a smooth histogram, calculated using a kernel density smoother using the `kdensity` command.

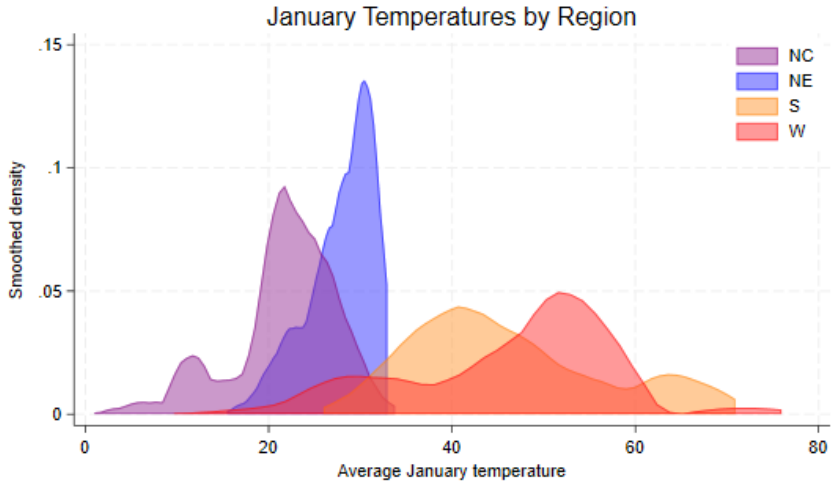
Let us run separate kernel density estimates for January temperatures in each region using all the defaults, and save the results.

```
. kdensity tempjan if region== 1, generate(x1  d1)
. kdensity tempjan if region== 2, generate(x2  d2)
. kdensity tempjan if region== 3, generate(x3  d3)
. kdensity tempjan if region== 4, generate(x4  d4)
. generate zero = 0
```

Because we are using essentially the same command four times we could have used a loop, explained later in Section 5.2 of this tutorial, but perhaps it is clearer this way. We also generate a baseline at zero.

Next we plot the density estimates using area plots with a floor at zero. Because the densities overlap, I use the opacity option introduced in Stata 15 to make them 50% transparent. In this case I used color names, followed by a `%` symbol and the opacity. I also simplify the legend a bit, match the order of the densities, and put it in the top right corner of the plot.

```
. twoway rarea d1 zero x1, color("blue%50") ///
>    ||  rarea d2 zero x2, color("purple%50") ///
>    ||  rarea d3 zero x3, color("orange%50")  ///
>    ||  rarea d4 zero x4, color("red%50") ///
>        title(January Temperatures by Region) ///
>        ytitle("Smoothed density") ///
>        legend(ring(0) pos(2) col(1) order(2 "NC" 1 "NE" 3 "S" 4 "W"))
. graph export kernel.png, width(550) replace
file kernel.png saved as PNG format
```



The plot gives us a clear picture of regional differences in January temperatures, with colder and narrower distributions in the north-east and north-central regions, and warmer with quite a bit of overlap in the south and west.

4.4 Managing Graphs

Stata keeps track of the last graph you have drawn, which is stored in memory, and calls it `Graph`. You can actually keep more than one graph in memory if you use the `name()` option to name the graph when you create it. This is useful for combining graphs, type `help graph combine` to learn more. Note that graphs kept in memory disappear when you exit Stata, even if you save the data, unless you save the graph itself.

To save the current graph on disk using Stata's own format, type `graph save filename`. This command has two options, `replace`, which you need to use if the file already exists, and `asis`, which freezes the graph (including its current style) and then saves it. The default is to save the graph in a live format that can be edited in future sessions, for example by changing the scheme. After saving a graph in Stata format you can load it from the disk with the command `graph use filename`. (Note that `graph save` and `graph use` are analogous to `save` and `use` for Stata files.) Any graph stored in memory can be displayed

using `graph display [name]`. (You can also list, describe, rename, copy, or drop graphs stored in memory, type `help graph_manipulation` to learn more.)

If you plan to incorporate the graph in another document you will probably need to save it in a more portable format. Stata's command `graph export filename` can export the graph using a wide variety of vector or raster formats, usually specified by the file extension. *Vector* formats such as Windows metafile (wmf or emf) or Adobe's PostScript and its variants (ps, eps, pdf) contain essentially drawing instructions and are thus resolution independent, so they are best for inclusion in other documents where they may be resized. *Raster* formats such as Portable Network Graphics (png) save the image pixel by pixel using the current display resolution, and are best for inclusion in web pages. Stata 15 added Scalable Vector Graphics (SVG), a vector image format that is supported by all major modern web browsers.

You can also print a graph using `graph print`, or copy and paste it into a document using the Windows clipboard; to do this right click on the window containing the graph and then select copy from the context menu.

5 Programming Stata

This section is a gentle introduction to programming Stata. I discuss *macros* and *loops*, and show how to write your own (simple) programs. This is a large subject and all I can hope to do here is provide a few tips that hopefully will spark your interest in further study. However, the material covered will help you use Stata more effectively.

Stata 9 introduced a new and extremely powerful matrix programming language called *Mata*, and Stata expanded the choice of languages by integrating Python in version 16 and Java in version 17. In addition, it is possible to write Stata plugins in C. All of these languages are beyond the scope of this introductory tutorial. Your efforts here will not be wasted, however, because these options are complementary to -not a complete substitute for- classic Stata programming.

To learn more about programming Stata I recommend Kit Baum's *An Introduction to Stata Programming*, now in its second edition, and William Gould's *The Mata Book*. You may also find useful Chapter 18 in the *User's Guide*, referring to the *Programming* volume and/or the online help as needed. Nick Cox's regular columns in the *Stata Journal* are a wonderful resource for learning about Stata. Other resources were listed in Section 1 of this tutorial.

5.1 Macros

A macro is simply a name associated with some text. Macros can be local or global in scope.

5.1.1 Storing Text in Local Macros

Local macros have names of up to 31 characters and are known only in the current context (the console, a do file, or a program).

You *define* a local macro using `local name [=] text` and you *evaluate* it using `` `name' ``. (Note the use of an opening backtick or left quote and a closing straight quote.)

The first variant, without an equal sign, is used to store arbitrary text of up to ~64k characters (up to a million in Stata SE). The text is often enclosed in quotes, but it doesn't have to be.

Example: Control Variables in Regression. You need to run a bunch of regression equations that include a standard set of control variables, say `age`, `agesq`, `education`, and `income`. You could, of course, type these names in each equation, or you could cut and paste the names, but these alternatives are tedious and error prone. The smart way is to define a macro

```
local controls age agesq education income
```

You then type commands such as

```
regress outcome treatment `controls'
```

which in this case is exactly equivalent to typing `regress outcome treatment age agesq education income`.

If there's only one regression to run you haven't saved anything, but if you have to run several models with different outcomes or treatments, the macro saves work and ensures consistency.

This approach also has the advantage that if later on you realize that you should have used log-income rather than income as a control, all you need to do is change the macro definition at the top of your do file, say to read `logincome` instead of `income` and all subsequent models will be run with income properly logged (assuming these variables exist).

Warning: Evaluating a macro that doesn't exist is not an error; it just returns an empty string. So be careful to spell macro names correctly. If you type `regress outcome treatment `contrls'`, Stata will read `regress outcome treatment`, because the macro `contrls` does not exist. The same would happen if you type `` `control' `` because macro names cannot be abbreviated the way variable names can. Either way, the regression will run without any controls. But you always check your output, right?

A Technical Note on Nested Macros Macro definitions may include other macros. For example we could define `age` and then use it in `controls`:

```
local age "age agesq"
local controls `age' education income
```

The first ocurrence of `age` in the first line is the name of the macro, and the second occurrence the name of a variable. I used quotes to make the code clearer, but they are optional. Stata never gets confused.

The macro `age` in the definition of `controls` is resolved at the time the macro is defined, not when it is evaluated. Therefore changing the contents of `age` at a later time will not change `controls`. Suppose you run a few models and then decide to control for age using `age5`, a factor variable with age in five-year groups. You define `local age i.age5`. The problem is that `controls` still has `age` and `agesq`.

There is, however, a way to achieve that particular effect. The trick is to escape the macro evaluation character when you define the outer macro, typing `local controls \`age' education income`. Now Stata does not evaluate the inner macro (but eats the escape character), so the contents of `controls` becomes `` `age' education income ``. When the `controls` macro is evaluated, Stata sees that it includes the macro `age` and substitutes its current contents.

In one case substitution occurs when the macro is defined, in the other when it is evaluated.

5.1.2 Storing Results in Local Macros

The second type of macro definition, `local name = text` with an equal sign, is used to store *results*. It instructs Stata to treat the text on the right hand side as an expression, evaluate it, and store a text representation of the result under the given name.

Suppose you just run a regression and want to store the resulting R-squared, for comparison with a later regression. You know that `regress` stores R-squared in `e(r2)`, so you think `local rsq e(r2)` would do the trick. Well, does it?

Your macro stored the formula `e(r2)`, as you can see by typing `display "`rsq'"`. What you needed to store was the value. The solution is to type `local rsq = e(r2)`, with an equal sign. This causes Stata to evaluate the expression and store the result.

To see the difference try this

```
. sysuse auto, clear
(1978 automobile data)
. quietly regress mpg weight
. local rsqf e(r2)
. local rsqv = e(r2)
. di `rsqf´       // this has the current R-squared
.65153125
. di `rsqv´       // as does this
.65153125
. quietly regress mpg weight foreign
. di `rsqf´       // the formula has the new R-squared
.66270291
. di `rsqv´       // this guy has the old one
.65153125
```

Another way to force evaluation is to enclose `e(r2)` in single quotes when you define the macro. This is called a *macro expression*, and is also useful when you want to display results. It allows us to type `display "R-squared=`rsqv'"` instead of `display "R-squared=" `rsq'`. (What do you think would happen if you type `display "``rsqf''"`?)

An alternative way to store results for later use is to use *scalars* (type `help scalars` to learn more.) This has the advantage that Stata stores the result in binary form without loss of precision. A macro stores a text representation that is good only for about 8 digits. The downside is that scalars are in the global namespace, so there is a potential for name conflicts, particular in programs (unless you use temporary names, which we discuss later).

You *can* use an equal sign when you are storing text, but this is not necessary, and is not

a good idea if you are using an old version of Stata. The difference is subtle. Suppose we had defined the `controls` macro by saying `local controls = "age agesq education income"`. This would have worked fine, but the quotes cause the right-hand-side to be *evaluated*, in this case as a string, and strings used to be limited to 244 characters (or 80 in Stata/IC before 9.1), whereas macro text can be much longer. Type `help limits` to be reminded of the limits in your version.

5.1.3 Keyboard Mapping with Global Macros

Global macros have names of up to 32 characters and, as the name indicates, have global scope.

You *define* a global macro using `global name [=] text` and *evaluate* it using `$name`. (You may need to use `${name}` to clarify where the name ends.)

I suggest you avoid global macros because of the potential for name conflicts. A useful application, however, is to map the function keys on your keyboard. If you work with a repository on GitHub, for example, try something like this

`global F5 https://raw.githubusercontent.com/`*`username`*`/`*`repositoryname`*`/main`

Then when you hit F5 Stata will substitute the full name. And you can execute a do file in the repository using `do $F5/`*`dofilename`*. (The use of a `/` makes it clear where the macro name ends, and we just append the name of the do file.)

Obviously you don't want to type this macro each time you use Stata. Solution? Enter it in your `profile.do` file, a set of commands that is executed each time you run Stata.

To learn where to store your profile type `help profile` and then follow the link for your operating system, as there are some differences between Windows, Mac and Unix computers.

5.1.4 More on Macros

Macros can also be used to obtain and store information about the system or the variables in your dataset using *extended macro functions*. For example you can retrieve variable and value labels, a feature that can come handy in programming.

There are also commands to manage your collection of macros, including `macro list` and `macro drop`. Type `help macro` to learn more.

5.2 Looping

Loops are used to do repetitive tasks. Stata has commands that allow looping over sequences of numbers and various types of lists, including lists of variables.

Before we start, however, don't forget that Stata does a lot of looping all by itself. If you want to compute the log of income, you can do that in Stata with a single line:

```
gen logincome = log(income)
```

This loops implicitly over all observations, computing the log of each income, in what is sometimes called a *vectorized* operation. You could code the loop yourself, but you shouldn't because (i) you don't need to, and (ii) your code will be a lot slower that Stata's built-in loop.

5.2.1 Looping Over Sequences of Numbers

The basic looping command takes the form

```
forvalues number = sequence {
    ... body of loop using `number' ...
}
```

Here `forvalues` is a keyword, `number` is the name of a local macro that will be set to each number in the sequence, and `sequence` is a range of values which can have the form

- `min/max` to indicate a sequence of numbers from `min` to `max` in steps of one, for example `1/3` yields 1, 2 and 3, or
- `first(step)last` which yields a sequence from `first` to `last` in steps of size `step`. For example `15(5)50` yields 15,20,25,30,35,40,45 and 50.

(There are two other ways of specifying the second type of sequence, but I find the one listed here the clearest, see `help forvalues` for the alternatives.)

The opening left brace must be the last thing on the first line (other than comments), and the loop must be closed by a matching right brace on a line all by itself. The loop is executed once for each value in the sequence with your local macro `number` (or whatever you called it) holding the value.

Creating Dummy Variables Here's my favorite way of creating dummy variables to represent age groups. Stata 11 introduced factor variables and Stata 13 improved the labeling of tables of estimates, so there's really no need to "roll your own" dummies, but the code remains instructive.

```
forvalues bot = 20(5)45 {
    local top = `bot' + 4
    gen age`bot'to`top' = age >= `bot' & age <= `top'
}
```

This will create dummy variables `age20to24` to `age45to49`. The way the loop works is that the local macro `bot` will take values between 20 and 45 in steps of 5 (hence 20, 25, 30, 35, 40, and 45), the lower bounds of the age groups.

Inside the loop we create a local macro `top` to represent the upper bounds of the age groups, which equals the lower bound plus 4. The first time through the loop `bot` is 20, so `top` is 24. We use an equal sign to store the result of adding 4 to `bot`.

The next line is a simple generate statement. The first time through the loop the line will say `gen age20to24 = age >= 20 & age <= 24`, as you can see by doing the macro substitution yourself. This will create the first dummy, and Stata will then go back to the top to create the next one.

5.2.2 Looping Over Elements in a List

The second looping command is `foreach` and comes in six flavors, dealing with different types of lists. I will start with the generic list:

```
foreach item in a-list-of-things {
    ... body of loop using `item' ...
}
```

Here `foreach` is a keyword, `item` is a local macro name of your own choosing, `in` is another keyword, and what comes after is a list of blank-separated words. Try this example

```
foreach animal in cats and dogs {
    display "`animal'"
}
```

This loop will print "cats", "and", and "dogs", as the local macro `animal` is set to each of the words in the list. Stata doesn't know "and" is not an animal, but even if it did, it wouldn't care because the list is generic.

If you wanted to loop over an irregular sequence of numbers –for example you needed to do something with the Coale-Demeny regional model life tables for levels 2, 6 and 12– you could write

```
foreach level in 2 6 12 {
    ... do something with `level' ...
}
```

That's it. This is probably all you need to know about looping.

5.2.3 Looping Over Specialized Lists

Stata has five other variants of `foreach` which loop over specific types of lists, which I now describe briefly.

Lists of Variables Perhaps the most useful variant is

```
foreach varname of varlist list-of-variables {
    ... body of loop using `varname' ...
}
```

Here `foreach`, `of` and `varlist` are keywords, and must be typed exactly as they are. The `list-of-variables` is just that, a list of *existing* variable names typed using standard Stata conventions, so you can abbreviate names (at your own peril), use `var*` to refer to all variables that start with "var", or type `var1-var3` to refer to variables `var1` to `var3`.

The advantages of this loop over the generic equivalent `foreach varname in list-of-variables` is that Stata checks that each name in the list is indeed an existing variable name, and lets you abbreviate or expand the names.

If you need to loop over *new* as opposed to *existing* variables use `foreach varname of newlist list-of-new-variables`. The `newlist` keyword replaces `varlist` and tells Stata to check that all the list elements are legal names of variables that don't exist already.

Words in Macros Two other variants loop over the words in a local or global macro; they use the keyword `global` or `local` followed by a macro name (in lieu of a list). For example here's a way to list the control variables from the section on local macros:

```
foreach control of local controls {
    display "`control'"
}
```

Presumably you would do something more interesting than just list the variable names. Because we are looping over variables in the dataset we could have achieved the same purpose using `foreach` with a `varlist`; here we save the checking.

Lists of Numbers Stata also has a `foreach` variant that specializes in lists of numbers (or `numlists` in Stataspeak) that can't be handled with `forvalues`.

Suppose a survey had a baseline in 1980 and follow ups in 1985 and 1995. (They actually planned a survey in 1990 but it was not funded.) To loop over these you could use

```
foreach year of numlist 1980 1985 1995 {
    display "`year'"
}
```

Of course you would do something more interesting than just print the years. A numlist may be specified as `1 2 3`, or `1/5` (meaning 1 2 3 4 5), or `1(2)7` (count from 1 to 7 in steps of 2 to get 1 3 5 7); type `help numlist` for more examples.

The advantage of this command over the generic `foreach` is that Stata will check that each of the elements of the list of numbers is indeed a number.

5.2.4 Looping for a While

In common with many programming languages, Stata also has a `while` loop, which has the following structure

```
while condition {
    ... do something ...
}
```

where condition is an expression. The loop executes as long as the condition is true (nonzero). Usually something happens inside the loop to make the condition false, otherwise the code would run forever.

A typical use of `while` is in iterative estimation procedures, where you may loop while the difference in successive estimates exceeds a predefined tolerance. Usually an iteration count is used to detect lack of convergence.

The `continue [,break]` command allows breaking out of any loop, including `while`, `forvalues` and `foreach`. The command stops the current iteration and continues with the next, unless `break` is specified, in which case it exits the loop.

5.2.5 Conditional Execution

Stata also has an `if` programming command, not to be confused with the `if`*qualifier* that can be used to restrict any command to a subset of the data, as in `summarize mpg if foreign`. The `if`*command* has the following structure

```
if expression {
    ... commands to be executed if expression is true ...
}
else {
    ... optional block to be executed if expression is false ...
}
```

Here `if` and the optional `else` are keywords, and `expression` is a logical condition (type `help exp` for an explanation of expressions). The opening brace `{` must be the last thing on a line (other than comments) and the closing brace `}` must be on a new line by itself.

If the `if` or `else` parts consist of a single command they can go on the same line *without* braces, as in `if expression command`. But `if expression { command }` is not legal. You could use the braces by spreading the code into three lines, and this often improves readability of the code.

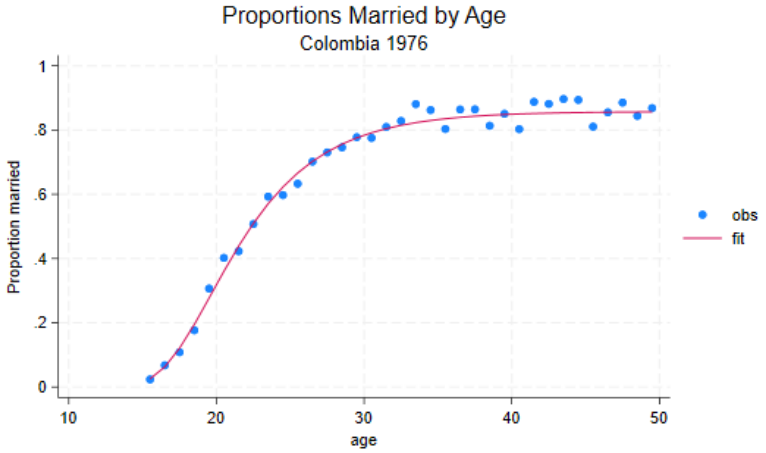
So here we have a silly loop where we break out after five of the possible ten iterations:

```
forvalues iter=1/10 {
    display "`iter'"
    if `iter' >= 5 continue, break
}
```

And with that, we break out of looping.

5.3 Writing Commands

We now turn to the fun task of writing your own Stata commands. Follow along as we develop a few simple programs, ending with an `egen` extension to evaluate the Coale-McNeil model nuptiality schedule, so we can create a plot like the figure below.



5.3.1 Programs With No Arguments

There is a long tradition that the first program one writes in a new language is "Hello World!". The simplest way to develop a new program is to start with a do file:

```
capture program drop hello
program define hello
    version 14
    display as text "hello, world"
end
```

That's it. If you now type `hello` Stata will say "hello, world", in lowercase, with a comma, and without an exclamation mark, just as Brian Kernighan's original.

The `program drop` statement is needed in case we make changes and need to rerun the do file, because you can't define an existing program. The `capture` is needed the very first time, when there is nothing to drop.

The `version` is set to 14 so that users of earlier versions of Stata can run the program. (I will do that for all programs in this section, as they do not rely on more recent features.)

All Stata output goes through SMCL, pronounced "smickle" and short for Stata Markup Control Language. SMCL uses plain text combined with commands enclosed in braces. Below we use a bit of SMCL to frame this inmortal sentence

```
. capture program drop hello
. program define hello
  1.     version 14
  2.     display "{txt}{c TLC}{hline 14}{c TRC}"
  3.     display "{c |} hello, world {c |}"
  4.     display "{c BLC}{hline 14}{c BRC}"
  5. end
. hello
┌──────────────┐
│ hello, world │
└──────────────┘
```

Here `{txt}` sets the style to text (as opposed to input, error or result), `{c TLC}` and its cousins are used to draw top-left, top-right, bottom-left and bottom-right corners, `{hline 14}` draws a horizontal line 14 characters long, and `{c |}` draws a tall `|`. To learn more about SMCL type `help smcl`. This will be essential to write a help file for your commands.

5.3.2 A Program with an Argument

To make useful programs you will often need to pass information to them, in the form of "arguments" you type after the command. Let's write a command that echoes what you say. I used to call it `echo`, but now Stata has its own (undocumented) echo program, so we'll call it `parrot`. (Stata reserves all english words, so you have to be careful naming your programs. You can check if the name is taken by typing `which` followed by the name, in our case `which parrot`.)

```
capture program drop parrot
program define parrot
    version 14
```

```
    display "`0'"
end
```

Try typing `parrot hello, world` to see what happens.

When you call a command Stata stores the arguments in a local macro called `0`. We use a display command with `` `0' `` to evaluate the macro. The result is text, so we enclose it in quotes. (Suppose you typed `parrot hi`, so the local macro `0` has `hi`; the command would read `display hi` and Stata will complain, saying 'hi not found'. We want the command to read `display "hi"`, which is why we code `display "`0'"`.)

If we don't specify anything, the local macro `0` will be an empty string, the command will read `display ""` and Stata will print a blank line.

5.3.3 Compound Quotes

Before we go out to celebrate we need to fix a small problem with our new command. Try typing `parrot The hopefully "final" run`. Stata will complain. Why? Because after macro substitution the all-important display command will read

```
display "The hopefully "final" run"
```

The problem is that the quote before final closes the initial quote, so Stata sees this is as `"The hopefully "` followed by `final" run"`, which looks to Stata like an invalid name. Obviously we need some way to distinguish the inner and outer quotes.

Incidentally you could see exactly where things went south by typing `set trace on` and running the command. You can see in (often painful) detail all the steps Stata goes through, including all macro substitutions. Don't forget to type `set trace off` when you are done. Type `help trace` to learn more.

The solution to our problem? Stata's *compound double quotes*: `` `" `` to open and `"' ` to close, as in `` `"compound quotes"' ``. Because the opening and closing symbols are different, these quotes can be nested. Compound quotes

- *can* be used anywhere a double quote is used.
- *must* be used if the text being quoted includes double quotes.

So our program must `display `"`0'"'`. Here's the final version.

```
program define parrot
    version 14
    if `"`0'"' != "" display as text `"`0'"'
end
```

You will notice that I got rid of the `capture drop line`. I also used `display as text` to make sure we print using the text style. For error messages you may want to use `display as error` instead. Type `help display` to learn more about this command.

We are now ready to save the program as an `ado` file. Type `sysdir` to find out where your personal ado directory is, and then save the file there with the name `parrot.ado`. The command will now be available any time you use Stata.

5.3.4 Positional Arguments

In addition to storing all arguments together in local macro `0`, Stata parses the arguments (using white space as a delimiter) and stores all the words it finds in local macros `1`, `2`, `3`, etc.

Typically you would do something with `` `1' `` and then move on to the next one. The command `mac shift` comes handy then, because it shifts all the macros down by one, so the contents of `2` is now in `1`, and `3` is in `2`, and so on. This way you always work with what's in `1` and shift down. When the list is exhausted `1` is empty and you are done.

So here is the canonical program that lists its arguments

```
capture program drop listargs
program define listargs
    version 14
    while "`1'" != "" {
        display `"`1'"'
        mac shift
    }
end
```

Don't forget the `mac shift`, otherwise your program may run forever. (Or until you hit the break key.)

Try `listargs one two three testing`. Now try `listargs one "two and three" four`. Notice how one can group words into a single argument by using quotes.

This method is useful, and sometimes one can given the arguments more meaningful names using `args`. We will give an example in 5.3.8. But let us discuss Stata syntax first, a more powerful and robust approach.

(By the way one can pass arguments not just to commands, but to *do* files as well. Type `help do` to learn more.)

5.3.5 Using Stata Syntax

If your command uses standard Stata syntax –which means the arguments are a list of variables, possibly a weight, maybe an `if` or `in` clause, and perhaps a bunch of *options*– you can take advantage of Stata's own parser, which conveniently stores all these elements in local macros ready for you to use.

A Command Prototype Let us write a command that computes the probability of marrying by a certain age in a Coale-McNeil model with a given mean, standard deviation, and proportion marrying. The syntax of our proposed command is

```
pnupt age, generate(married) [ mean(25) stdev(5) pem(1) ]
```

So we require an existing variable with age in exact years, and a mandatory option specifying a new variable to be generated with the proportions married. There are also options to

specify the mean, the standard deviation, and the proportion ever married in the schedule, all with defaults. Here's a first cut at the command

```
capture program drop pnupt
program define pnupt
    version 14
    syntax varname, Generate(name) ///
        [ Mean(real 25) Stdev(real 5) Pem(real 1) ]
    // ... we don't do anything yet ...
end
```

The first thing to note is that the `syntax` command looks remarkably like our prototype. That's how easy this is.

Variable Lists The first element in our syntax is an example of a *list of variables* or `varlist`. You can specify minima and maxima, for example a program requiring exactly two variables would say `varlist(min=2 max=2)`. When you have only one variable, as we do, you can type `varname`, which is short for `varlist(min=1 max=1)`.

Stata will then make sure that your program is called with exactly one name of an existing variable, which will be stored in a local macro called `varlist`. (The macro is always called `varlist`, even if you have only one variable and used `varname` in your syntax statement.) Try `pnupt nonesuch` and Stata will complain, saying "variable nonesuch not found".

(If you have done programming before, and you spent 75% of your time writing checks for input errors and only 25% focusing on the task at hand, you will really appreciate the `syntax` command. It does a lot of error checking for you.)

Options and Defaults Optional syntax elements are enclosed in square brackets `[` and `]`. In our command the `generate` option is required, but the other three are optional. Try these commands to generate a little test dataset with an age variable ranging from 15 to 50

```
drop _all
set obs 36
gen age = 14 + _n
```

Now try `pnupt age`. This time Stata is happy with `age` but notes 'option generate() required'. Did I say `syntax` saves a lot of work? Options that take arguments need to specify the type of argument (`integer`, `real`, `string`, `name`) and, optionally, a default value. Our `generate` takes a `name`, and is required, so there is no default. Try `pnupt age, gen(2)`. Stata will complain that 2 is not a name.

If all is well, the contents of the option is stored in a local macro with the same name as the option, here `generate`.

Checking Arguments Now we need to do just a bit of work to check that the name is a valid variable name, which we do with `confirm`:

```
confirm new variable `generate'
```

Stata then checks that you could in fact generate this variable, and if not issues error 110. Try `pnupt age, gen(age)` and Stata will say 'age already defined'.

It should be clear by now that Stata will check that if you specify a mean, standard deviation or proportion ever married, abbreviated as `m()`, `s()` and `p()`, they will be real numbers, which will be stored in local macros called `mean`, `stdev`, and `pem`. If an option is omitted the local macro will contain the default.

You could do more checks on the input. Let's do a quick check that all three parameters are non-negative and the proportion is no more than one.

```
if (`mean' <= 0 | `stdev' <= 0 | `pem' <= 0 | `pem' > 1) {
    di as error "invalid parameters"
    exit 110
}
```

You could be nicer to your users and have separate checks for each parameter, but this will do for now.

Temporary Variables We are now ready to do some calculations. We take advantage of the relation between the Coale-McNeil model and the gamma distribution, as explained in Rodríguez and Trussell (1980). Here's a working version of the program

```
program define pnupt
*! Coale-McNeil cumulative nuptiality schedule v1 GR 24-Feb-06
    version 14
    syntax varname, Generate(name) [Mean(real 25) Stdev(real 5) Pem(real 1)]
    confirm new var `generate'
    if `mean' <= 0 | `stdev' <= 0 | `pem' <= 0 | `pem' > 1 {
        display as error "invalid parameters"
        exit 198
    }
    tempname z g
    gen `z' = (`varlist' - `mean')/`stdev'
    gen `g' = gammap(0.604, exp(-1.896 * (`z' + 0.805)))
    gen `generate' = `pem' * (1 - `g')
end
```

We could have written the formula for the probability in one line, but only by sacrificing readability. Instead we first standardize age, by subtracting the mean and dividing by the standard deviation. What can we call this variable? You might be tempted to call it `z`, but what if the user of your program has a variable called `z`? Later we evaluate the gamma function. What can we call the result?

The solution is the `tempname` command, which asks Stata to make up unique temporary variable names, in this case two to be stored in local macros `z` and `g`. Because these macros are local, there is no risk of name conflicts. Another feature of temporary variables is that they disappear automatically when your program ends, so Stata does the housekeeping for you.

The line `gen `z' = (`varlist' - `mean')/`stdev'` probably looks a bit strange at first.

Remember that all names and values of interest are now stored in local macros and we need to evaluate them to get anywhere, hence the profusion of backticks: `` `z' `` gets the name of our temporary variable, `` `varlist' `` gets the name of the age variable specified by the user, `` `mean' `` gets the value of the mean, and `` `stdev' `` gets the value of the standard deviation. After macro substitution this line will read something like `gen _000001 = (age-22.44)/5.28`, which probably makes a lot more sense.

If/In You might consider allowing the user to specify `if` and `in` conditions for your command. These would need to be added to the syntax, where they would be stored in local macros, which can then be used in the calculations, in this case passed along to generate.

For a more detailed discussion of this subject type `help syntax` and select `if` and then `in`. The entry in `help mark` is also relevant.

5.3.6 Creating New Variables

Sometimes all your command will do is create a new variable. This, in fact, is what our little command does. Wouldn't it be nice if we could use an `egen` type of command like this:

```
egen married = pnupt(age), mean(22.48) stdev(5.29) pem(0.858)
```

Well, we can! As it happens, `egen` is user-extendable. To implement a function called `pnupt` you have to create a program (ado file) called `_gpnupt`, in other words add the prefix `_g`. The documentation on egen extensions is a bit sparse, but once you know this basic fact all you need to do is look at the source of an `egen` command and copy it. (I looked at `_gmean`.)

So here's the `egen` version of our Coale-McNeil command.

```
program define _gpnupt
*! Coale-McNeil cumulative nuptiality schedule v1 GR 24-Feb-06
    version 14
    syntax newvarname=/exp [, Mean(real 25) Stdev(real 5) Pem(real 1)]
    if `mean' <= 0 | `stdev' <= 0 | `pem' <= 0 | `pem' > 1 {
        display as error "invalid parameters"
        exit 198
    }
    tempname z g
    gen `z' = (`exp' - `mean')/`stdev'
    gen `g' = gammap(0.604, exp(-1.896 * (`z' + 0.805)))
    gen `typlist' `varlist' = `pem' * (1 - `g')
end
```

There are very few differences between this program and the previous one. Instead of an input variable `egen` accepts an expression, which gets evaluated and stored in a temporary variable called `exp`. The output variable is specified as a `varlist`, in this case a `newvarname`. That's why `z` now works with `exp`, and `gen` creates `varlist`. The mysterious `typlist` is there because `egen` lets you specify the type of the output variable (`float` by default) and that gets passed to our function, which passes it along to `gen`.

5.3.7 A Coale-McNeil Fit

We are ready to reveal how the initial plot was produced. The data are available in a Stata file in the datasets section of my website, which has counts of ever married and single women by age. We compute the observed proportion married, compute fitted values based on the estimates in Rodríguez and Trussell (1980), and plot the results. It's all done in a handful of lines

```
. use https://grodri.github.io/datasets/cohhnupt, clear
(WFS Colombia Household Survey)
. gen agem = age + 0.5
. gen obs = ever/total
. egen fit = pnupt(agem), mean(22.44) stdev(5.28) pem(.858)
. twoway (scatter obs agem) (line fit agem), ///
>     title(Proportions Married by Age) subtitle(Colombia 1976) ///
>     ytitle(Proportion married) xtitle(age)
. graph export cohhnup.png,     width(550) replace
file cohhnup.png saved as PNG format
```

The actual estimation can be implemented using Stata's maximum likelihood procedures, but that's a story for another day.

5.3.8 Returning Results

So far our commands have printed results or created a new variable. How do you return results to the user? A general command can declare itself to be `rclass` and then return results in `r()`, while an estimation command can declare `eclass` and return results in `e()`. Let us illustrate the former with a command to compute Tukey's trimean $T = (Q_1 + 2\ Q_2 + Q_3)/4$, a weighted average of the quartiles and median.

```
. capture program drop trimean
. program trimean, rclass
  1.     version 14
  2.     args varname
  3.     confirm variable `varname´
  4.     quietly summarize `varname´, detail
  5.     if r(N) == 0 {
  6.         display as error "No observations"
  7.         exit
  8.     }
  9.     tempname trimean
 10.     scalar `trimean´ = (r(p25) + 2*r(p50) + r(p75))/4
 11.     display "trimean = ", `trimean´
 12.     return scalar trimean = `trimean´
 13. end
```

The first thing to notice is that the `program` statement includes the `rclass` option. This is required to be able to return results.

We then use `args` to name the single argument `varname`, and we then use `confirm variable` to check that the variable exists. The quartiles we need are computed by `summarize` with the `detail` option, which we do `quietly` to skip printing the results. If the variable is a string or all values are missing, `summarize` will set `r(N)` to 0, in which case we display an error message and exit.

We could store the trimean in a local macro, but we will lose precision. Instead we use a

scalar. The `tempname` line will store a unique name in the local macro `trimean`. We then store the trimean in a scalar with that name, and print it. The final step uses the `return` statement to store the scalar result in `r(trimean)`. Let us run the program and then list the results

```
. sysuse auto, clear
(1978 automobile data)
. trimean mpg
trimean =  20.75
. return list
scalars:
            r(trimean) =  20.75
```

Each `rclass` command erases the results of the previous one. However we could use `return add` to add our results to what's on `r()` already. Try it. After running our command, `return list` will list 20 results, ready for inclusion in a customized table.

We could expand the program to more variables, perhaps using `syntax varlist`, but note that `summarize` only stores results for the last variable listed, and our command should probably do the same.

5.4 Other Topics

To keep this tutorial from becoming too long I have skipped or cut short many topics. To learn more about returning results from your commands type `help return`. For estimation commands, which can post estimation results to `e()`, see `help ereturn` and `help _estimates`. An essential reference on estimation is *Maximum Likelihood Estimation with Stata*, Fourth Edition, by Gould, Pitblado and Poi (2010).

Other subjects of interest are matrices (start with `help matrix`), and how to make commands "byable" (type `help byable`). To format your output you need to learn more about SMCL, start with `help smcl`. For work on graphics you may want to study class programming (`help class`) and learn about sersets (`help serset`). To provide a graphical user interface to your command try `help dialog programming`. It is also possible to read and write text and binary files (see `help file`).

The biggest omission here is Mata, a full-fledged matrix programming language that was introduced in Version 9 of Stata. Mata is compiled to byte code, so it is much faster than Stata's classic ado programs. I find that the best way to write new Stata commands is to use classic ado for the user interface and Mata for the actual calculations. If you are interested in learning Mata I strongly recommend Gould's (2018) *The Mata Book*.

References

Acock, Alan C. 2023. *A Gentle Introduction to Stata*. Revised Sixth Edition. College Station, TX: Stata Press.

Baum, Christopher F. 2016. *An Introduction to Stata Programming*. 2nd edition. College Station, TX: Stata Press.

Cleves, Mario, William Gould, and Julia Marchenko. 2016. *An Introduction to Survival Analysis Using Stata*. Revised 3rd edition. College Station, TX: Stata Press.

Gould, William W. 2018. *The Mata Book, a Book for Serious Programmers and Those Who Want to Be*. College Station, TX: Stata Press.

Gould, William W., Jeffrey Pitblado, and Brian Poi. 2010. *Maximum Likelihood Estimation with Stata*. College Station, TX: Stata Press.

Long, Scott, and Jeremy Freese. 2014. *Regression Models for Categorical Dependent Variables Using Stata*. 3rd edition. College Station, TX: Stata Press.

Mitchell, Michael N. 2022. *A Visual Guide to Stata Graphics*. 4th edition. College Station, TX: Stata Press.

Rodríguez, Germán, and James Trussell. 1980. "Maximum Likelihood Estimation of the Parameters of Coale's Model Nuptiality Schedule from Survey Data." Technical Bulletin 7. World Fertility Survey.