Using Stata for data management and reproducible research

Christopher F Baum

Boston College and DIW Berlin

IMF Institute, Spring 2011

Overview of the Stata environment

Stata is a full-featured statistical programming language for Windows, Mac OS X, Unix and Linux. It can be considered a “stat package,” like SAS, SPSS, RATS, or eViews.

Stata is available in several versions: Stata/IC (the standard version), Stata/SE (an extended version) and Stata/MP (for multiprocessing). The major difference between the versions is the number of variables allowed in memory, which is limited to 2,047 in standard Stata/IC, but can be much larger in Stata/SE or Stata/MP. The number of observations in any version is limited only by memory.

Stata/SE relaxes the Stata/IC constraint on the number of variables, while Stata/MP is the multiprocessor version, capable of utilizing 2, 4, 8... processors available on a single computer. Stata/IC will meet most users' needs; if you have access to Stata/SE or Stata/MP, you can use that program to create a subset of a large survey dataset with fewer than 2,047 variables. Stata runs on all 64-bit operating systems, and can access larger datasets on a 64-bit OS, which can address a larger memory space.

All versions of Stata provide the full set of features and commands: there are no special add-ons or 'toolboxes'. Each copy of Stata 11 includes a complete set of manuals (over 6,000 pages) in PDF format, hyperlinked to the on-line help.

A Stata license may be used on any machine which supports Stata (Mac OS X, Windows, Linux): there are no machine-specific licenses for Stata 11. You may install Stata on a home and office machine, as long as they are not used concurrently. Licenses can be either annual or perpetual.

Stata works differently than some other packages in requiring that the entire dataset to be analyzed must reside in memory. This brings a considerable speed advantage, but implies that you may need more RAM (memory) on your computer. There are 32-bit and 64-bit versions of Stata, with the major difference being the amount of memory that the operating system can allocate to Stata (or any other application).

In some cases, the memory requirement may be of little concern. Stata is capable of holding data very efficiently, and even a quite sizable dataset (e.g., more than one million observations on 20–30 variables) may only require 500 Mb or so. You should take advantage of the `compress` command, which will check to see whether each variable may be held in fewer bytes than its current allocation.

For instance, indicator (dummy) variables and categorical variables with fewer than 100 levels can be held in a single byte, and integers less than 32,000 can be held in two bytes: see `help datatypes` for details. By default, floating-point numbers are held in four bytes, providing about seven digits of accuracy. Some other statistical programs routinely use eight bytes to store all numeric variables.

The memory available to Stata may be considerably less than the amount of RAM installed on your computer. If you have a 32-bit operating system, it does not matter that you might have 4 Gb or more of RAM installed; Stata will only be able to access about 1 Gb, depending on other processes' demands.

To make most effective use of Stata with large datasets, use a computer with a 64-bit operating system. Stata will automatically install a 64-bit version of the program if it is supported by the operating system. All Linux, Unix and Mac OS X computers today come with 64-bit operating systems.

Stata is eminently portable, and its developers are committed to cross-platform compatibility. Stata runs the same way on Windows, Mac OS X, Unix, and Linux systems. The only platform-specific aspects of using Stata are those related to native operating system commands: e.g. is the file to be accessed

```
C:\Stata\StataData\myfile.dta
or
/users/baum/statadata/myfile.dta
```

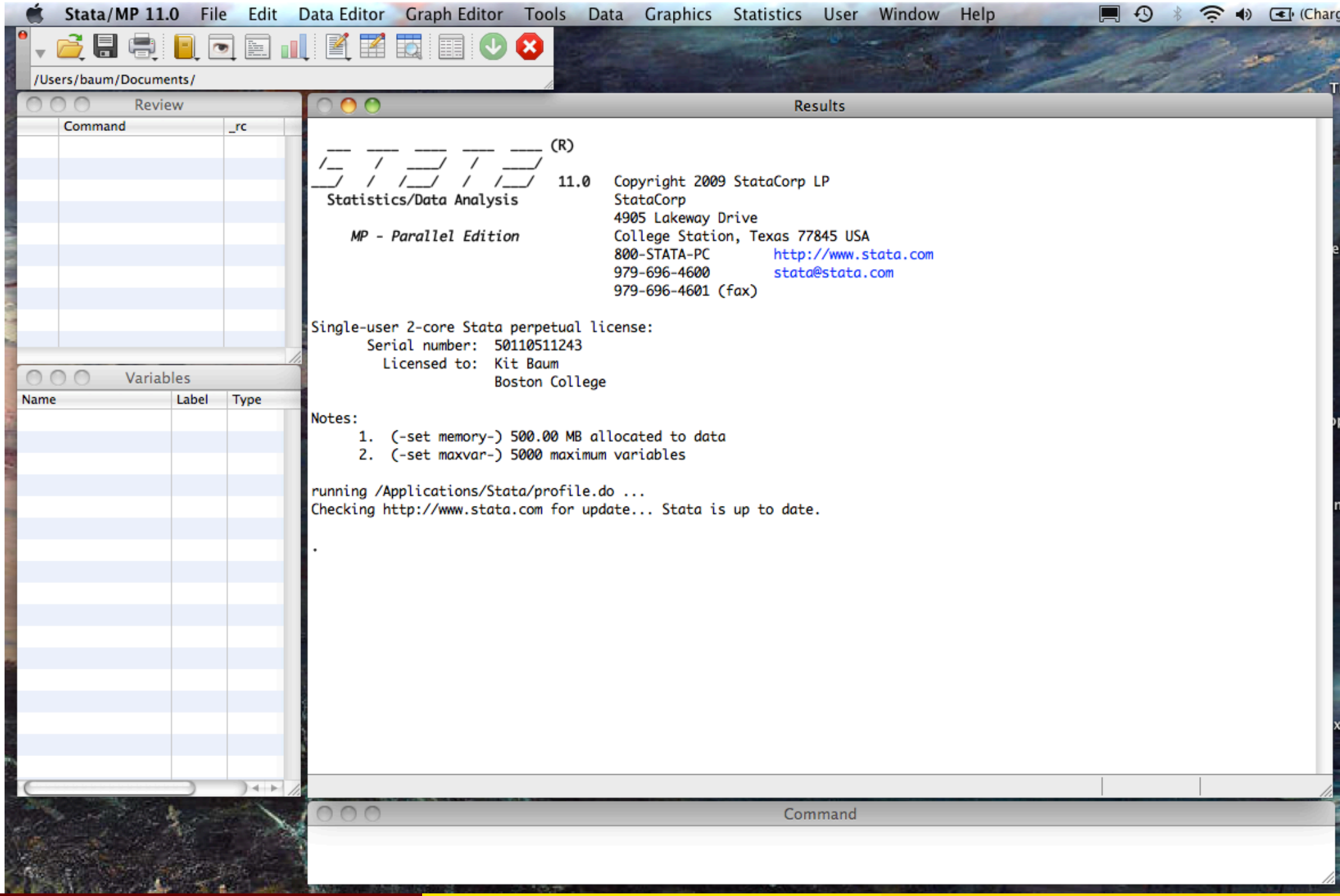
Perhaps unique among statistical packages, Stata's binary data files may be freely copied from one platform to any other, or even accessed over the Internet from any machine that runs Stata. You may store Stata's binary datafiles on a webserver (HTTP server) and open them on any machine with access to that server.

Stata's user interface

Stata has traditionally been a command-line-driven package that operates in a graphical (windowed) environment. Stata version 11 (released June 2009) contains a graphical user interface (GUI) for command entry via menus and dialogs. Stata may also be used in a command-line environment on a shared system (e.g., a Unix server) if you do not have a graphical interface to that system.

A major advantage of Stata's GUI system is that you always have the option of reviewing the command that has been entered in Stata's Review window. Thus, you may examine the syntax, revise it in the Command window and resubmit it. You may find that this is a more efficient way of using the program than relying wholly on dialogs.

Stata's default screen appearance:



The Toolbar contains icons that allow you to Open and Save files, Print results, control Logs, and manipulate windows. Some very important tools allow you to open the Do-File Editor, the Data Editor and the Data Browser.

The Data Editor and Data Browser present you with a spreadsheet-like view of the data, no matter how large your dataset may be. The Do-File editor, as we will discuss, allows you to construct a file of Stata commands, or "do-file", and execute it in whole or in part from the editor.

The Toolbar also contains an important piece of information: the Current Working Directory, or *cwd*. In the screenshot, it is listed as `/Users/Baum/Documents/` as I am working on a Mac OS X (Unix) laptop. The *cwd* is the directory to which any files created in your Stata session will be saved. Likewise, if you try to open a file and give its name alone, it is assumed to reside in the *cwd*. If it is in another location, you must change the *cwd* [File− >Change Working Directory] or qualify its name with the directory in which it resides.

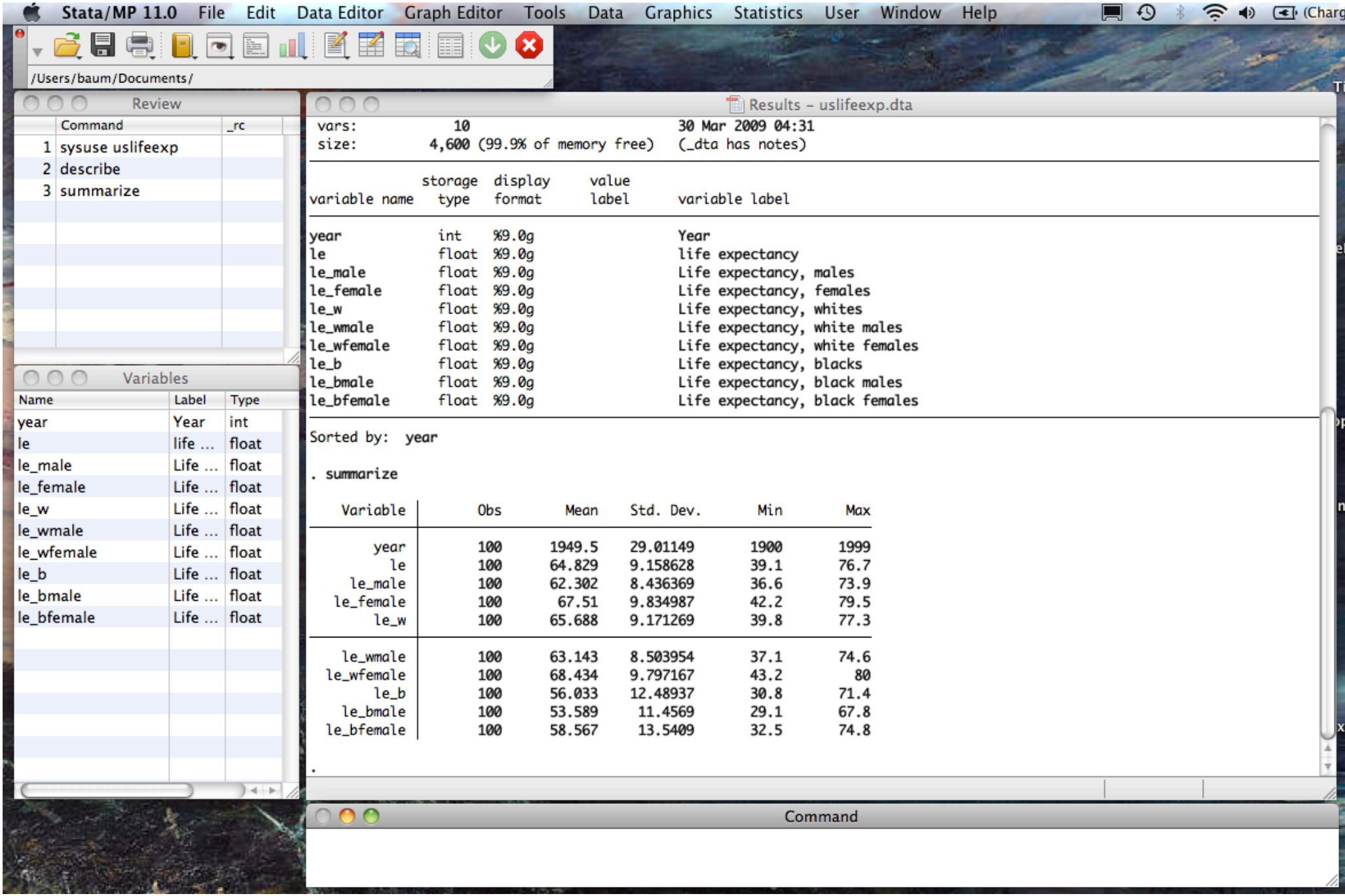
You generally will not want to locate or save files in the default *cwd*. A common strategy is to set up a directory for each project or task in a convenient location in the filesystem and change the *cwd* to that directory when working on that task. This can be automated in a do-file with the `cd` command.

There are four windows in the default interface: the Review, Results, Command and Variables window. You may alter the appearance of any window in the GUI using the Preferences$->$General dialog, and make those changes on a temporary or permanent basis.

As you might expect, you may type commands in the Command window. You may only enter one command in that window, so you should not try pasting a list of several commands. When a command is executed—with or without error—it appears in the Review window, and the results of the command (or an error message) appears in the Results window. You may click on any command in the Review window and it will reappear in the Command window, where it may be edited and resubmitted.

Once you have loaded data into the program, the Variables window will be populated with information on each variable. That information includes the variable name, its label (if any), its type and its format. This is a subset of information available from the `describe` command.

Let's look at the interface after I have loaded one of the datasets provided with Stata, `uslifeexp`, with the `sysuse` command and given the `describe` and `summarize` commands:


Stata/MP 11.0 File Edit Data Editor Graph Editor Tools Data Graphics Statistics User Window Help
/Users/baum/Documents/
Review
Command _rc
1 sysuse uslifeexp
2 describe
3 summarize
Variables
Name Label Type
year Year int
le life ... float
le_male Life ... float
le_female Life ... float
le_w Life ... float
le_wmale Life ... float
le_wfemale Life ... float
le_b Life ... float
le_bmale Life ... float
le_bfemale Life ... float
Results – uslifeexp.dta
vars: 10 30 Mar 2009 04:31
size: 4,600 (99.9% of memory free) (_dta has notes)
variable name storage type display format value label variable label
year int %9.0g Year
le float %9.0g life expectancy
le_male float %9.0g Life expectancy, males
le_female float %9.0g Life expectancy, females
le_w float %9.0g Life expectancy, whites
le_wmale float %9.0g Life expectancy, white males
le_wfemale float %9.0g Life expectancy, white females
le_b float %9.0g Life expectancy, blacks
le_bmale float %9.0g Life expectancy, black males
le_bfemale float %9.0g Life expectancy, black females
Sorted by: year
. summarize
Variable Obs Mean Std. Dev. Min Max
year 100 1949.5 29.01149 1900 1999
le 100 64.829 9.158628 39.1 76.7
le_male 100 62.302 8.436369 36.6 73.9
le_female 100 67.51 9.834987 42.2 79.5
le_w 100 65.688 9.171269 39.8 77.3
le_wmale 100 63.143 8.503954 37.1 74.6
le_wfemale 100 68.434 9.797167 43.2 80
le_b 100 56.033 12.48937 30.8 71.4
le_bmale 100 53.589 11.4569 29.1 67.8
le_bfemale 100 58.567 13.5409 32.5 74.8
Command

Notice that the three commands are listed in the Review window. If any had failed, the `_rc` column would contain a nonzero number, in red, indicating the error code. The Variables window contains the list of variables and their labels. The Results window shows the effects of `summarize`: for each variable, the number of observations, their mean, standard deviation, minimum and maximum. If there were any string variables in the dataset, they would be listed as having zero observations.

Try it out: type the commands

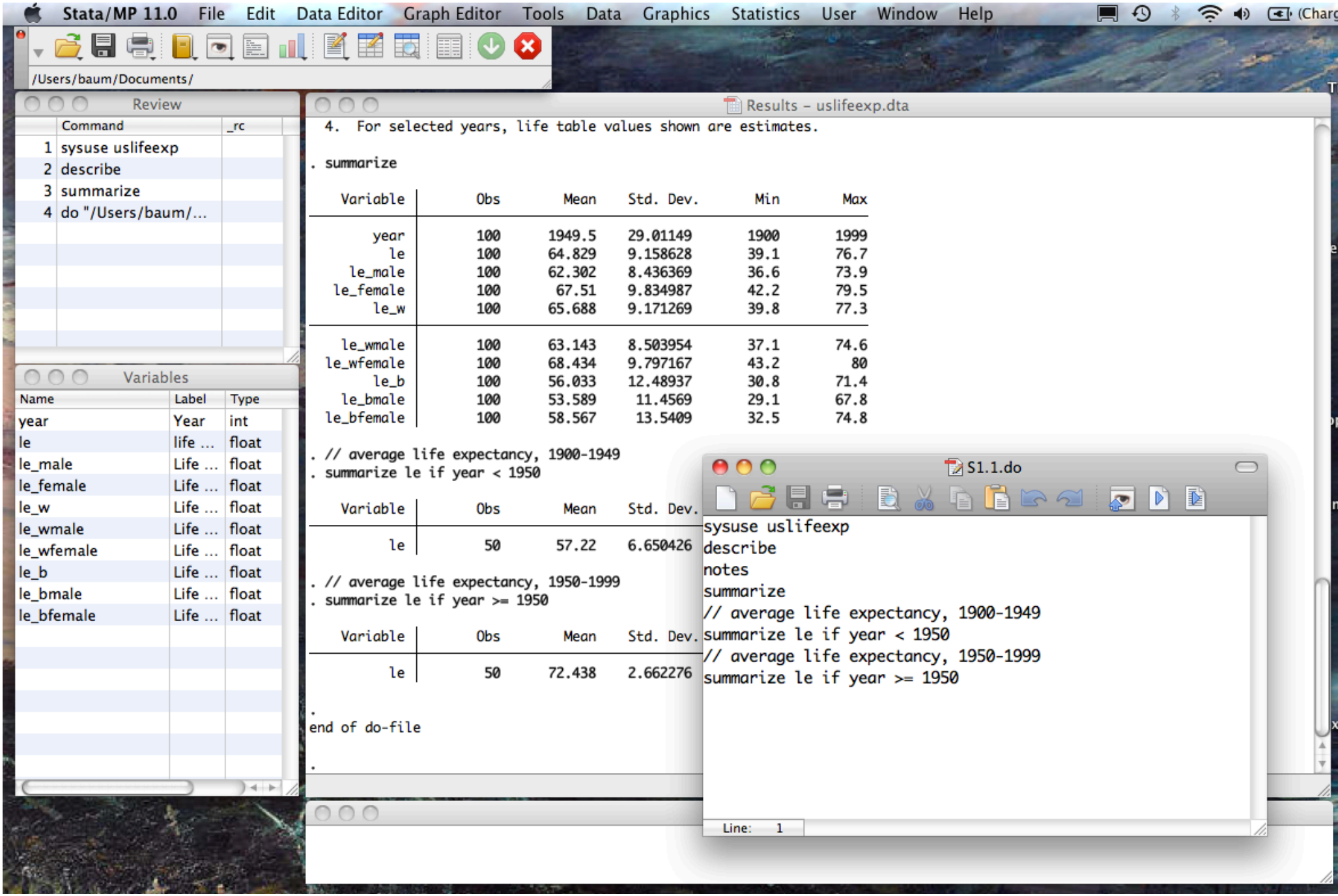
```
sysuse uslifeexp
describe
summarize
```

Take note of an important design feature of Stata. If you do not say what to `describe` or `summarize`, Stata assumes you want to perform those commands for every variable in memory, as shown here. As we shall see, this design principle holds throughout the program.

We may also write a do-file in the do-file editor and execute it. The Do-File Editor icon on the Toolbar brings up a window in which we may type those same three commands, as well as a few more:

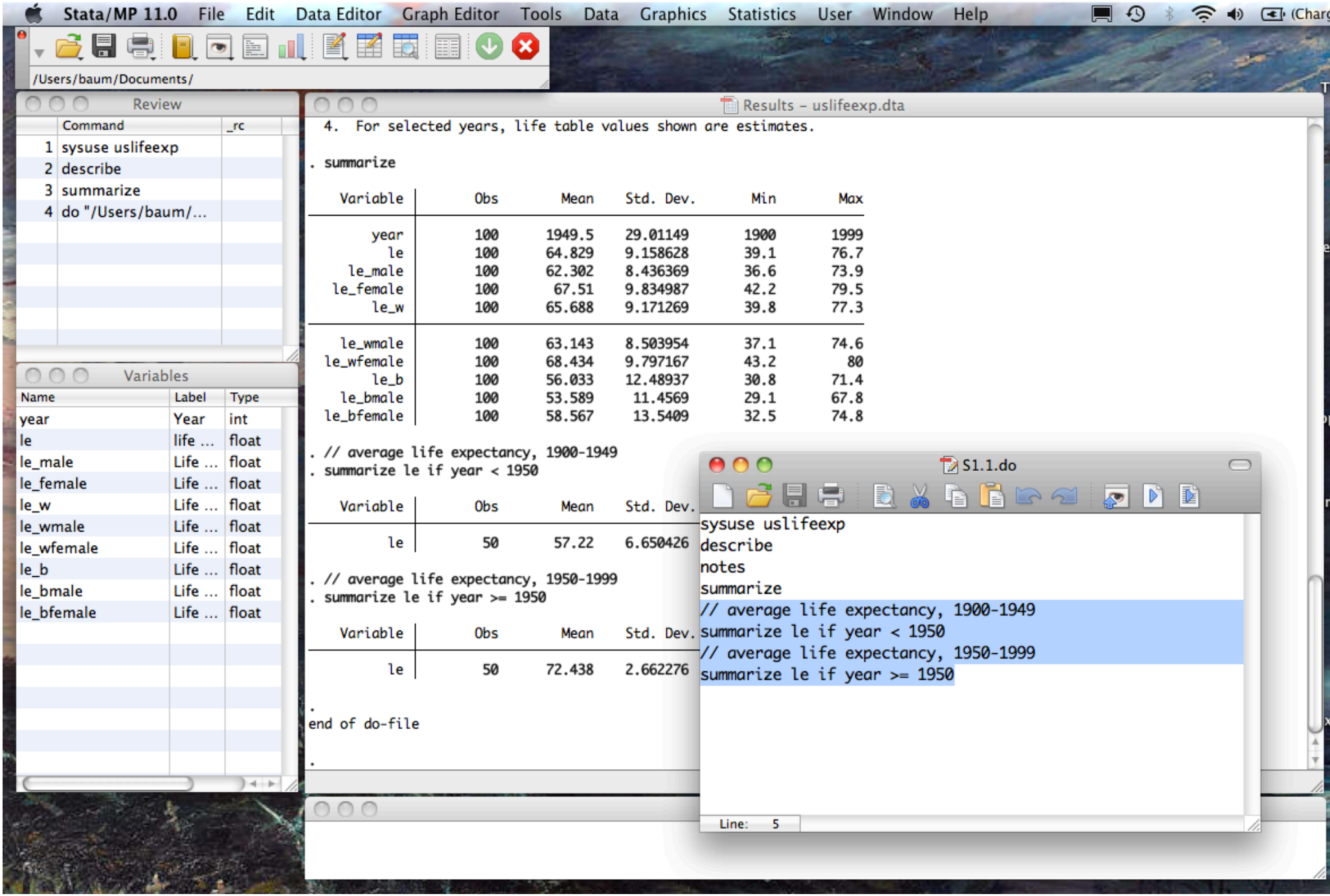
```
sysuse uslifeexp
describe
summarize
notes
summarize le if year < 1950
summarize le if year >= 1950
```

After typing those commands into the window, the rightmost icon, with tooltip `Do`, may be used to execute them.


Stata/MP 11.0 File Edit Data Editor Graph Editor Tools Data Graphics Statistics User Window Help
/Users/baum/Documents/
Review
Command _rc
1 sysuse uslifeexp
2 describe
3 summarize
4 do "/Users/baum/...
Variables
Name Label Type
year Year int
le life ... float
le_male Life ... float
le_female Life ... float
le_w Life ... float
le_wmale Life ... float
le_wfemale Life ... float
le_b Life ... float
le_bmale Life ... float
le_bfemale Life ... float
Results – uslifeexp.dta
4. For selected years, life table values shown are estimates.
. summarize
Variable Obs Mean Std. Dev. Min Max
year 100 1949.5 29.01149 1900 1999
le 100 64.829 9.158628 39.1 76.7
le_male 100 62.302 8.436369 36.6 73.9
le_female 100 67.51 9.834987 42.2 79.5
le_w 100 65.688 9.171269 39.8 77.3
le_wmale 100 63.143 8.503954 37.1 74.6
le_wfemale 100 68.434 9.797167 43.2 80
le_b 100 56.033 12.48937 30.8 71.4
le_bmale 100 53.589 11.4569 29.1 67.8
le_bfemale 100 58.567 13.5409 32.5 74.8
. // average life expectancy, 1900-1949
. summarize le if year < 1950
Variable Obs Mean Std. Dev.
le 50 57.22 6.650426
. // average life expectancy, 1950-1999
. summarize le if year >= 1950
Variable Obs Mean Std. Dev.
le 50 72.438 2.662276
.
end of do-file
.
S1.1.do
sysuse uslifeexp
describe
notes
summarize
// average life expectancy, 1900-1949
summarize le if year < 1950
// average life expectancy, 1950-1999
summarize le if year >= 1950
Line: 1

In this do-file, I have included the `notes` command to display the notes saved with the dataset, and included two comment lines. There are several styles of comments available. In this style, anything on a line following a double slash (//) is ignored.

You may use the other icons in the Do-File Editor window to save your do-file (to the *cwd* or elsewhere), print it, or edit its contents. You may also select a portion of the file with the mouse and execute only those commands. Note that the tooltip changes to `Do Selected Lines`.


Stata/MP 11.0 File Edit Data Editor Graph Editor Tools Data Graphics Statistics User Window Help
/Users/baum/Documents/
Review
Command _rc
1 sysuse uslifeexp
2 describe
3 summarize
4 do "/Users/baum/...
Variables
Name Label Type
year Year int
le life ... float
le_male Life ... float
le_female Life ... float
le_w Life ... float
le_wmale Life ... float
le_wfemale Life ... float
le_b Life ... float
le_bmale Life ... float
le_bfemale Life ... float
Results – uslifeexp.dta
4. For selected years, life table values shown are estimates.
. summarize
Variable Obs Mean Std. Dev. Min Max
year 100 1949.5 29.01149 1900 1999
le 100 64.829 9.158628 39.1 76.7
le_male 100 62.302 8.436369 36.6 73.9
le_female 100 67.51 9.834987 42.2 79.5
le_w 100 65.688 9.171269 39.8 77.3
le_wmale 100 63.143 8.503954 37.1 74.6
le_wfemale 100 68.434 9.797167 43.2 80
le_b 100 56.033 12.48937 30.8 71.4
le_bmale 100 53.589 11.4569 29.1 67.8
le_bfemale 100 58.567 13.5409 32.5 74.8
. // average life expectancy, 1900-1949
. summarize le if year < 1950
Variable Obs Mean Std. Dev.
le 50 57.22 6.650426
. // average life expectancy, 1950-1999
. summarize le if year >= 1950
Variable Obs Mean Std. Dev.
le 50 72.438 2.662276
.
end of do-file
.
S1.1.do
sysuse uslifeexp
describe
notes
summarize
// average life expectancy, 1900-1949
summarize le if year < 1950
// average life expectancy, 1950-1999
summarize le if year >= 1950
Line: 5

Try it out: use the Do-File Editor to open the do-file `S1.1.do`, and run the file.

Try selecting only those last four lines and run those commands.

The rightmost menu on the menu bar is labeled Help. From that menu, you can search for help on any command or feature. The Help Browser, which opens in a Viewer window, provides hyperlinks, in blue, to additional help pages. At the foot of each help screen, there are hyperlinks to the full manuals, which are accessible in PDF format. The links will take you directly to the appropriate page of the manual.

You may also search for help at the command line with `help` *command*. But what if you don't know the exact command name? Then you may use `search` or its expanded version, `findit`, each of which may be followed by one or several words.

Results from `search` are presented in the Results window, while `findit` results will appear in a Viewer window. Those commands will present results from a keyword database and from the Internet: for instance, FAQs from the Stata website, articles in the *Stata Journal* and *Stata Technical Bulletin*, and downloadable routines from the SSC Archive (about which more later) and user sites.

Try it out: when you are connected to the Internet, type the command

```
search baum, au
```

and then try

```
findit baum
```

Note the hyperlinks that appear on URLs for the books and journal articles, and on the individual software packages (e.g., `st0030_3`, `archlm`).

Stata's update facility

One of Stata's great strengths is that it can be updated over the Internet. Stata is actually a web browser, so it may contact Stata's web server and enquire whether there are more recent versions of either Stata's executable (the kernel) or the ado-files. This enables Stata's developers to distribute bug fixes, enhancements to existing commands, and even entirely new commands during the lifetime of a given major release (including 'dot-releases' such as Stata 11.1).

Updates during the life of the version you own are free. You need only have a licensed copy of Stata and access to the Internet (which may be by proxy server) to check for and, if desired, download the updates.

Extensibility of official Stata

Another advantage of the command-line driven environment involves *extensibility*: the continual expansion of Stata's capabilities. A *command*, to Stata, is a verb instructing the program to perform some action.

Commands may be "built in" commands—those elements so frequently used that they have been coded into the "Stata kernel." A relatively small fraction of the total number of official Stata commands are built in, but they are used very heavily.

The vast majority of Stata commands are written in Stata's own programming language–the "ado-file" language. If a command is not built in to the Stata kernel, Stata searches for it along the `adopath`. Like the `PATH` in Unix, Linux or DOS, the `adopath` indicates the several directories in which an ado-file might be located. This implies that the "official" Stata commands are not limited to those coded into the kernel. *Try it out*: give the `adopath` command in Stata.

If Stata's developers tomorrow wrote a new command named "foobar", they would make two files available on their web site: `foobar.ado` (the ado-file code) and `foobar.sthlp` (the associated help file). Both are ordinary, readable ASCII text files. These files should be produced in a text editor, not a word processing program.

The importance of this program design goes far beyond the limits of official Stata. Since the adopath includes both Stata directories and other directories on your hard disk (or on a server's filesystem), you may acquire new Stata commands from a number of web sites. The *Stata Journal (SJ)*, a quarterly refereed journal, is the primary method for distributing user contributions. Between 1991 and 2001, the *Stata Technical Bulletin* played this role, and a complete set of issues of the *STB* are available on line at the Stata website.

The *SJ* is a subscription publication (articles more than three years old freely downloadable), but the `ado`- and `sthlp`-files may be freely downloaded from Stata's web site. The Stata `help` command accesses help on all installed commands; the Stata command `findit` will locate commands that have been documented in the STB and the SJ, and with one click you may install them in your version of Stata. Help for these commands will then be available in your own copy.

User extensibility: the SSC archive

But this is only the beginning. Stata users worldwide participate in the *StataList* listserv, and when a user has written and documented a new general-purpose command to extend Stata functionality, they announce it on the *StataList* listserv (to which you may freely subscribe: see Stata's web site).

Since September 1997, all items posted to `StataList` (over 1,300) have been placed in the Boston College Statistical Software Components (SSC) Archive in *RePEc* (Research Papers in Economics), available from IDEAS (`http://ideas.repec.org`) and EconPapers (`http://econpapers.repec.org`).

Any component in the SSC archive may be readily inspected with a web browser, using IDEAS' or EconPapers' search functions, and if desired you may install it with one command from the archive from within Stata. For instance, if you know there is a module in the archive named `mvsumm`, you could use `ssc describe mvsumm` to learn more about it, and `ssc install mvsumm` to install it if you wish. Anything in the archive can be accessed via Stata's `ssc` command: thus `ssc describe mvsumm` will locate this module, and make it possible to install it with one click.

Windows users should not attempt to download the materials from a web browser; it won't work.

Try it out: when you are connected to the Internet, type

```
ssc describe mvsumm
ssc install mvsumm
help mvsumm
```

The command `ssc new` lists, in the Stata Viewer, all SSC packages that have been added or modified in the last month. You may click on their names for full details. The command `ssc hot` reports on the most popular packages on the SSC Archive.

The Stata command `adoupdate` checks to see whether all packages you have downloaded and installed from the SSC archive, the *Stata Journal*, or other user-maintained `net from...` sites are up to date. `adoupdate` alone will provide a list of packages that have been updated. You may then use `adoupdate, update` to refresh your copies of those packages, or specify which packages are to be updated.

The importance of all this is that Stata is *infinitely extensible*. Any ado-file on your `adopath` is a full-fledged Stata command. Stata's capabilities thus extend far beyond the official, supported features described in the Stata manual to a vast array of additional tools.

Since the current directory is on the `adopath`, if you create an ado-file **hello.ado**:

```
program define hello
display "Stata says hello!"
end
exit
```

Stata will now respond to the command `hello`. It's that easy. *Try it out!*

Stata command syntax

Let us consider the form of Stata commands. One of Stata's great strengths, compared with many statistical packages, is that its command syntax follows strict rules: in grammatical terms, there are no irregular verbs. This implies that when you have learned the way a few key commands work, you will be able to use many more without extensive study of the manual or even on-line help.

The fundamental syntax of all Stata commands follows a *template*. Not all elements of the template are used by all commands, and some elements are only valid for certain commands. But where an element appears, it will appear in the same place, following the same grammar.

Like Unix or Linux, Stata is case sensitive. Commands must be given in lower case. For best results, keep all variable names in lower case to avoid confusion.

The general syntax of a Stata command is:

```
[prefix_cmd:] cmdname [varlist] [=exp]
                      [if exp] [in range]
                      [weight] [using...] [,options]
```

where elements in square brackets are optional for some commands.

Programmability of tasks

Stata may be used in an interactive mode, and those learning the package may wish to make use of the menu system. But when you execute a command from a pull-down menu, it records the command that you could have typed in the Review window, and thus you may learn that with experience you could type that command (or modify it and resubmit it) more quickly than by use of the menus.

Stata makes reproducibility very easy through a log facility, the ability to generate a command log (containing only the commands you have entered), and the do-file editor which allows you to easily enter, execute and save sequences of commands, or program fragments.

Going one step further, if you use the do-file editor to create a sequence of commands, you may save that do-file and reuse it tomorrow, or use it as the starting point for a similar set of data management or statistical operations. Working in this way promotes reproducibility, which makes it very easy to perform an alternate analysis of a particular model. Even if many steps have been taken since the basic model was specified, it is easy to go back and produce a variation on the analysis if all the work is represented by a series of programs.

One of the implications of the concern for reproducible work: avoid altering data in a non-auditable environment such as a spreadsheet. Rather, you should transfer external data into the Stata environment as early as possible in the process of analysis, and only make permanent changes to the data with do-files that can give you an audit trail of every change made to the data.

Programmable tasks are supported by *prefix commands*, as we will soon discuss, that provide implicit loops, as well as explicit looping constructs such as the `forvalues` and `foreach` commands.

To use these commands you must understand Stata's concepts of local and global *macros*. Note that the term macro in Stata bears no resemblance to the concept of an Excel macro. A macro, in Stata, is an alias to an object, which may be a number or string.

Local macros and scalars

In programming terms, *local macros* and *scalars* are the "variables" of Stata programs (not to be confused with the variables of the data set). The distinction: a local macro can contain a string, while a scalar can contain a single number (at maximum precision). You should use these constructs whenever possible to avoid creating variables with constant values merely for the storage of those constants. This is particularly important when working with large data sets.

When you want to work with a scalar object—such as a counter in a `foreach` or `forvalues` command—it will involve defining and accessing a local macro. As we will see, all Stata commands that compute results or estimates generate one or more objects to hold those items, which are saved as numeric scalars, local macros (strings or numbers) or numeric matrices.

The local macro

The *local macro* is an invaluable tool for do-file authors. A local macro is created with the `local` statement, which serves to name the macro and provide its content. When you next refer to the macro, you extract its value by *dereferencing* it, using the backtick (‘) and apostrophe (’) on its left and right:

```
local george 2
local paul = ‘george’ + 2
```

In this case, I use an equals sign in the second local statement as I want to *evaluate* the right-hand side, as an arithmetic expression, and store it in the macro `paul`. If I did not use the equals sign in this context, the macro `paul` would contain the string `2 + 2`.

forvalues and foreach

In other cases, you want to *redefine* the macro, not evaluate it, and you should not use an equals sign. You merely want to take the contents of the macro (a character string) and alter that string. The two key programming constructs for repetition, `forvalues` and `foreach`, make use of local macros as their "counter". For instance:

```
forvalues i=1/10 {
      summarize PRweek`i'
}
```

Note that the value of the local macro `i` is used within the body of the loop when that counter is to be referenced. Any Stata *numlist* may appear in the `forvalues` statement. Note also the curly braces, which must appear at the end of their lines.

In many cases, the `forvalues` command will allow you to substitute explicit statements with a single loop construct. By modifying the range and body of the loop, you can easily rewrite your do-file to handle a different case.

The `foreach` command is even more useful. It defines an iteration over any one of a number of lists:

- the contents of a varlist (list of existing variables)
- the contents of a newlist (list of new variables)
- the contents of a numlist (list of integers)
- the separate words of a macro
- the elements of an arbitrary list

For example, we might want to `summarize` each of these variables' detailed statistics from this World Bank data set:

```
sysuse lifeexp
foreach v of varlist popgrowth lexp gnppc {
      summarize `v', detail
}
```

Or, run a regression on variables for each region, and graph the data and fitted line:

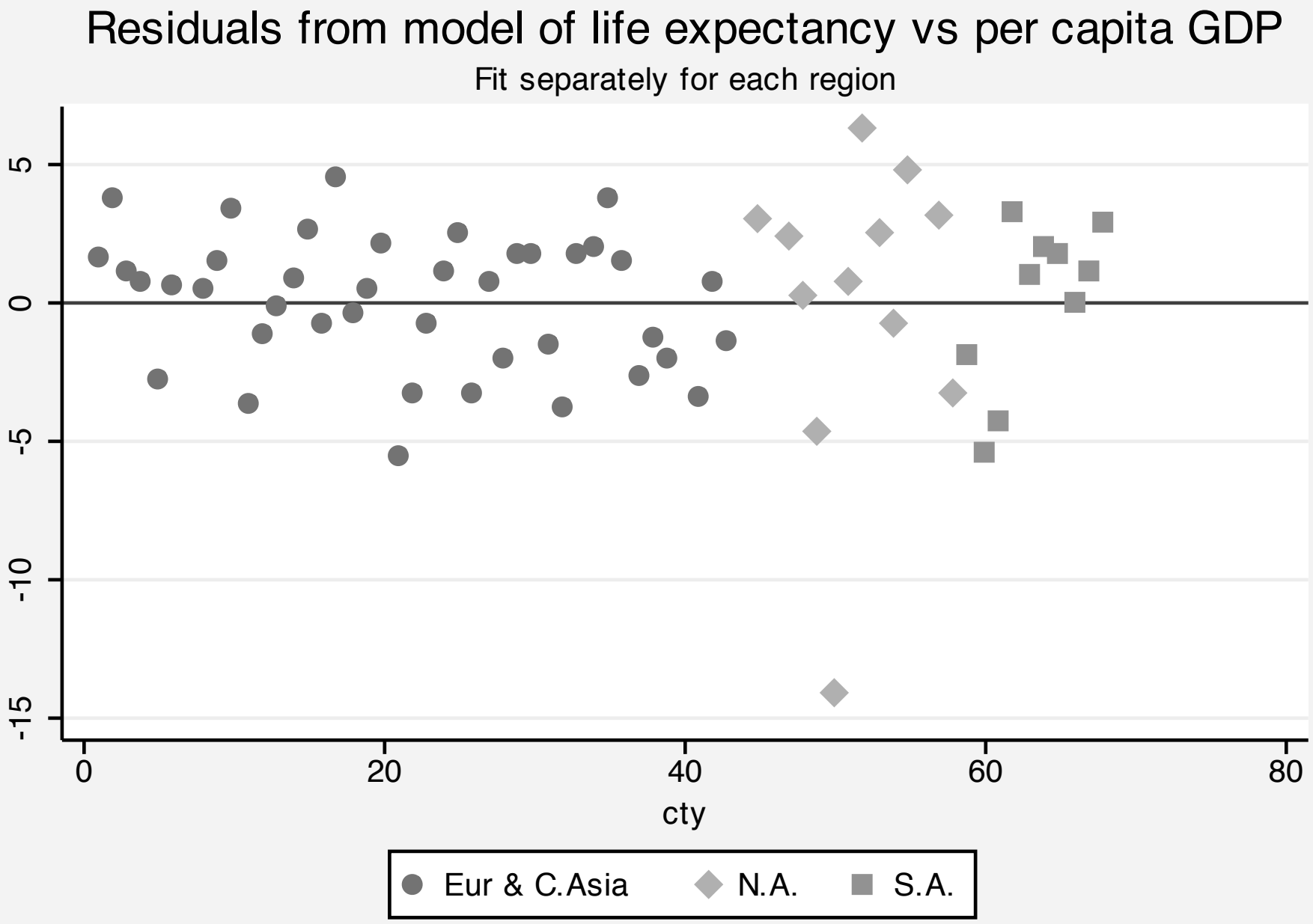
```
levelsof region, local(regid)
foreach c of local regid {
local rr : label region `c'
    regress lexp gnppc if region ==`c'
    twoway (scatter lexp  gnppc if region ==`c') ///
           (lfit lexp gnppc if region ==`c', ///
           ti(Region: `rr') name(fig`c', replace))
 }
```

A local macro can be built up by redefinition:

```
local alleps
foreach c of local regid {
regress lexp gnppc if region ==`c'
predict double eps`c' if e(sample), residual
local alleps "`alleps'  eps`c'"
}
```

Within the loop we redefine the macro `alleps` (as a double-quoted string) to contain itself and the name of the residuals from that region's regression. We could then use the macro `alleps` to generate a graph of all three regions' residuals:

```
gen cty = _n
scatter `alleps´ cty, yline(0) scheme(s2mono) legend(rows(1)) ///
  ti("Residuals from model of life expectancy vs per capita GDP") ///
  t2("Fit separately for each region")
```


Residuals from model of life expectancy vs per capita GDP
Fit separately for each region
5
0
-5
-10
-15
0
20
40
60
80
cty
Eur & C.Asia
N.A.
S.A.

Global macros

Stata also supports *global macros*, which are referenced by a different syntax (`$country` rather than `` `country' ``). Global macros are useful when particular definitions (e.g., the default working directory for a particular project) are to be referenced in several do-files that are to be executed. However, the creation of persistent objects of global scope can be dangerous, as global macro definitions are retained for the entire Stata session. One of the advantages of local macros is that they disappear when the do-file or ado-file in which they are defined finishes execution.

Prefix commands

A number of Stata commands can be used as *prefix commands*, preceding a Stata command and modifying its behavior. The most commonly employed is the *by prefix*, which repeats a command over a set of categories. The *statsby:* prefix repeats the command, but collects statistics from each category. The *rolling:* prefix runs the command on moving subsets of the data (usually time series).

Several other command prefixes: *simulate:*, which simulates a statistical model; *bootstrap:*, allowing the computation of bootstrap statistics from resampled data; and *jackknife:*, which runs a command over jackknife subsets of the data. The *svy:* prefix can be used with many statistical commands to allow for survey sample design.

Missing values

Missing value codes in Stata appear as the dot (.) in printed output (and a string missing value code as well: “”, the null string). It takes on the largest possible positive value, so in the presence of missing data you do not want to say

```
generate hiprice = (price > 10000)
```

but rather

```
generate hiprice = (price > 10000) if price <.
```

which then generates an indicator (dummy) variable equal to 1 for high-priced cars. The indicator will be zero for low-priced cars and missing for cars with missing prices.

Stata allows for multiple missing value codes (`.a, .b, .c, ..., .z`). The standard missing value code `(.)` is the smallest among them, so testing for $<$. will always work. You may also use the missing function: `mi(varname)` will return 1 if the observation is a missing value, 0 otherwise.

Missing data handling

An issue that often arises when importing data from external sources is the proper handling of missing data codes. Spreadsheet files often use `NA` to denote missing values, while in some datasets codes such as `-9`, `-999`, or `-0.001` are used. The latter instances are particularly worrisome as they may not be detected unless the variables' values are carefully scrutinized.

Note also that there is a missing value for string variables—the null, or zero-length string—which looks identical to a string of one or more space characters.

To properly handle missing values so that they are understood as such in Stata, use the `mvdecode` command. This command allows you to map various numeric values into numeric missing, or into one of the extended missing value codes `.a, .b, ..., .z`.

The `mvencode` command provides the inverse operation: particularly useful if you must transfer data to another package that uses some other convention for missing values.

No matter what methods you have used to input external data to the Stata workspace, you should immediately `save` the file in Stata format and perform the `describe` and `summarize` commands. It is much more efficient to read a Stata-format `.dta` file with `use` than to repeatedly input a text file with any of the commands discussed above. If the file is large, you may want to use the `compress` command to optimise Stata's memory usage before saving it. `compress` is non-destructive; it never reduces the stored precision of a variable.

Before any further use is made of this datafile, examine the results of the `describe` and `summarize` commands and ensure that each variable has been input properly, and that numeric variables have sensible values for their minima and maxima.

Display formats

Each variable may have its own default display format. This does not alter the contents of the variable, but only affects how it is displayed. For instance, `%9.2f` would display a two-decimal-place real number. The command

```
format varname %9.2f
```

will save that format as the default format of the variable, and

```
format date %tm
```

will format a Stata date variable into a monthly format (e.g., `1998m10`).

Variable labels

Each variable may have its own variable label. The variable label is a character string (maximum 80 characters) which describes the variable, associated with the variable via

```
label variable varname "text"
```

Variable labels, where defined, will be used to identify the variable in printed output, space permitting.

Value labels

Value labels associate numeric values with character strings. They exist separately from variables, so that the same mapping of numerics to their definitions can be defined once and applied to a set of variables (e.g. 1=very satisfied...5=not satisfied may be applied to all responses to questions about consumer satisfaction). Value labels are saved in the dataset. For example:

```
label define sexlbl 0 male 1 female
label values sex sexlbl
```

The latter command associates the label `sexlbl` with the variable `sex`. Unlike other packages, Stata's value labels are independent of variables, and the same label may be attached to any number of variables. If value labels are defined, they will be displayed in printed output instead of the numeric values.

The by prefix

You can often save time and effort by using the *by* prefix. When a command is prefixed with a *bylist*, it is performed repeatedly for each element of the variable or variables in that list, each of which must be categorical. You may *try it out*:

```
sysuse census
by region:  summ pop medage
```

will provide descriptive statistics for each of four US Census regions. If the data are not already sorted by the bylist variables, the prefix `bysort` should be used. The option `,total` will add the overall summary.

This can be extended to include more than one by-variable:

```
generate large = (pop > 5000000) & !mi(pop)
bysort region large: summ popurban death
```

This is a very handy tool, which often replaces explicit loops that must be used in other programs to achieve the same end.

The by-group logic will work properly even when some of the defined groups have no observations. However, its limitation is that it can only execute a single command for each category. If you want to estimate a regression for each group and save the residuals or predicted values, you must use an explicit loop.

The by *prefix* should not be confused with the by *option* available on some commands, which allows for specification of a grouping variable: for instance

```
ttest price, by(foreign)
```

will run a t-test for the difference of sample means across domestic and foreign cars.

Another useful aspect of *by* is the way in which it modifies the meanings of the observation number symbol. Usually _*n* refers to the current observation number, which varies from 1 to _*N*, the maximum defined observation. Under a bylist, _*n* refers to the observation within the bylist, and _*N* to the total number of observations for that category. This is often useful in creating new variables.

For instance, if you have individual data with a family identifier, these commands might be useful:

```
sort famid age
by famid: generate famsize = _N
by famid: generate birthorder = _N - _n +1
```

Here the `famsize` variable is set to _N, the total number of records for that family, while the `birthorder` variable is generated by sorting the family members' ages within each family.

Generating new variables

The command `generate` is used to produce new variables in the dataset, whereas `replace` must be used to revise an existing variable—and the command `replace` must always be spelled out.

A full set of functions are available for use in the `generate` command, including the standard mathematical functions, recode functions, string functions, date and time functions, and specialized functions (`help functions` for details). Note that `generate`'s `sum()` function is a running or cumulative sum.

As mentioned earlier, `generate` operates on all observations in the current data set, producing a result or a missing value for each. You need not write explicit loops over the observations. You can, but it is usually bad programming practice to do so. You may restrict `generate` or `replace` to operate on a subset of the observations with the `if` *exp* or `in` *range* qualifiers.

The `if` *exp* qualifier is usually more useful, but the `in` *range* qualifier may be used to list a few observations of the data to examine their validity. To list observations at the end of the current data set, use `if -5/`ℓ to see the last five.

You can take advantage of the fact that the *exp* specified in `generate` may be a logical condition rather than a numeric or string value. This allows producing both the 0s and 1s of an indicator (dummy, or Boolean) variable in one command. For instance:

```
generate large = (pop > 5000000) & !mi(pop)
```

The condition `& !mi(pop)` makes use of two logical operators: `&`, AND, and `!`, NOT to add the qualifier that the result variable should be missing if `pop` is missing, using the `mi()` function. Although numeric functions of missing values are usually missing, creation of an indicator variable requires this additional step for safety.

The third logical operator is the Boolean OR, written as `|`. Note also that a test for equality is specified with the == operator (as in C). The single = is used only for assignment.

Keep in mind the important difference between the `if` *exp* qualifier and the `if` (or 'programmer's if) command. Users of some alternative software may be tempted to use a construct such as

```
if (race == "Black") {
      raceid = 2
}
```

which is perfectly valid syntactically. It is also useless, in that it will define the entire `raceid` variable based on the value of `race` of the first observation in the data set! This is properly written in Stata as

```
generate raceid = 2 if race == "Black"
```

Functions for generate and replace

A number of lesser-known functions may be helpful in performing data transformations. For instance, the `inlist()` and `inrange()` functions return an indicator of whether each observation meets a certain condition: matching a value from a list or lying in a particular range.

```
generate byte newengland = ///
inlist(state, "CT", "ME", "MA", "NH", "RI", "VT")

generate byte middleage = inrange(age, 35, 49)
```

The generated variables will take a value of 1 if the condition is met and 0 if it is not. To guard against definition of missing values of `state` or `age`, add the clause `if !missing(`*varname*`)`:

```
generate byte middleage = inrange(age, 35, 49) if !mi(age)
```

Another common data manipulation task involves extracting a part of an integer variable. For instance, firms in the US are classified by four-digit Standard Industrial Classification (SIC) codes. The first two digits represent an industrial sector. To define an industry variable from the firm's SIC,

```
generate ind2d = int(SIC/100)
```

To extract the third and fourth digits, you could use

```
generate code34 = mod(SIC, 100)
```

using the modulo function to produce the remainder.

The `cond()` function may often be used to avoid more complicated coding. It evaluates its first argument, and returns the second argument if true, the third argument if false:

```
generate endqtr = cond( mod(month, 3) == 0, ///
"Filing month", "Non-filing month")
```

Notice that in this example the `endqtr` variable need not be defined as string in the `generate` statement.

Stata contains both a `recode` command and a `recode()` function. These facilities may be used in lieu of a number of `generate` and `replace` statements. There is also a `irecode` function to create a numeric code for values of a continuous variable falling in particular brackets. For example, using a dataset containing population and median age for a number of US states:

```
. generate size=irecode(pop, 1000, 4000, 8000, 20000)
. label define popsize 0 "<1m" 1 "1-4m" 2 "4-8m" 3 ">8m"
. label values size popsize
. tabstat pop, stat(mean min max) by(size)
Summary for variables: pop
     by categories of: size
```

size	mean	min	max
<1m	744.541	511.456	947.154
1-4m	2215.91	1124.66	3107.576
4-8m	5381.751	4075.97	7364.823
>8m	12181.64	9262.078	17558.07
Total	5142.903	511.456	17558.07

Rather than categorizing a continuous variable using threshold values, we may want to group observations based on *quantiles*: quartiles, quintiles, deciles, or any other percentiles of their empirical distribution. We can readily create groupings of that sort with `xtile`:

```
. xtile medagequart = medage, nq(4)
. tabstat medage, stat(n mean min max) by(medagequart)
Summary for variables: medage
     by categories of: medagequart (4 quantiles of medage)
```

medagequart	N	mean	min	max
1	7	29.02857	28.3	29.4
2	4	29.875	29.7	30
3	5	30.54	30.1	31.2
4	5	32	31.8	32.2
Total	21	30.25714	28.3	32.2

String-to-numeric conversion

A problem that commonly arises with data transferred from spreadsheets is the automatic classification of a variable as string rather than numeric. This often happens if the first value of such a variable is `NA`, denoting a missing value. If Stata's convention for numeric missings—the dot, or full stop (`.`) is used, this will not occur. If one or more variables are misclassified as string, how can they be modified?

First, a warning. Do not try to maintain long numeric codes (such as US Social Security numbers, with nine digits) in numeric form, as they will generally be rounded off. Treat them as string variables, which may contain up to 244 bytes.

If a variable has merely been misclassified as string, the brute-force approach can be used:

```
generate patid = real( patientid )
```

Any values of `patientid` that cannot be interpreted as numeric will be missing in `patid`. Note that this will also occur if numbers are stored with commas separating thousands.

A more subtle approach is given by the `destring` command, which can transform variables in place (with the `replace` option) and can be used with a *varlist* to apply the same transformation to a set of variables. Like the `real()` function, `destring` should only be used on variables misclassified as strings.

If the variable truly has string content and you need a numeric equivalent, for statistical analysis, you may use `encode` on the variable. To illustrate, let us read in some tab-delimited data with `insheet`.

```
. insheet using insheetdata.txt
(4 vars, 7 obs)

. format pop2008 %7.3f

. list, sep(0)
```

	state	abbrev	yearjo~d	pop2008
1.	Massachusetts	MA	1788	6.498
2.	New Hampshire	NH	1788	1.316
3.	Vermont	VT	1791	0.621
4.	New Jersey	NJ	1787	8.683
5.	Michigan	MI	1837	10.003
6.	Arizona	AZ	1912	6.500
7.	Alaska	AK	1959	0.686

As the data are tab-delimited, I can read a file with embedded spaces in the `state` variable.

I want to create a categorical variable identifying each state with an (arbitrary) numeric code. This can be achieved with `encode`:

```
. encode state, generate(stid)

. list state stid, sep(0)
```

	state	stid
1.	Massachusetts	Massachusetts
2.	New Hampshire	New Hampshire
3.	Vermont	Vermont
4.	New Jersey	New Jersey
5.	Michigan	Michigan
6.	Arizona	Arizona
7.	Alaska	Alaska

```
. summarize stid
```

Variable	Obs	Mean	Std. Dev.	Min	Max
stid	7	4	2.160247	1	7

Although `stid` is a numeric variable (as `summarize` shows) it is automatically assigned a *value label* consisting of the contents of `state`. The variable `stid` may now be used in analyses requiring numeric variables.

You may also want to make a variable into a string (for instance, to reinstate leading zeros in an id code variable). You may use the `string()` function, the `tostring` command or the `decode` command to perform this operation.

The egen command

Stata is not limited to using the set of defined `generate` functions. The `egen` (*e*xtended *gen*erate) command makes use of functions written in the Stata ado-file language, so that *_gzap.ado* would define the extended generate function `zap()`. This would then be invoked as

```
egen newvar = zap(oldvar)
```

which would do whatever `zap` does on the contents of `oldvar`, creating the new variable `newvar`.

A number of `egen` functions provide row-wise operations similar to those available in a spreadsheet: row sum, row average, row standard deviation, and so on. Users may write their own `egen` functions. In particular, `findit egenmore` for a very useful collection.

Although the syntax of an `egen` statement is very similar to that of `generate`, several differences should be noted. As only a subset of `egen` functions allow a `by` *varlist*`:` prefix or `by(`*varlist*`)` option, the documentation should be consulted to determine whether a particular function is *byable*, in Stata parlance. Similarly, the explicit use of `_n` and `_N`, often useful in `generate` and `replace` commands is not compatible with `egen`.

Wildcards may be used in row-wise functions. If you have state-level U.S. Census variables `pop1890, pop1900, ..., pop2000` you may use `egen nrCensus = rowmean(pop*)` to compute the average population of each state over those decennial censuses. The row-wise functions operate in the presence of missing values. The mean will be computed for all 50 states, although several were not part of the US in 1890.

The number of non-missing elements in the row-wise *varlist* may be computed with `rownonmiss()`, with `rowmiss()` as the complementary value. Other official row-wise functions include `rowmax()`, `rowmin()`, `rowtotal()` and `rowsd()` (row standard deviation). The functions `rowfirst()` and `rowlast()` give the first (last) non-missing values in the *varlist*. You may find this useful if the variables refer to sequential items: for instance, wages earned per year over several years, with missing values when unemployed. `rowfirst()` would return the earliest wage observation, and `rowlast()` the most recent.

Official `egen` also provides a number of statistical functions which compute a statistic for specified observations of a variable and place that constant value in each observation of the new variable. Since these functions generally allow the use of `by` *varlist*`:`, they may be used to compute statistics for each *by-group* of the data. This facilitates computing statistics for each household for individual-level data or each industry for firm-level data. The `count()`, `mean()`, `min()`, `max()` and `total()` functions are especially useful in this context.

As an illustration using our state-level data, we `egen` the average population in each of the `size` groups defined above, and express each state's population as a percentage of the average population in that size group. Size category 0 includes the smallest states in our sample.

```
. bysort size: egen avgpop = mean(pop)
. generate popratio = 100 * pop / avgpop
. format popratio %7.2f
. list state pop avgpop popratio if size == 0, sep(0)
```

	state	pop	avgpop	popratio
1.	Rhode Island	947.2	744.541	127.21
2.	Vermont	511.5	744.541	68.69
3.	N. Dakota	652.7	744.541	87.67
4.	S. Dakota	690.8	744.541	92.78
5.	New Hampshire	920.6	744.541	123.65

Other `egen` functions in this statistical category include `iqr()` (inter-quartile range), `kurt()` (kurtosis), `mad()` (median absolute deviation), `mdev()` (mean absolute deviation), `median()`, `mode()`, `pc()` (percent or proportion of total), `pctile()`, `p(`*n*`)` (n^{th} percentile), `rank()`, `sd()` (standard deviation), `skew()` (skewness) and `std()` (*z*-score).

Many other `egen` functions are available; see `help egen` for details.

Time series calendar

Stata supports date (and time) variables and the creation of a time series calendar variable. Dates are expressed, as they are in Excel, as the number of days from a base date. In Stata's case, that date is 1 Jan 1960 (like Unix/Linux). You may set up data on an annual, half-yearly, quarterly, monthly, weekly or daily calendar, as well as a calendar that merely uses the observation number.

You may also set the `delta` of the calendar variable to be other than 1: for instance, if you have data at five-year intervals, you may define the data as annual with `delta=5`. This ensures that the lagged value of the 2005 observation is that of 2000.

An observation-number calendar is generally necessary for business-daily data where you want to avoid gaps for weekends, holidays etc. which will cause lagged values and differences to contain missing values. However, you may want to create two calendar variables for the same time series data: one for statistical purposes and one for graphical purposes, which will allow the series to be graphed with calendar-date labels. This procedure is illustrated in "Stata Tip 40: Taking care of business...", Baum, CF. *Stata Journal*, 2007, 7:1, 137-139, included in your materials.

A useful utility for setting the appropriate time series calendar is `tsmktim`, available from the SSC Archive (`ssc describe tsmktim` and described in "Utility for time series data", Baum, CF and Wiggins, VL. *Stata Technical Bulletin*, 2000, 57, 2-4. It will set the calendar, issuing the appropriate `tsset` command and the display format of the resulting calendar variable, and can be used in a panel data context where each time series starts in the same calendar period.

Time series operators

The `D.`, `L.`, and `F.` operators may be used under a time series calendar (including in the context of panel data) to specify first differences, lags, and leads, respectively. These operators understand missing data, and numlists: e.g. `L(1/4).x` is the first through fourth lags of `x`, while `L2D.x` is the second lag of the first difference of the `x` variable.

It is important to use the time series operators to refer to lagged or led values, rather than referring to the observation number (e.g., `_n-1`). The time series operators respect the time series calendar, and will not mistakenly compute a lag or difference from a prior period if it is missing. This is particularly important when working with panel data to ensure that references to one individual do not reach back into the prior individual's data.

Using time series operators, you may not only consistently generate differences, lags, and leads, but may refer to them 'on the fly' in statistical and estimation commands. That is, to estimate an AR(4) model, you need not create the lagged variables:

```
regress y L(1/4).y
```

or, to test Granger causality,

```
regress y (-4/4).x
```

which would regress y_t on four leads, four lags and the current value of x_t.

For a "Dickey–Fuller" style regression,

```
regress D.y L.y
```

Factor variables

A valuable new feature in Stata version 11 is the *factor variable*. Stata has only one kind of numeric variable (although it supports several different data types, which define the amount of storage needed and possible range of values). However, if a variable is *categorical*, taking on non-negative integer values, it may be used as a factor variable with the `i.` prefix.

The use of factor variables not only avoids explicit generation of indicator (dummy) variables for each level of the categorical variable, but it means that the needed indicator variables are generated 'on the fly', as needed. Thus, to include the variable `region`, a categorical variable in `census.dta` which takes on values 1–4, we need only refer to `i.region` in an estimation command.

This in itself merely mimics a preexisting feature of Stata: the `xi:` prefix. But factor variables are much more powerful, in that they can be used to define interactions, both with other factor variables and with continuous variables. Traditionally, you would define interactions by creating new variables representing the product of two indicators, or the product of an indicator with a continuous variable.

There is a great advantage in using factor variables rather than creating new interaction variables. if you define interactions with the factor variable syntax, Stata can then interpret the expression in postestimation commands such as `margins`. For instance, you can say `i.race#i.sex`, or `i.sex#c.bmi`, or `c.bmi#c.bmi`, where `c.` denotes a continuous variable, and `#` specifies an interaction.

With interactions between indicator and continuous variables specified in this syntax, we can evaluate the total effect of a change without further programming. For instance,

```
regress healthscore i.sex#c.bmi c.bmi#c.bmi
margins, dydx(bmi) at(sex = (0 1))
```

which will perform the calculation of $\partial healthscore/\partial bmi$ for each level of categorical variable `sex`, taking into account the squared term in `bmi`. We will discuss `margins` more fully in later talks in this series.

File handling

File extensions usually employed (but not required) include:

```
.ado    automatic do-file (defines a Stata command)
.dct    data dictionary, optionally used with infile
.do     do-file (user program)
.dta    Stata binary dataset
.gph    graphics output file (binary)
.log    text log file
.smcl   SMCL (markup) log file, for use with Viewer
.raw    ASCII data file
.sthlp  Stata help file
```

These extensions need not be given (except for `.ado`). If you use other extensions, they must be explicitly specified.

Reading external data with insheet

Comma-separated (CSV) files or tab-delimited data files may be read very easily with the `insheet` command—which despite its name does not read spreadsheet files. If your file has variable names in the first row that are valid for Stata, they will be automatically used (rather than default variable names). You usually need not specify whether the data are tab- or comma-delimited. Note that `insheet` cannot read space-delimited data (or character strings with embedded spaces, unless they are quoted).

If the file extension is `.raw`, you may just use

```
insheet using filename
```

to read it. If other file extensions are used, they must be given:

```
insheet using filename.csv
insheet using filename.txt
```

Reading external data with infile

A free-format ASCII text file with space-, tab-, or comma-delimited data may be read with the `infile` command. The missing-data indicator (.) may be used to specify that values are missing.
The command must specify the variable names. Assuming `auto.raw` contains numeric data,

```
infile price mpg displacement using auto
```

will read it. If a file contains a combination of string and numeric values in a variable, it should be read as string, and `encode` used to convert it to numeric with string value labels.

If some of the data are string variables without embedded spaces, they must be specified in the command:

```
infile str3 country price mpg displacement using auto2
```

would read a three-letter country of origin code, followed by the numeric variables. The number of observations will be determined from the available data.

The `infile` command may also be used with fixed-format data, including data containing undelimited string variables, by creating a dictionary file which describes the format of each variable and specifies where the data are to be found. The dictionary may also specify that more than one record in the input file corresponds to a single observation in the data set.

Sometimes data fields are not delimited—for instance, the sequence '102' might actually represent three integer variables. A `dictionary` must then be used to define the variables' locations.

The `byvariable()` option allows a variable-wise dataset to be read, where one specifies the number of observations available for each series.

Reading external data with infix

An alternative to infile with a dictionary is the `infix` command, which presents a syntax similar to that used by SAS for the definition of variables' data types and locations in a fixed-format ASCII data set: that is, a data file in which certain columns contain certain variables. The `_column()` directive allow contents of a fixed-format data file to be retrieved selectively.

`infix` may also be used for more complex record layouts where one individual's data are contained on several records in an ASCII file.

A logical condition may be used on the `infile` or `infix` commands to read only those records for which certain conditions are satisfied: i.e.

```
infix using employee if sex=="M"
infile price mpg using auto in 1/20
```

where the latter will read only the first 20 observations from the external file. This might be very useful when reading a large data set, where one can check to see that the formats are being properly specified on a subset of the file.

Reading external data with Stat/Transfer

If your data are already in the internal format of SAS, SPSS, Excel, GAUSS, MATLAB, or a number of other packages, the best way to get it into Stata is by using the third-party product Stat/Transfer.

Stat/Transfer will preserve variable labels, value labels, and other aspects of the data, and can be used to convert a Stata binary file into other packages' formats. It can also produce subsets of the data (selecting variables, cases or both) so as to generate an extract file that is more manageable. This is particularly important when the 2,047-variable limit on standard Stata data sets is encountered. Stat/Transfer is well documented, with on-line help available in both Windows, Mac OS X and Unix versions, and an extensive manual. It is remarketed by StataCorp.

Writing external data: outfile, outsheet and file

If you want to transfer data to another package, Stat/Transfer is very useful. But if you just want to create an ASCII file from Stata, the `outfile` command may be used. It takes a *varlist*, and the `if` or `in` clauses may be used to control the observations to be exported. Applying `sort` prior to outfile will control the order of observations in the external file. You may specify that the data are to be written in comma-separated format.

The `outsheet` command can write a comma-delimited or tab-delimited ASCII file, optionally placing the variable names in the first row. Such a file can be easily read by a spreadsheet program such as Excel. Note that `outsheet` does *not* write spreadsheet files.

For customized output, the `file` command can write out information (including scalars, matrices and macros, text strings, etc.) in any ASCII or binary format of your choosing.

Writing external data: postfile and post

A very useful capability is provided by the `postfile` and `post` commands, which permit a Stata data set to be created in the course of a program. For instance, you may be simulating the distribution of a statistic, fitting a model over separate samples, or bootstrapping standard errors. Within the looping structure, you may `post` certain numeric values to the `postfile`. This will create a separate Stata binary data set, which may then be opened in a later Stata run and analysed. Note, however, that only numeric expressions may be written to the `postfile`, and the parens () given in the documentation, surrounding each *exp*, are required.

Combining data sets

In many empirical research projects, the raw data to be utilized are stored in a number of separate files: separate "waves" of panel data, timeseries data extracted from different databases, and the like. Stata only permits a single data set to be accessed at one time. How, then, do you work with multiple data sets? Several commands are available, including `append`, `merge`, and `joinby`.

How, then, do you combine datasets in Stata? First of all, it is important to understand that at least one of the datasets to be combined must already have been saved in Stata format. Second, you should realize that each of Stata's commands for combining datasets provides a certain functionality, which should not confused with that of other commands.

The append command

The `append` command combines two Stata-format data sets that possess variables in common, adding observations to the existing variables. The same variables need not be present in both files, as long as a subset of the variables are common to the “master” and “using” data sets. It is important to note that “`PRICE`" and “`price`” are different variables, and one will not be appended to the other.

You might have a dataset on the demographic characteristics in 2007 of the largest municipalities in China, `cityCN`. If you were given a second dataset containing the same variables for the largest municipalities in Japan in 2007, `cityJP`, you might want to combine those datasets with `append`. With the `cityCN` dataset in memory, you would `append using cityJP`, which would add those records as additional observations. You could then save the combined file under a different name. `append` can be used to combine multiple datasets, so if you had the additional files `cityPH` and `cityMY`, you could list those filenames in the `using` clause as well.

Prior to using `append`, it is a good idea to create an identifier variable in each dataset that takes on a constant value: e.g., `gen country = 1` in the CN dataset, `gen country = 2` in the JP dataset, etc.

For instance, consider the `append` command with two stylized datasets:

$$\text{dataset1}: \begin{pmatrix} id & var1 & var2 \\ 112 & \vdots & \vdots \\ 216 & \vdots & \vdots \\ 449 & \vdots & \vdots \end{pmatrix}$$

$$\text{dataset2}: \begin{pmatrix} id & var1 & var2 \\ 126 & \vdots & \vdots \\ 309 & \vdots & \vdots \\ 421 & \vdots & \vdots \\ 604 & \vdots & \vdots \end{pmatrix}$$

These two datasets contain the same variables, as they must for `append` to sensibly combine them. If `dataset2` contained `idcode, Var1, Var2` the two datasets could not sensibly be appended without renaming the variables (recall that in Stata, `var1` and `Var1` are two separate variables). Appending these two datasets with common variable names creates a single dataset containing all of the observations:

$$\text{combined} : \begin{pmatrix} id & var1 & var2 \\ 112 & \vdots & \vdots \\ 216 & \vdots & \vdots \\ 449 & \vdots & \vdots \\ 126 & \vdots & \vdots \\ 309 & \vdots & \vdots \\ 421 & \vdots & \vdots \\ 604 & \vdots & \vdots \end{pmatrix}$$

The rule for `append`, then, is that if datasets are to be combined, they should share the same variable names and datatypes (string vs. numeric). In the above example, if `var1` in `dataset1` was a `float` while that variable in `dataset2` was a `string` variable, they could not be appended.

It is permissible to append two datasets with differing variable names in the sense that `dataset2` could also contain an additional variable or variables (for example, `var3, var4`). The values of those variables in the observations coming from `dataset1` would then be set to missing.

Some care must be taken when appending datasets in which the same variable may exist with different data types (string in one, numeric in another). For details, see "Stata tip 73: append with care!", Baum CF, *Stata Journal*, 2008, 9:1, 166-168, included in your materials.

The merge command

We now describe the `merge` command, which is Stata's basic tool for working with more than one dataset. Its syntax has changed considerably in Stata version 11.

The merge command takes a first argument indicating whether you are performing a *one-to-one*, *many-to-one*, *one-to-many* or *many-to-many* merge using specified key variables. It can also perform a one-to-one merge by observation.

Like the `append` command, the `merge` works on a "master" dataset—the current contents of memory—and a single "using" dataset (prior to Stata 11, you could specify multiple using datasets). One or more key variables are specified, and in Stata 11 you need not sort either dataset prior to merging.

The distinction between "master" and "using" is important. When the same variable is present in each of the files, Stata's default behavior is to hold the master data inviolate and discard the using dataset's copy of that variable. This may be modified by the `update` option, which specifies that non-missing values in the using dataset should replace missing values in the master, and the even stronger `update replace`, which specifies that non-missing values in the using dataset should take precedence.

A "*one-to-one*" merge (written `merge 1:1`) specifies that each record in the using data set is to be combined with one record in the master data set. This would be appropriate if you acquired additional variables for the same observations.

In any use of `merge`, a new variable, `_merge`, takes on integer values indicating whether an observation appears in the master only, the using only, or appears in both. This may be used to determine whether the merge has been successful, or to remove those observations which remain unmatched (e.g. merging a set of households from different cities with a comprehensive list of postal codes; one would then discard all the unused postal code records). The `_merge` variable must be dropped before another `merge` is performed on this data set.

Consider these two stylized datasets:

$$\text{dataset1}: \begin{pmatrix} id & var1 & var2 \\ 112 & \vdots & \vdots \\ 216 & \vdots & \vdots \\ 449 & \vdots & \vdots \end{pmatrix}$$

$$\text{dataset3}: \begin{pmatrix} id & var22 & var44 & var46 \\ 112 & \vdots & \vdots & \vdots \\ 216 & \vdots & \vdots & \vdots \\ 449 & \vdots & \vdots & \vdots \end{pmatrix}$$

We may `merge` these datasets on the common *merge key*: in this case, the `id` variable:

$$\text{combined}: \begin{pmatrix} id & var1 & var2 & var22 & var44 & var46 \\ 112 & \vdots & \vdots & \vdots & \vdots & \vdots \\ 216 & \vdots & \vdots & \vdots & \vdots & \vdots \\ 449 & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

The rule for `merge`, then, is that if datasets are to be combined on one or more *merge keys*, they each must have one or more variables with a common name and datatype (string vs. numeric). In the example above, each dataset must have a variable named `id`. That variable can be numeric or string, but that characteristic of the merge key variables must match across the datasets to be merged. Of course, we need not have exactly the same observations in each dataset: if `dataset3` contained observations with additional `id` values, those observations would be merged with missing values for `var1` and `var2`.

This is the simplest kind of merge: the *one-to-one merge*. Stata supports several other types of merges. But the key concept should be clear: the `merge` command combines datasets "horizontally", adding variables' values to existing observations.

The `merge` command can also do a "many-to-one"' or "one-to-many" merge. For instance, you might have a dataset named `hospitals` and a dataset named `discharges`, both of which contain a hospital ID variable `hospid`. If you had the `hospitals` dataset in memory, you could `merge 1:m hospid using discharges` to match each hospital with its prior patients. If you had the `discharges` dataset in memory, you could `merge m:1 hospid using hospitals` to add the hospital characteristics to each discharge record. This is a very useful technique to combine aggregate data with disaggregate data without dealing with the details.

Although "many-to-one"' or "one-to-many" merges are commonplace and very useful, you should rarely want to do a "many-to-many" (`m:m`) merge, which will yield seemingly random results.

The long-form dataset is very useful if you want to add aggregate-level information to individual records. For instance, we may have panel data for a number of companies for several years. We may want to attach various macro indicators (interest rate, GDP growth rate, etc.) that vary by year but not by company. We would place those macro variables into a dataset, indexed by year, and sort it by year.

We could then `use` the firm-level panel dataset and sort it by `year`. A `merge` command can then add the appropriate macro variables to each instance of `year`. This use of `merge` is known as a *one-to-many* match merge, where the `year` variable is the *merge key*.

Note that the merge key may contain several variables: we might have information specific to industry and year that should be merged onto each firm's observations.

Reconfiguring data sets

Data are often provided in a different orientation than that required for statistical analysis. The most common example of this occurs with panel, or longitudinal, data, in which each observation conceptually has both cross-section *(i)* and time-series *(t)* subscripts. Often one will want to work with a "pure" cross-section or "pure" time-series. If the microdata themselves are the objects of analysis, this can be handled with sorting and a loop structure. If you have data for N firms for T periods per firm, and want to fit the same model to each firm, one could use the `statsby` command, or if more complex processing of each model's results was required, a `foreach` block could be used. If analysis of a cross-section was desired, a `bysort` would do the job.

But what if you want to use average values for each time period, averaged over firms? The resulting dataset of T observations can be easily created by the `collapse` command, which permits you to generate a new data set comprised of summary statistics of specified variables. More than one summary statistic can be generated per input variable, so that both the number of firms per period and the average return on assets could be generated. `collapse` can produce counts, means, medians, percentiles, extrema, and standard deviations.

Different models applied to longitudinal data require different orientations of those data. For instance, seemingly unrelated regressions (`sureg`) require the data to have T observations ("wide"), with separate variables for each cross–sectional unit. Fixed–effects or random-effects regression models `xtreg`, on the other hand, require that the data be stacked or "vec"'d in the "long" format. It is usually much easier to generate transformations of the data in stacked format, where a single variable is involved.

The `reshape` command allows you to transfer the data from the former ("wide") format to the latter ("long") format or vice versa. It is a complicated command, because of the many variations on this process one might encounter, but it is very powerful.

When data have more than one identifier per record, they may be organized in different ways. For instance, it is common to find on-line displays or downloadable spreadsheets of data for individual units—for instance, U.S. states—with the unit's name labeling the row and the year labeling the column. If these data were brought into Stata in this form, they would be in the *wide form*, wide form with the same measurement (population) for different years denoted as separate Stata variables:

```
. list, noobs
```

state	pop1990	pop1995	pop2000
CT	3291967	3324144	3411750
MA	6022639	6141445	6362076
RI	1005995	1017002	1050664

There are a number of Stata commands—such as `egen` row-wise functions—which work effectively on data stored in the wide form. It may also be a useful form of data organization for producing graphs.

Alternatively, we can imagine stacking each year's population figures from this display into one variable, `pop`. In this format, known in Stata as the *long form*, each datum is identified by two variables: the state name and the year to which it pertains.

We use `reshape` to transform the data, indicating that `state` should be the main row identifier (`i`) with `year` as the secondary identifier (`j`):

```
. reshape long pop, i(state) j(year)

. list, noobs sepby(state)
```

state	year	pop
CT	1990	3291967
CT	1995	3324144
CT	2000	3411750
MA	1990	6022639
MA	1995	6141445
MA	2000	6362076
RI	1990	1005995
RI	1995	1017002
RI	2000	1050664

This data structure is required for many of Stata's statistical commands, such as the `xt` suite of panel data commands. The long form is also very useful for data management using `by`-groups and the computation of statistics at the individual level, often implemented with the `collapse` command.

Inevitably, you will acquire data (either raw data or Stata datasets) that are stored in either the wide or the long form and will find that translation to the other format is necessary to carry out your analysis. In statistical packages lacking a data-reshape feature, common practice entails writing the data to one or more external text files and reading it back in.

With the proper use of `reshape`, writing data out and reading them back in is not necessary in Stata. But `reshape` requires, first of all, that the data to be reshaped are labelled in such a way that they can be handled by the mechanical rules that the command applies. In situations beyond the simple application of `reshape`, it may require some experimentation to construct the appropriate command syntax. This is all the more reason for enshrining that code in a do-file as some day you are likely to come upon a similar application for `reshape`.

An illustration of advanced use of `reshape` on data from *International Financial Statistics* is provided in Baum CF, Cox NJ, "Stata tip 45: Getting those data into shape," *Stata Journal*, 2007, 7, 268–271, included in your materials.