Using Stata for data management and reproducible research

Christopher F Baum

Boston College and DIW Berlin

Birmingham Business School, March 2013

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 12.1).

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

An 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:

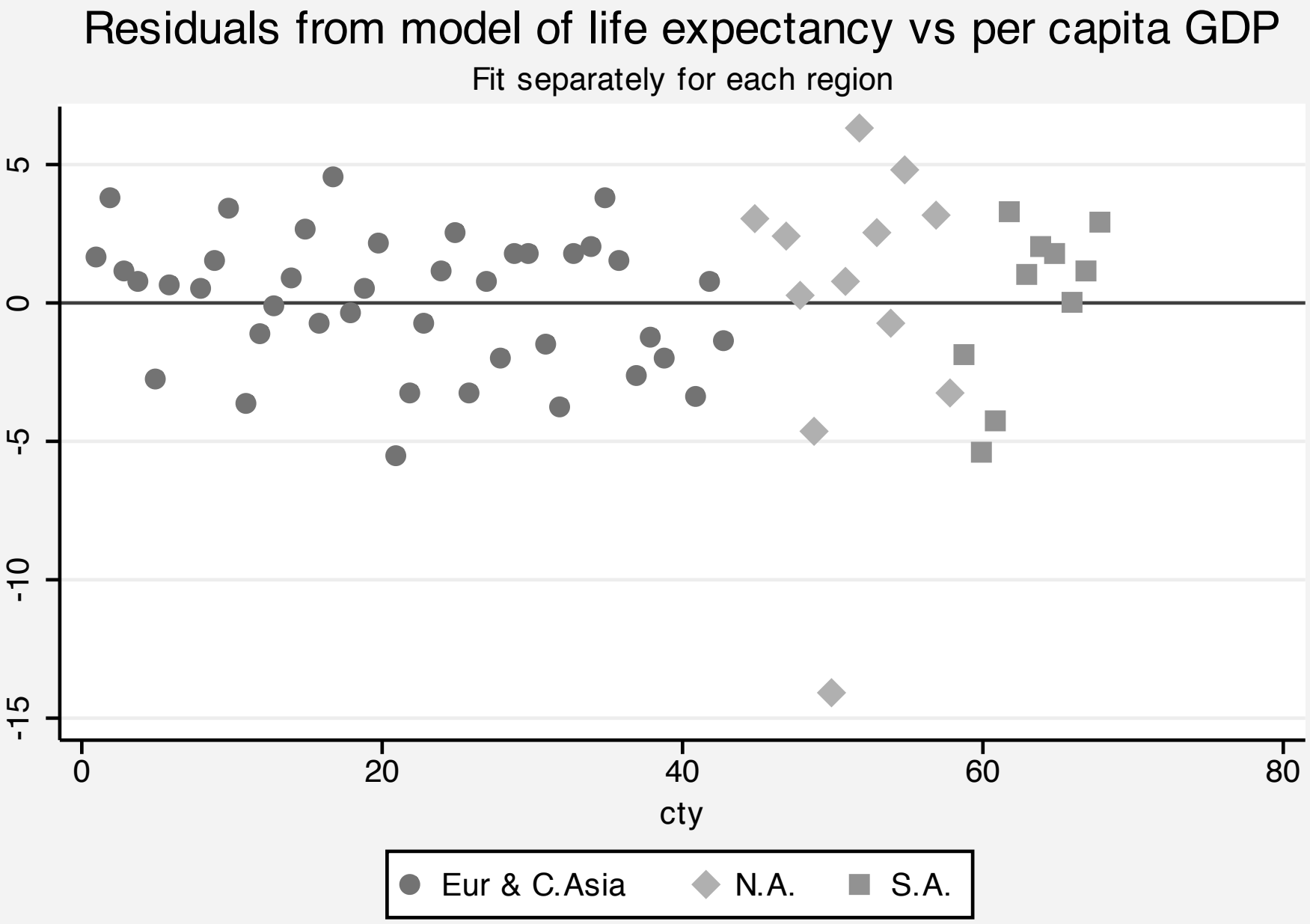
```
levelsof region, local(regid)
foreach c of local regid {
local rr : label region `c'
    regress lexp gnppc if region ==`c'
    twoway (scatter lexp  gnppc if region ==`c') ///
           (lfit lexp gnppc if region ==`c', ///
           ti(Region: `rr') name(fig`c', replace))
 }
```

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.

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.

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 L(-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 and 12 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.

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 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 or 12 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.

Storing and retrieving estimates

The `estimates` suite of commands allow you to store the results of a particular estimation for later use in a Stata session. For instance, after the commands

```
sysuse auto
regress price mpg length turn
estimates store model1
regress price weight length displacement
estimates store model2
regress price weight length gear_ratio foreign
estimates store model3
```

the command

```
estimates table model1 model2 model3
```

will produce a nicely-formatted table of results. Options on `estimates table` allow you to control precision, whether standard errors or t-values are given, significance stars, summary statistics, etc.

For example:

```
estimates table model1 model2 model3, b(%10.3f) ///
se(%7.2f) stats(r2 rmse N) ///
title(Some models of auto price)
```

```
. estimates table model1 model2 model3, b(%10.3f) se(%7.2f) stats(r2 rmse N) title(Some models of
```

Some models of auto price

Variable	model1	model2	model3
mpg	**-186.667**		
	87.54		
length	**52.583**	**-97.634**	**-88.027**
	31.42	**39.57**	**33.27**
turn	**-198.981**		
	138.59		
weight		**4.613**	**5.479**
		1.40	**1.05**
displacement		**0.727**	
		6.97	
gear_ratio			**-669.054**
			926.68
foreign			**3837.913**
			738.98
_cons	**8148.017**	**10440.629**	**7041.466**
	6057.16	**4369.32**	**4838.73**
r2	**0.251**	**0.348**	**0.552**
rmse	**2606.576**	**2432.740**	**2030.035**
N	**74**	**74**	**74**

legend: b/se

Publication-quality tables

Although `estimates table` can produce a summary table quite useful for evaluating a number of specifications, we often want to produce a publication-quality table for inclusion in a word processing document. Ben Jann's `estout` command suite processes stored `estimates` and provides a great deal of flexibility in generating such a table.

Programs in the `estout` suite can produce tab-delimited tables for MS Word, HTML tables for the web, and—my favorite—LaTeX tables for professional papers. In the LaTeX output format, `estout` can generate Greek letters, sub- and superscripts, and the like. `estout` is available from SSC, with extensive on-line help, and was described in the *Stata Journal*, 5(3), 2005 and 7(2), 2007. It has its own website at `http://repec.org/bocode/e/estout`.

From the example above, rather than using `estimates save` and `estimates table` we use Jann's `eststo` (store) and `esttab` (table) commands:

```
eststo clear
eststo: reg price mpg length turn
eststo: reg price weight length displacement
eststo: reg price weight length gear_ratio foreign
esttab using auto1.tex, stats(r2 bic N)  ///
subst(r2 \$R^2$) title(Models of auto price) ///
replace
```

Table 1: Models of auto price

	(1) price	(2) price	(3) price
mpg	-186.7*		
	(-2.13)		
length	52.58	-97.63*	-88.03*
	(1.67)	(-2.47)	(-2.65)
turn	-199.0		
	(-1.44)		
weight		4.613**	5.479***
		(3.30)	(5.24)
displacement		0.727	
		(0.10)	
gear_ratio			-669.1
			(-0.72)
foreign			3837.9***
			(5.19)
_cons	8148.0	10440.6*	7041.5
	(1.35)	(2.39)	(1.46)
R^2	0.251	0.348	0.552
bic	1387.2	1377.0	1353.5
N	74	74	74

t statistics in parentheses

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

Producing sets of results

We illustrate how the computation of statistics on a per-firm basis can be automated using daily CRSP stock returns data for 2001–2006. In this example, we create a subset of the full dataset, merge it with the NYSE value-weighted market return index, and restrict it to include only those firms with full coverage over the period.

DurhamP1a.do

```
// DurhamP1a  cfb A306
set more off
capt log close
cd "/Users/baum/Documents/Stata/StataWorkshops"
log using DurhamP1a.smcl, replace
// clear all
// set mem 600m
use crspret200106, clear
format caldt %td
// create subset with ~15% of original dataset
keep if inrange(permno, 28118, 75039)
merge n:1 caldt using nyse_vwretd
drop _merge
sort permno caldt
// keep only firms with full history, non-missing ret
drop if mi(ret)
// compute number of days per firm
by permno: g nfirm = _N
// drop those with less than full coverage
su nfirm, meanonly
drop if nfirm < r(max)
// create sequential time trend variable
by permno: g t = _n
save crsp_nyse200106, replace
log close
```

In the second step, we create a year variable from the daily calendar, and use the `statsby` prefix command to collect the coefficients and standard errors from a CAPM regression of each firm's return on the market return, year by year, and save those results in a new Stata dataset.

DurhamP1b.do

```
// DurhamP1b  cfb A306
set more off
capt log close
cd "/Users/baum/Documents/Stata/StataWorkshops"
log using DurhamP1b.smcl, replace
use crsp_nyse200106, clear
// set up with sequential time series calendar
tsset permno t
// count number of firms
egen ftag = tag(permno)
count if ftag
// create year variable
g year = yofd(caldt)
// loop over firms and years, saving coefficients and s.e.s
// in a new dataset
statsby _b _se, by(permno year) saving(capmbetas, replace)
    nodots: reg ret vwretd
log close
```

We may now compute summary statistics from the estimated α and β coefficients of the CAPM for each year. By applying the `collapse` command to the dataset, we can produce figures illustrating how the mean estimates have changed over the period, in point and interval form.

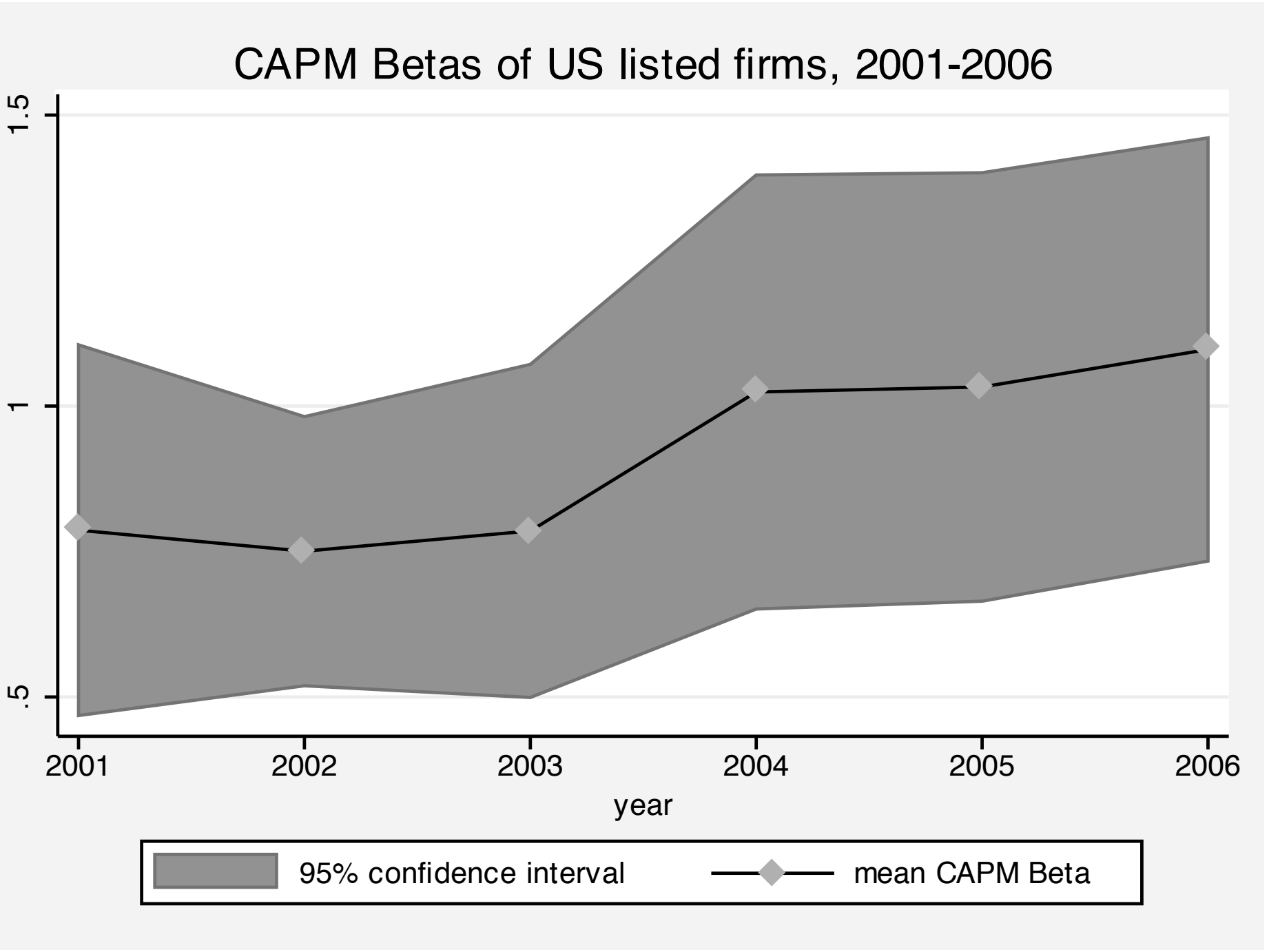
```
. tabstat _b_vwretd, by(year) stat(N mean min p50 max) col(stat)
Summary for variables: _b_vwretd
     by categories of: year
```

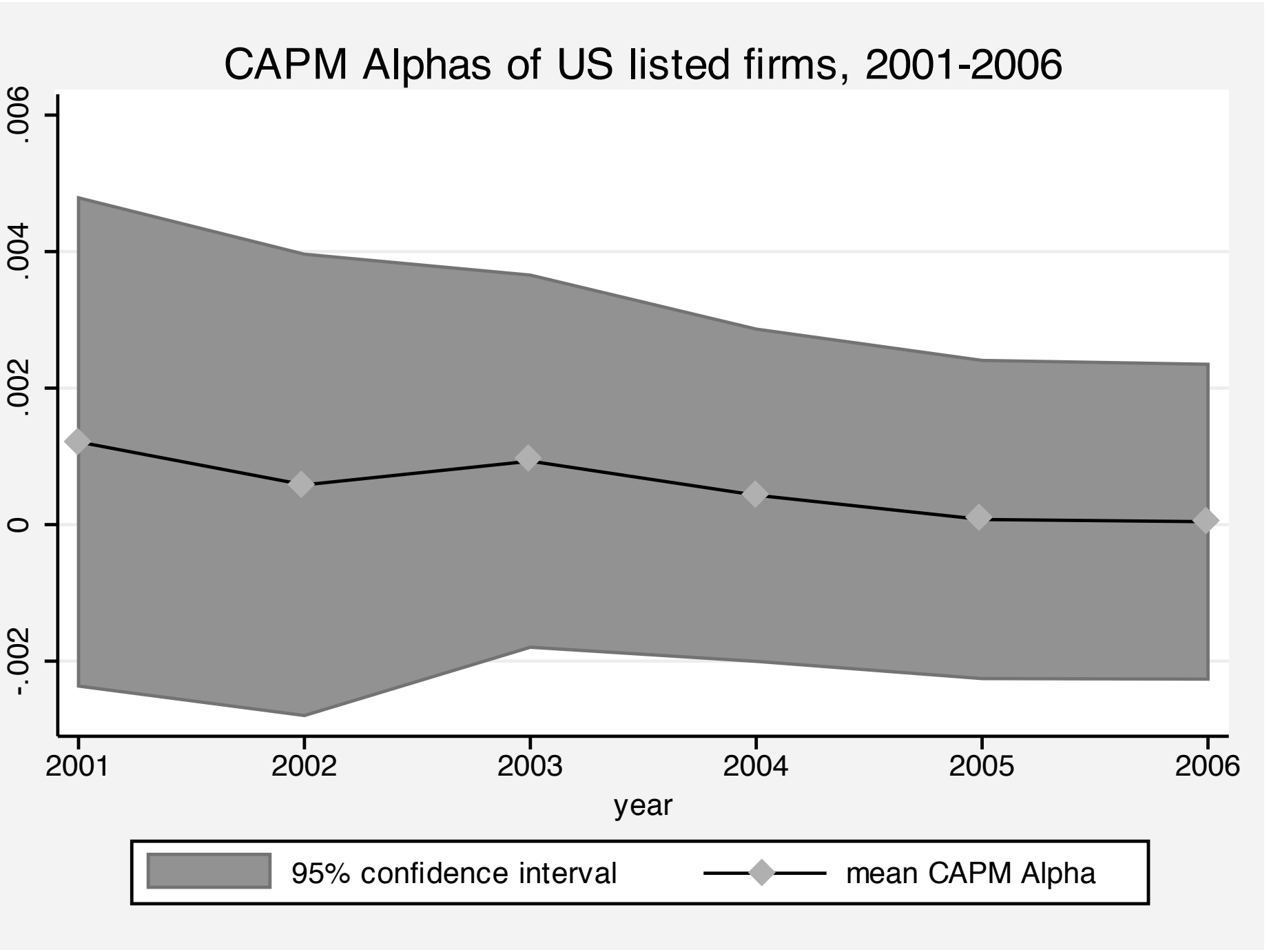
year	N	mean	min	p50	max
2001	827	.7866741	-1.373731	.7259448	3.431876
2002	827	.7505243	-.8597023	.7641528	3.190149
2003	827	.7853887	-1.020443	.7860328	2.790095
2004	827	1.024096	-1.385138	1.014639	3.641934
2005	827	1.03278	-.9080679	1.036636	3.142925
2006	827	1.097159	-.6041541	1.047806	3.672354
Total	4962	.9127703	-1.385138	.8836224	3.672354

```
. tabstat _b_cons, by(year) stat(N mean min p50 max) col(stat)
Summary for variables: _b_cons
     by categories of: year
```

year	N	mean	min	p50	max
2001	827	.001211	-.0072738	.0009004	.0114698
2002	827	.0005825	-.0050554	.0005992	.0101182
2003	827	.0009296	-.0026094	.0005611	.014773
2004	827	.0004305	-.0033237	.0002714	.0128312
2005	827	.0000746	-.0047677	-.0000388	.0072736
2006	827	.0000426	-.0041016	.0000312	.0080123
Total	4962	.0005451	-.0072738	.0003614	.014773

```
 . collapse (mean) _b* _se*, by(year)
. g bup = _b_vwretd + 1.96 * _se_vwretd
. g bdn = _b_vwretd - 1.96 * _se_vwretd
. g aup = _b_cons + 1.96 * _se_cons
. g adn = _b_cons - 1.96 * _se_cons
. lab var _b_vwretd "mean CAPM Beta"
. lab var bup "95% confidence interval"
. lab var bdn "95% confidence interval"
. lab var _b_cons "mean CAPM Alpha"
. lab var aup "95% confidence interval"
. lab var adn "95% confidence interval"
. tw (rarea bup bdn year) (scatter _b_vwretd year, c(l)), ///
> ti("CAPM Betas of US listed firms, 2001-2006") scheme(s2mono)
. graph export DurhamP1c_f1.pdf, replace
(file /Users/baum/Documents/Stata/StataWorkshops/DurhamP1c_f1.pdf written in PD
> F format)
. tw (rarea aup adn year) (scatter _b_cons year, c(l)), ///
> ti("CAPM Alphas of US listed firms, 2001-2006") scheme(s2mono)
. graph export DurhamP1c_f2.pdf, replace
(file /Users/baum/Documents/Stata/StataWorkshops/DurhamP1c_f2.pdf written in PD
> F format)
```


CAPM Betas of US listed firms, 2001-2006
1.5
1
.5
2001
2002
2003
2004
2005
2006
year
95% confidence interval
mean CAPM Beta


CAPM Alphas of US listed firms, 2001-2006
.006
.004
.002
0
-.002
2001
2002
2003
2004
2005
2006
year
95% confidence interval
mean CAPM Alpha