A BRIEF INTRODUCTION TO STATA WITH 50+ BASIC COMMANDS

Tobias Pfaff [*]

Institute for Economic Education, University of Münster

Version: October 2009

[*] Address: Scharnhorststr. 100, 48151 Münster, Germany, *tobias.pfaff@uni-muenster.de*

Table of contents

1. Why Stata?

Choosing a statistical software package in a company or research institution is often a strategic decision. The decision entails the investment of time and money, and you should think about the future development and compatibility of the software. Often, it is also influential what type of software your peer group is using, since this is usually the main source for getting support and exchanging experience.

The main software bundles for statistical computing are R (www.r-project.org), SAS (www.sas.com), SPSS (www.spss.com), and Stata (www.stata.com). However, there are many more packages in the market, some of them specialized for specific statistical problems. A general overview as well as a simple comparison of statistical packages is given on Wikipedia (http://en.wikipedia.org/wiki/Comparison_of_statistical_packages).

Statistical software can either be used by command line or by point-and-click menus, or both. The command line usage has the invaluable advantage that all steps of the analysis, and thus all results, are easily replicable. In contrast, menu usage might make it very difficult to replicate results, especially in larger projects. However, it might be more difficult in the beginning to learn a new command structure, especially for those users who have never worked with programming languages. Nonetheless, initial ease of use should be weighed against long-term payoffs before choosing the software.

Each software package can be distinguished by several factors. Some of the major factors are depicted (very) simplistically in the following table:[1]

	R	**SAS**	**SPSS**	**Stata**
Price	++ (free)	-	--	+
Command structure	+	+	--	++
Support	+	-	--	++
Ease of teaching	+	--	-	++

[1] This and the rest of this section has partly been adopted from: Acock, A. (2005), *SAS, Stata, SPSS: A Comparison*, Journal of Marriage & Family, 67 (4), pp. 1093–1095. And from: Mitchell, M. (2007), *Strategically using General Purpose Statistics Packages: A Look at Stata, SAS and SPSS*, Report No. 1, Technical Report Series, UCLA Academic Technology Services.

R:

R is a free software package which is designed for use with command line only. While being a language is one of R's greatest strengths, it can make it harder to learn for those without programming experience. However, once learnt, you are no longer subject to price increases. The developer's community ensures to constantly provide add-ons and also ensures that the software will continue to exist. R is extremely versatile in graphics, and generally good for people who really want to find out "what their data have to say".

SAS:

SAS is the second most costly package. It can be used with, both, command line and graphical user interface (GUI). SAS is particularly strong on data management (especially with large files), and good for cutting edge research. It covers many graphical and statistical tasks. The main focus is on business customers now.

SPSS:

SPSS is the first choice for the occasional user. However, it is the most expensive of the four. SPSS is clearly designed for point-and-click usage on the GUI. A command structure exists, but it is not well defined and sometimes inconsistent. SPSS is good for basic data management and basic statistical analysis, but rather weak in graphics. In the future, SPSS might be the weakest of the four packages with regard to the scope of statistical procedures it offers due to its main focus on business customers.

Stata:

Stata is designed for the usage by command line, but it also offers a GUI that allows for working with menus. The simple and consistent command structure makes it rather easy to learn. It is the cheapest of the packages that entail costs, and it offers additional reductions for the educational sector. Stata is relatively weak on ANOVA, but extraordinary on regression analysis and complex survey designs. Stata is completely focused on scholars. In the future, Stata may have the strongest collection of advanced statistical procedures.

2. How to work with the software

2.1. User-interface

The Stata user-interface consists of the following elements:

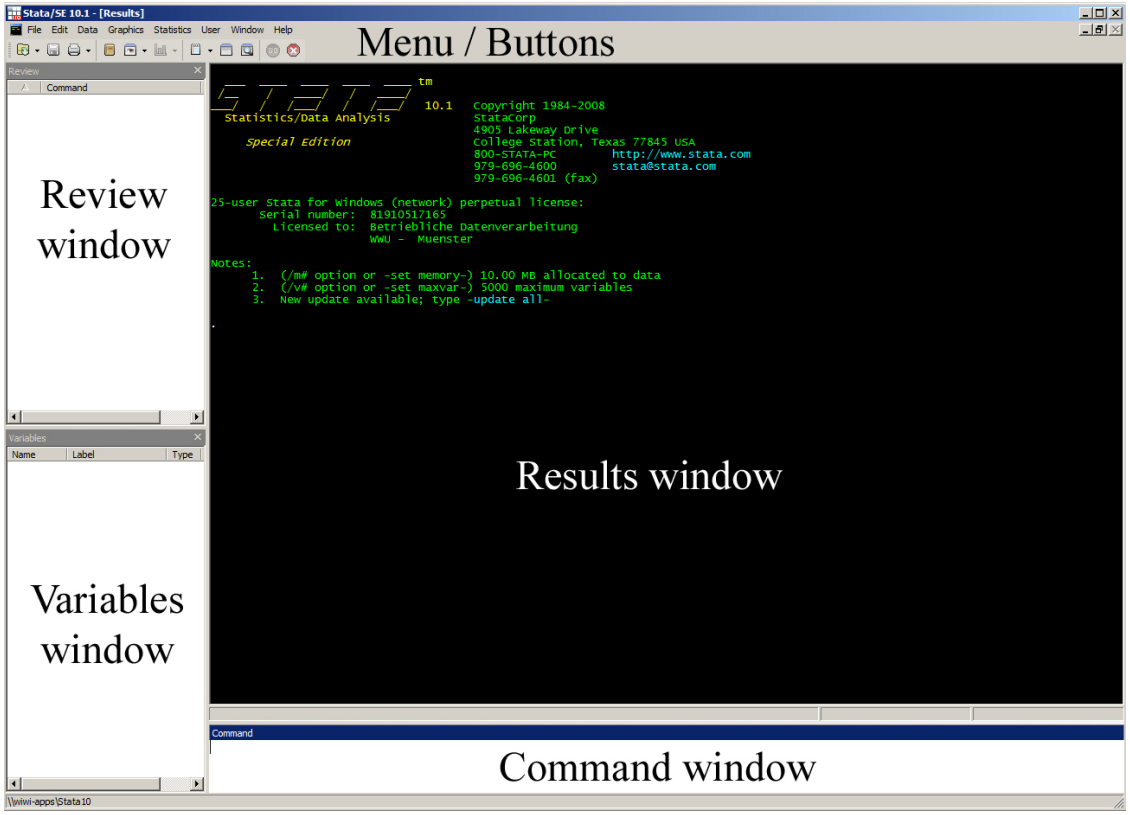


Figure 1: Screenshot of Stata user-interface

`Results window`

All outputs appear in this window. Only graphics will appear in a separate window.

`Command window`

This is the command line where commands are entered for execution.

`Variables window`

All variables in the currently open dataset will appear here. By clicking on a variable its name can be transferred to the command window.

`Review window`

Previously used commands are listed here and can be transferred to the command window by clicking on them.

`Buttons`

The most important button functions are the following:

- Open (use): Opens a new data file.
- Save: Saves the current data file.
- Print results: Prints the content of the results window.
- New Viewer: Opens a new viewer window, e.g. to open log-files.
- New Do-file Editor: Opens a new instance of the do-file editor (same as `doedit`).
- Data Editor: Opens the data editor window (same as `edit`).
- Data Browser: Opens the data browser (same as `browse`).
- Break: Allows to cancel currently running calculations.

`Menu`

Almost all commands can be called from the menu. However, we do not recommend to learn Stata using the menu commands since the command line will give the user much better control and allows for a much faster and more exact working process.

2.2. Do-files

The crucial advantage of using the command line instead of point-and-click menus is that it allows for the replication of results. However, all typed commands are lost once Stata is closed (unless you manually start a command log). This can be avoided by using so-called "do-files" where Stata commands are saved as a script in a simple text file with the ending ".do". When the do-file is run using the do-file editor all commands are executed subsequently. If all steps of a project have been documented in one or more do-files, all analyses and results can be reproduced and the whole process can be retraced by third party people.

However, saving all commands for a (bigger) project in a single do-file should be avoided. Rather, it is recommended to split up commands in several do-files named according to the respective step in the process (e.g., data import, data management, data

analysis). The following shows an example of how such a do-file cascade could look like in a project folder (leading numbers indicate the chronology of the working process):

Figure 2: Example of a project folder

2.3. Limits of the software

At certain points during your work with Stata you might encounter its limits. In order not to be surprised by sudden error messages, it is useful to be aware of the limits, which depend on the package version of Stata (the command `help limits` shows the limits):

Stata package:	Small	Intercooled	SE
	See www.stata.com/order for licence fees of the packages.		
# of observations	1,200	unlimited	unlimited
# of variables	99	2,047	32,767
# of characters in a command	8,697	67,800	1,081,527
# of options for a command	70	70	70
Length of a string variable	244	244	244
Length of a variable name	32	32	32

3. General commands

`update`

Stata offers a convenient update function over the internet. The update status of the currently installed Stata version can be compared with the one on the Stata website using `update query`. The actual update can then be performed with `update all`.

`cd`

Stata uses a working directory where datasets are saved if no path has been entered. The current working directory is displayed on the status bar on the bottom of the user-interface. It can also be displayed in the results window by using the command `pwd`. The working directory can be changed by using the command `cd` (**c**hange **d**irectory). An example would be:

→ **Example**: `cd D:/data/project1`

or: `cd data/project1` if you are already on drive D.

If a directory name contains spaces, the whole path has to be entered with quotation marks, e.g. `cd "C:/Documents and Settings/Admin/My Documents/data"`. Use `cd ..` (mind the space in between) to get to the subordinate directory. The content of the current working directory can be displayed with `dir`.

If the directory path is long, using the menu can save a lot of time: File > Change Working Directory.

An alternative way to get to a certain working directory is to open any dataset or do-file from the directory in which you want to work. Stata then automatically sets the working directory to this path. The dataset or do-file can be closed again, but the path is retained, which is sometimes quicker than entering the whole path with the `cd` command.

`help`

The help screen for any command can be displayed in a separate window with the help command:

→ **Syntax**: `help` *`command`*

→ **Example**: `help cd`

For functions using parentheses, like `sum()` for example, the brackets also need to be entered for the help: `help sum()`.

`findit`

The command `findit` is the best way to search for information on a topic across all sources, including the online help, the FAQs at the Stata web site, the Stata Journal, and all other Stata-related internet sources:

→ **Syntax**: `findit` *`word`* `[`*`word…`*`]` *Code in square brackets [] is optional*

→ **Example**: `findit anova`

You can look up the meaning of error messages by either clicking on the return code or by using `findit rc #`, whereas `#` stands for the number of the return code (e.g., `findit rc 131`).

`set memory`

Stata reads the whole dataset into the working memory, thus, sufficient memory has to be reserved (or an error message will be displayed). Therefore, you should set the size of the working memory reserved for Stata before loading a (big) dataset with the command `set memory`:

→ **Syntax**: `set memory` *`Xm`* `[, permanently]`

→ **Example**: `set memory 100m`

`X` represents the number of megabytes and the `permanently` option allows for a permanent setting of the respective memory size. Nonetheless, not more than ¾ of the

available memory of the computer should be reserved for Stata in order to guarantee a good performance of the system.

`display`

The `display` command displays strings and values of scalar expressions (e.g., 2+3) in the results window. Strings have to be entered with quotation marks, e.g. `display "Hello"` would simply print the word *Hello* on the screen. Interactively, `display` can be used as a substitute for a hand calculator, for example `display sqrt(5*6)+(7-2)^2` would return 30.477226 as a result:

→ **Syntax**: `display [`*`subcommand`* `[`*`subcommand`* `[...]]]`

→ **Example**: `display "Hello"`

4. Data input and saving

`insheet`

If the data come from an external source (SQL database, Excel, Access, SPSS, etc.) they first have to be read into Stata. In the external program the data should be exported as tab-separated, comma-separated or semi-colon-separated text (ASCII) files. This option can be often times found in the file menu under Save as… or Export… (e.g., in Excel under File → Save as → Tab-delimited (*.csv)). Other methods for reading non-Stata data are described in `help infiling`.

In Stata this text file is then read with the `insheet` command:

→ **Syntax**: `insheet using` *`filename`* `[,` *`options`*`]`

→ **Example**: `insheet using spss_income.dat, tab`

It can be specified in the options if the external data file is tab-separated or otherwise (see `help insheet`). The raw data needs then to be checked if the data are complete, and if further data management tasks need to be done. Common data management tasks are renaming of variables, changing string variables to numerical or date format, replacing comma as decimal separator with period, and labeling. Vice versa, data can be exported from Stata to a tab-separated text file with `outsheet using` *`filename`*.

use

Datasets with the Stata specific ending .dta can be opened with the `use` command:

→ **Syntax**: `use` *`filename`*`.dta`

→ **Example**: `use income_prepared.dta`

or: `use ../income_prepared.dta` for a file from a parent directory

Stata only opens a dataset if the data in memory are unchanged from their state on the disk. Otherwise, the memory can be cleared using `clear`, which also works as an option of `use (use` *`filename`*`.dta, clear`).

edit

Data can also be manually entered or changed using the data editor with `edit`.

compress

The dataset can be compressed using `compress`, where, if possible, variables will be saved in a format that needs less storage space.

save

Finally, the data is saved with the `save` command:

→ **Syntax**: `save` *`filename`*`.dta [, options]`

→ **Example**: `save income_prepared.dta, replace`

In do-files you would use the `replace` option most of the times as datasets are overwritten every time the do-file is.

5. Data management

5.1. General command syntax

Most of the Stata commands can be abbreviated. For example, instead of typing `generate`, Stata will also accept `gen`. The help screen demonstrates for each command how it can be abbreviated, by showing underlined letters in the syntax section of the help.

Stata syntax follows mostly the following basic structure, whereas square brackets denote optional qualifiers (see `help language`):

→ **Syntax**: `[by varlist1:] command [varlist2] [=exp] [if] [in] [using filename] [, options]`

→ **Example**: `bysort gender: tabulate age if weight < 50, nolabel`

A variable list (*varlist*) is a list of variable names with blanks in between. There are a number of shorthand conventions to reduce the amount of typing. For instance:

myvar	Just one variable
myvar var1 var2	Three variables
*myvar**	All variables starting with *myvar*
**var*	All variables ending with *var*
*my*var*	All variables starting with my and ending with *var*
my~var	A single variable starting with *my* and ending with *var*
my?var	All variables starting with *my* and ending with *var* with one other character between
myvar1-myvar6	*myvar1*, *myvar2*, ..., *myvar6* (probably)
this-that	All variables in the order of the variables window *this* through *that*

The `*` character indicates to match one or more characters. All variables matching the pattern are returned. The `~` character also indicates to match one or more characters, but unlike `*`, only one variable is allowed to match. If more than one variable match, an error message is returned. The `?` character matches a single character. All variables matching the pattern are returned. The `-` character indicates that all variables in the dataset, starting with the variable to the left of the `-` and ending with the variable to the right of the `-` are to be returned. Any command that takes `varlist` understands the keyword `_all` to mean all variables. Some commands are using all variables by default if none are specified (e.g., `summarize` shows summary statistics for all variables, and is equivalent to `summarize _all`).

by

The `by`-qualifier tells Stata to execute the subsequent command repeatedly along the different values of *varlist1* (not all commands support this feature, though). This

requires the data to be sorted by *varlist1*. Using `bysort` instead of `by` makes previous sorting redundant. An example would be to summarize happiness scores by gender:

```
. bysort gender: summarize happiness
-------------------------------------------------------
->    gender = female

      Variable    |     Obs   Mean  Std. Dev.   Min   Max
-------------------------------------------------------
      happiness   |     5     6.4   1.949359    4     9

-------------------------------------------------------
->    gender = male

      Variable    |     Obs   Mean  Std. Dev.   Min   Max
-------------------------------------------------------
      happiness   |     5     3.6   1.516575   2     6
```

`if`

`if` can be put at the end of a command in order to use only the data specified. `if` is allowed with most Stata commands.

→ **Example**: `summarize happiness if gender == "male"`

Several `if`-qualifiers can be used to define the range of the data, e. g.

→ **Example**: `summarize happiness if (age > 45 & happiness >= 5)`

`if`-qualifiers are connected with logical operators and are used with relational operators. Logical operators are:

`&`	AND
`|`	OR
`!`	NOT

Relational operators are:

`>`	Greater than
`<`	Less than
`>=`	Greater or equal
`<=`	Less or equal
`==`	Equal
`!=`	Not equal

Note that string values have to be put in quotation marks. Note also that Stata marks a missing value for numerical variables as `.` (period) and interprets it as infinite. This is important when referring to "bigger than" without wanting to include missing values. An appropriate statement would be for example:

→ **Example**: `summarize happiness if (age > 50 &! missing(age))`

`in`

The qualifier `in` at the end of a command means the command should only use the specified observations. `in` is allowed with most Stata commands:

→ **Example**: `summarize happiness in 1/10`

The syntax of the `in`-qualifier is the following:

`in 10`	Observation 10 only
`in 1/10`	Observations 1 through 10
`in 10/20`	Observations 10 through 20
`in -10/l`	Last 10 observations (beware: lowercase L, not 1, at end of range)

5.2. Commenting

There are several comment indicators in Stata:

`*`	At the beginning of a line
`/*` and `*/`	Everything in between is ignored
`//`	Can be used at the beginning or end of a line
`///`	Used to break long lines of code

The comment indicator `*` can only be used at the beginning of a line, and means that the whole line is ignored. The `/*` and `*/` comment delimiter can be used in the middle of a line or over several lines. Anything inside the two delimiters is ignored. The comment indicator `//` can be used at the beginning or at the end of a line. `//` means that the rest of the line is ignored (if it is at the end of a line it must be preceded by at least one blank). The `///` comment indicator tells Stata to view from `///` to the end of a line as a comment and to join the next line with the current line. This is how you can break long lines of code and make them more readable (again, if `///` is at the end of a line it must be preceded by at least one blank).

5.3. Data manipulation

generate

New variables are generated with the `generate` command:

→ **Syntax**: `generate [datatype] newvar =exp [if] [in]`

→ **Example**: `generate double new_income = old_income`

(see section 4.4. for information on data types and formats)

→ **Example**: `generate temp = .`

exp can be either an algebraic or a string expression. An empty algebraic variable can created with `generate varname = .`, an empty string variable with `generate varname = " "`. For an overview of functions that can be used in expressions, type `help functions`.

Arithmetical operators are:

+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Power

Important mathematical functions are (see `help math functions` for further mathematical functions):

`abs(x)`	Absolute value
`sqrt(x)`	Square root
`ln(x)`	Natural logarithm
`round(x)`	Round to nearest whole number

→ **Example**: `generate age_sq = age^2`

Important string functions are:

`substr(s,n1,n2)`	Extracts the substring of string *s* starting at *n1* for a length of *n2*
`subinstr(s1,s2,s3,n)`	Replaces the first *n* occurrences in *s1* of *s2* with *s3*

egen

Extensions to generate can be found in the `egen` command, which is used similarly to `generate`. `egen` offers a set of algebraic or string functions which are sometimes needed for data management tasks (see `help egen` for an overview of available functions):

→ **Syntax**: `egen [datatype] newvar = fcn(arguments) [if] [in] [, options]`

→ **Example**: `egen avg = mean(income)` Creates a variable with the average value of `income`

replace

The values of existing variables can be changed with the `replace` command. It works similar to the `generate` command expecting expressions and allowing for in- and if-qualifiers.

→ **Syntax**: `replace oldvar =exp [if] [in]`

→ **Example**: `replace income = income/100`

recode

The categories of a categorical variable can be conveniently changed with the `recode` command.

→ **Syntax**: `recode varlist (rule) [(rule)...] [, generate(newvar)]`

Rules have to be defined along the following example pattern:

`3 = 1`	3 recoded to 1
`2 . = 9`	2 and . recoded to 9
`1/5 = 4`	1 through 5 recoded to 4
`nonmissing = 8`	All other nonmissing to 8
`missing = 9`	All other missings to 9

Labeling can be done parallel in the command. The `generate` option allows the recoded variable to be saved as a new variable.

→ **Example**: `recode age_group (1=0 Low) (2=1 Medium), gen(n_age)`

drop

Variables or observations can be deleted using the `drop` command. Variables are deleted using the following version of `drop`:

→ **Syntax**: `drop` *varlist*

→ **Example**: `drop year91-year99`

Observations are deleted by applying another version of `drop`:

→ **Syntax**: `drop if` *exp* or `drop in` *range* `[if` *exp*`]`

→ **Example**: `drop if gender == "male"`

or: `drop in -100/l` for dropping the last 100 observations

keep

This command works opposite to `drop` as it keeps variables or observations rather than deleting them. Keeping variables is done with:

→ **Syntax**: `keep` *varlist*

→ **Example**: `keep year91-year99`

For keeping observations you use:

→ **Syntax**: `keep if` *exp* or `keep in` *range* `[if` *exp*`]`

→ **Example**: `keep if gender == "male"`

destring

After importing data into Stata from external sources, variables containing only numbers are sometimes saved in string format. Thus, no arithmetic operations can be performed with such a variable. It might be necessary to remove or replace non-numeric characters prior to converting the string variable into numerical format (e.g., decimal separator comma instead of period). The actual conversion can then be performed with the `destring` command:

→ **Syntax**: `destring [`*varlist*`], {generate(`*newvarlist*`)|replace}`

Variables in *varlist* that are already numeric will not be changed. Also, if any non-numerical character is found this variable will not be changed.

Either replace or `generate(varlist)` must be specified. `replace` specifies that the variables in *varlist* should be converted to numeric variables and replaced. New variables can be generated with `generate(varlist)`.

→ **Example**: `destring year91-year98, replace`

5.4. Data formatting

`rename`

A variable can be renamed with the `rename` command:

→ **Syntax**: `rename old_varname new_varname`

→ **Example**: `rename income hh_income`

If only a common prefix of several variables shall be renamed `renpfix` can be used:

→ **Syntax**: `renpfix old_stub [new_stub]`

→ **Example**: `renpfix year yr`

`recast`

Stata offers different data types (see `help data types`) for storaging variables. The data type has influence on the amount of memory that is needed. For datasets with a huge number of observations, the data or storage type can have significant influence on the performance of Stata. Usually, you would want to use the data type consuming the least amount of memory, while saving all the information contained in the variable.
Strings are stored as `str#` (e.g., `str1`, `str2`, ..., `str244`). The number after the `str` indicates the maximum length of the string. Numerical variables are stored as `byte`, `int`, `long`, `float`, or `double`, with the default being `float`. `byte`, `int`, and `long` are said to be of "integer" type in that they can hold only integers. If you are need to store precise results where interpretations are sensitive to a high decimal precision of the number, then `double` would be the most appropriate data type. Data types of the existing variables can be seen using the `describe` command. The storage type can then be changed with the `recast` command:

→ **Syntax**: `recast datatype varlist`

→ **Example**: `recast double perc_results`

format

format allows you to specify the display format for variables. The internal precision of the variables is unaffected. Various format types exist for string, date, and numerical variables (see help format):

→ **Syntax**: format *varlist %fmt*

→ **Example**: format income %9.2g

label

There are two ways to label variables. The first one is to label the variable itself:

→ **Syntax**: label variable *varname* ["*label*"]

→ **Example**: label variable hh_income "Household income"

The second option is to assign labels to the values of categorical variables. This is done in two steps. First, a value label has to be defined:

→ **Syntax**: label define *lblname # "label" [# "label" ...]*

→ **Example**: label define city_label 1 "Bonn" 2 "Hamburg"

Second, this value label is assigned to the respective variable:

→ **Syntax**: label values *varname* [*lblname*]

→ **Example**: label values city city_label

5.5. Data description

describe

General information about the dataset can be retrieved with describe. The command displays the number of observations, number of variables, the size of the dataset, and lists all variables together with basic information (such as storage type, etc.).

codebook

The codebook command delivers information about one or more variables, such as storage type, range, number of unique values, and number of missing values. The command offers further interesting features which can be seen with help codebook.

→ **Syntax**: `codebook [varlist] [if] [in] [, options]`

→ **Example**: `codebook income`

`sort`

Data is sorted in ascending order with the `sort` command:

→ **Syntax**: `sort varlist`

→ **Example**: `sort gender age income`

Descending ordering can be done with `gsort`, whereas a minus in front of a *varname* invokes descending order:

→ **Syntax**: `gsort [+|-] varname [[+|-] varname ...]`

→ **Example**: `gsort -age income`

`order`

The order of the variables as seen in the variable window can be changed with the `order` command:

→ **Syntax**: `order varlist`

→ **Example**: `order person_id date`

The command orders variables in the variables windows in the order of *varlist*. The command `order, alphabetic` puts all variables in alphabetical order. A single variable can be moved to a specified position with e.g. `order, before(varname)`.

`browse`

The data browser can be opened with the `browse` command:

→ **Syntax**: `browse [varlist] [if] [in] [, nolabel]`

→ **Example**: `browse age income`

It does not allow data manipulation (as does `edit`), but data can be sorted using the sort button. Sometimes it is useful not to display value labels in the data browser. This can be done using the `nolabel` option.

`list`

Similar to the data browser, values of variables can be listed in the results window with the `list` command. Here, if- and in-qualifiers are often useful:

→ **Syntax**: `list [varlist] [if] [in] [, options]`

→ **Example**: `list age income in 1/10`

`assert`

In large datasets, it is difficult to check every single observation with `browse` or `list`. Here, the command assert is often useful. It verifies whether a statement is true or false.

→ **Syntax**: `assert exp [if] [in]`

→ **Example**: `assert age>0` Checks that no value for age is negative.

If the statement is true, `assert` does not give any output in the results window. On the other hand, if it is false, `assert` displays an error message together with the number of contradictions.

`summarize`

The most important descriptive statistics for numerical variables are delivered with the `summarize` command:

→ **Syntax**: `summarize [varlist] [if] [in] [weight] [, options]`

→ **Example**: `browse age income`

It displays the number of (non-missing) observations, mean, standard deviation, minimum, and maximum. Additionally, `summarize varlist, detail` shows certain percentiles (including median), skewness, and kurtosis. User specific percentiles can be shown with `centile`. Tables of summary statistics can be drawn with `table`.

`tabulate`

One-way frequency tables for categorical variables can be drawn with the `tabulate` command:

→ **Syntax**: `tabulate varname [if] [in] [weight] [, options]`

→ **Example**: `tabulate city`

Two-way cross-tables for two categorical variables can be drawn with another version of tabulate:

→ **Syntax**: `tabulate` *`varname1 varname2`* `[if] [in] [weight] [, options]`

→ **Example**: `tabulate city age_group`

`inspect`

The `inspect` command provides a quick summary of a numeric variable that differs from that provided by `summarize` or `tabulate`:

→ **Syntax**: `inspect [varlist] [if] [in]`

→ **Example**: `inspect income`

It reports the number of negative, zero, and positive values; the number of integers and non-integers; the number of unique values; and the number of missing values; and it produces a small histogram. Its purpose is not analytical but it allows to quickly gain familiarity with unknown data.

5.6. Data merging

`append`

A second dataset can be appended to the end of the one currently used by using the `append` command. If the data types of the variables are not the same Stata will promote data types and will keep all variables when the two datasets have differing variable names:

→ **Syntax**: `append using` *`filename`*`.dta [, options]`

→ **Example**: `append using income_new.dta`

`merge`

Datasets sharing the same kind of observations, but having different variables, can be joined with the `merge` command. Then, the currently used dataset ("master" dataset) is extended with the corresponding observations from one or more other files ("using" datasets):

→ **Syntax**: `merge 1:1 varlist using filename.dta [, options]`

One-to-one merge

→ **Syntax**: `merge m:1 varlist using filename.dta [, options]`

Many-to-one merge

→ **Syntax**: `merge 1:m varlist using filename.dta [, options]`

Onee-to-many merge

→ **Syntax**: `merge m:m varlist using filename.dta [, options]`

Many-to-many merge

→ **Syntax**: `merge 1:1 _n using filename.dta [, options]`

One-to-one merge by observation

The "master" and "using" datasets need to share at least one common variable, the so-called primary key, in order to make the match possible. The match variable(s) is (are) defined in *varlist*.
After merging, Stata automatically generates a variable which contains information about the matching of the data:

`_merge == 1`	Observations only from "master" dataset
`_merge == 2`	Observations only from "using" dataset(s)
`_merge == 3`	Observations from "master" and "using" dataset(s)
`_merge == 4`	Observations from both, missing values updated
`_merge == 5`	Observations from both, conflicting nonmissing values

Note that if `_merge` only contains 3's, this means that "master" and "using" dataset(s) all have the same observations

The option `keepusing(varlist)` specifies the variables to be kept from the "using" dataset. If `keepusing()` is not specified, all variables are kept. Another option `nokeep` causes `merge` to ignore observations in the "using" dataset that have no corresponding observation in the "master". This is equal to deleting all observations that are marked with `_merge==2`. An example for merging would be to add demographical information to the income data of respondents which is stored in a different dataset:

→ **Example**: `merge 1:1 id using demogr.dta, keepusing(gender age)`

6. Further issues

6.1. Log files

Everything that runs through the results window can be recorded with so-called log files. These log files can then be printed or saved in other file formats so that the analysis can be retraced independently of Stata. The recording with a log file can be started with `log using filename`:

→ **Syntax**: `log using filename [, replace append]`

→ **Example**: `log using 0140_analysis_group1.scml, replace`

The option append specifies that results should be appended to an existing file. The replace option replaces an existing log file. Stata. The command be retraced independently of Stata. Eventually, the log file is closed using `log close`.

6.2. Graphs

One of the advantages of Stata is its vast graphics capabilities. On the other hand, commands for comprehensive graphs can get quite long, and it takes some time to get used to the code structure. Using dialog boxes might have an advantage in certain cases. The starting point for learning about graphs is `help graph`. Besides, Stata help offers a separate tutorial for basic graphs that can be accessed with `help graph_intro`. An example for a simple bar graph of the variables `pop_north` and `pop_south` would be:

→ **Example**: `graph bar pop_north pop_south`

Some graph commands are typed without the leading `graph`. For example, a basic histogram of the variable `age` would be:

→ **Example**: `histogram age`

Graphs are not saved in log files. In order to view them independently of Stata they need to be saved with the `graph export` command.

6.3. Probability distribution and density functions

Stata offers a wide range of distribution and density functions. Available functions are shown with `help density_functions`. For example, the value of the cumulative standard normal distribution of an existing variable `s` is generated with:

→ **Example**: `gen s_norm = normal(s)`

6.4. Random number generation

Random numbers can be drawn with one of Stata's random number functions (see `help random_number_functions`).

→ **Example**: `gen var1 = uniform()` Uniform

→ **Example**: `gen var2 = rnormal()` Standard normal

7. Shortcuts (that make "life" easier)

If you spend a lot of time using Stata it makes sense to use shortcuts that allow you to work quicker. First of all, most commands can be abbreviated. The maximum abbreviation can be seen in the respective help entry in the syntax section as the underlined part of the command. For example, `summarize` can be abbreviated to `su`, `rename` to `ren`.

Variable names can be abbreviated, too. For example, the dataset has the variables `pop_north` and `pop_south`. Stata would accept `sum pop_s`. Instead, `sum pop` would cause an error message due to ambiguous abbreviation. While typing commands in the command window, Stata completes variable stubs when the tab key is pressed.

A click on recent commands or variable names from the review or variables window brings them automatically to the command window. Also, it is possible to cycle back and forth through previous commands using the PageUp and PageDown keys.

8. Some sample do-files

8.1. Example for importing data

```
***
*** Import of raw data
***
*** (Hans Müller, 10/01/2008, version 1.0.0)
***

version 11

set more off

foreach domain in demographics 01_good_governance ///
   02_community_vitality 03_education 04_health ///
   08_living_standard 09_psychological_wellbeing {
```

```
   clear

   insheet using _raw_spss/`domain'.dat, tab

   quietly destring, replace

   sort respno

   save _raw_stata/`domain'_raw.dta, replace
}
```

8.2. Example for preparing data

```
***
*** Preparation of demographics data
***
*** (Hans Müller, 10/01/2008, version 1.0.0)
***

version 11

set more off

use _raw_stata/demographics_raw.dta, clear


** DE-STRING VARIABLES

foreach var in nfe gomam {
     // Replace comma as decimal separator with period
     replace `var' = subinstr(`var',",",".",1)
     destring `var', replace
}


** CLASSIFY AGE

gen n_age_group = irecode(age,17,30,45,60,.)+1

recode

replace n_age_group = 1 if (age <= 17)
replace n_age_group = 2 if (age >= 18 & age <= 30)
replace n_age_group = 3 if (age >= 31 & age <= 45)
replace n_age_group = 4 if (age >= 46 & age <= 60)
replace n_age_group = 5 if (age >= 61 & age != .)

assert n_age_group != missing(n_age_group)

label define age_group_labels 1 "0-17" 2 "18-30" 3 "31-45" ///
     4 "46-60"  5 ">60"
label values n_age_group age_group_labels
```

```
** RE-CLASSIFY MARITAL STATUS

gen n_marital = marital

recode n_marital (4 5 = 3)
assert n_marital != missing(n_marital)

label define n_marital_labels 1 "Never married" 2 "Married" ///
     3 "Divorced/separated/widowed"
label values n_marital n_marital_labels


save demographics_prepared.dta, replace
```

8.3. Example for analyzing data

```
***
*** Analysis of health data
***
*** (Hans Müller, 10/01/2008, version 1.0.0)
***

version 11

set more off

use 04_health_prepared.dta, clear


** MERGE FILES FOR ADDITIONAL VARIABLES

merge 1:1 respno using 09_psychological_wellbeing_prepared.dta, ///
     keepusing(hap n_happiness_group lifequa1 lifequa2 ///
     n_lifesatisfaction n_social_support n_social_support_group ///
     n_ghq n_ghq_group stress1 spirit1)
assert _merge==3
drop _merge

merge 1:1 respno using demographics_prepared.dta, keepusing(dzcode ///
     sex age n_age_group n_marital spoken hhsize)
assert _merge==3
drop _merge

merge 1:1 respno using 08_living_standard_prepared.dta, ///
     keepusing(indincom n_income_group finsec2)
capture br if _merge!=3
drop _merge


log using 0140_analysis_04_health.smcl, replace  // Start logging


** HEALTH STATUS DESCRIPTIVE STATISTICS

sum hstatus, detail
```

```
tabstat hstatus, stat(n mean median max min skewness kurtosis)

codebook hstatus, problems detail

hist hstatus, freq bin(3) name(hist_health_status)

kdensity hstatus, normal name(kdensity_health_status)


log close
```

8.4. Example for a do-file that runs the entire project

```
***
*** Run all do-files of survey project
***
*** (Hans Müller, 10/01/2008, version 1.0.0)
***

version 11

do 0001_import_files.do
do 0005_prepare_demographics.do
do 0010_prepare_01_good_governance.do
do 0020_prepare_02_community_vitality.do
do 0030_prepare_03_education.do
do 0040_prepare_04_health.do
do 0080_prepare_08_living_standards.do
do 0090_prepare_09_psychological_wellbeing.do
do 0140_analysis_04_health.do
do 1000_analysis_happiness_correlations.do
```

9. What is not captured in this introduction

The following basic features of Stata are not covered by this introduction:

- Macros (`help macro`)
- Loop programming (`help forvalues`, `help foreach`, `help while`)
- `if` and `else` programming commands (`help ifcmd`)
- System variables (`help _variables`)
- Temporary saved results (`help return`)
- Suppressing of output (`help quietly`)
- Capturing error messages (`help capture`)
- Temporary storage of dataset (`help preserve`, `help restore`)
- Debugging (`help trace`)
- Row-column transposition (`help xpose`, `help reshape`)

- Collapsing into summary dataset (`help collapse`, `help table`)
- Duplicate management (`help duplicates`)
- Date functions (`help date functions`)
- Weighting (`help weight`)

10. Where to find help

Stata has more than 800 different commands and you will surely get to a point where help is needed. Luckily, Stata is a statistical software package that offers convenient options to look for help.

10.1. In-built help and printed manuals

For every command Stata's in-built help can be called with `help` *`command`*. The information you can find there is an abbreviated version of the Stata manuals that come as pdf with the software. In the manuals, each command is extensively discussed and there are separate manuals for graphics, panel data, and survey data.

10.2. Online resources

Stata's webpage offers various articles collected in a FAQ. Also, the homepage has a very active forum which shows the questions and answers that Stata users have posted via email to a newslist ("Statalist"). It is not unusual to receive an answer within a couple of hours. Furthermore, online tutorials and introductions offer help that goes beyond this basic introduction. Links to some online resources are given in the following:

- Stata FAQ's
  → http://www.stata.com/support/faqs/
- Stata Forum
  → http://www.stata.com/statalist/archive/
  → How to post: http://www.stata.com/support/faqs/res/statalist.html#howto
- Statistical Computing Resources at UCLA
  → http://www.ats.ucla.edu/stat/stata/default.htm
- Statistical Tests Overview (with Stata commands)
  → http://www.wiwi.uni-muenster.de/ioeb/en/organisation/pfaff/stat_overview.html

- Resources for learning Stata

  → http://www.stata.com/links/resources1.html

- Princeton Stata Tutorial

  → http://www.princeton.edu/~erp/stata/main.html

- Baum, Christopher (2005): Introduction to Stata at Boston College

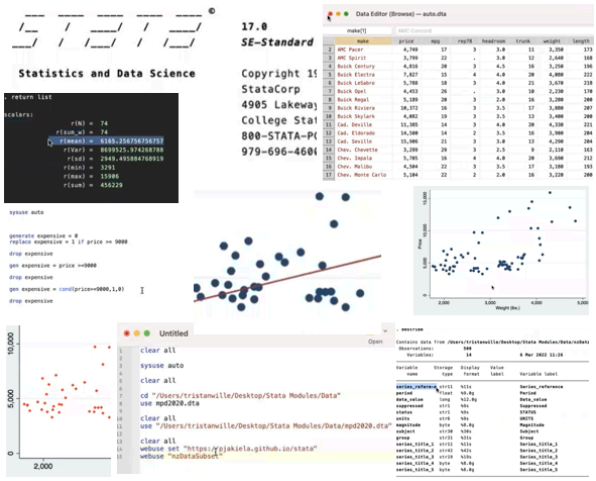
  → http://fmwww.bc.edu/GStat/docs/StataIntro.pdf

11. Literature

Some very useful books are:

- Acock, Alan C. (2006): *A Gentle Introduction to Stata*, Stata Press.
- Baum, Christopher F. (2006): *An Introduction to Modern Econometrics Using Stata*, Stata Press.
- Hamilton, Lawrence C. (2006): *Statistics with Stata*, Brooks/Cole.
- Kohler, Ulrich/Kreuter, Frauke (2005): *Data Analysis Using Stata*, Stata Press.
- Kohler, Ulrich/Kreuter, Frauke (2008): *Datenanalyse mit Stata*, 3rd ed., Oldenbourg.
- Mitchell, Michael (2008): *A Visual Guide to Stata Graphics*, Stata Press.
- Rabe-Hesketh, Sophia/Everitt, Brian (2007): *A Handbook of Statistical Analyses Using Stata*, Chapman & Hall.

stata



Faculty:
Pamela Jakiela
Owen Ozier

Student contributors:
Tristan Wille '22
Lily Levin '23
Agustin Aliaga '25

tables

A Regression Table

The stata commands `eststo` and `esttab` allow you to make attractive, self-contained tables with very little effort. Before you begin, set up a do file that downloads data from the paper Price Subsidies, Diagnostic Tests, and Targeting of Malaria Treatment: Evidence from a Randomized Controlled Trial by following the instructions here.

Storing Regression Results with `eststo`

`eststo` is a Stata command that allows you to save the results of a regression. Immediately after you run any regression, your results are saved in a collection of local macros and matrices (you can see what is saved by typing `ereturn list` immediately after running your regression). These locals are over-written as soon

as you run another regression - so we need to save them somewhere. That is what `eststo` does.

`eststo` is very easy to use. You can simply type `eststo` after running any regression. Alternatively, you can precede each regression command with `eststo:`, as you see in this example:

```
eststo:  reg c_act coartemprice
```

By default, `eststo` saves the results from your first regression as `est1`, the results from your second regression as `est2`, etc. But you can provide alternative names if you prefer: just type `eststo` followed by your preferred name for a particular specification (either after running the regression or before the colon in a single line of code).

To see which regression results are currently stored in memory, type `eststo dir`. The erase them and start fresh, type `eststo clear`. It is worth taking a look at the help file for `eststo` to familiarize yourself with its syntax before proceeding.

Exporting Regression Results to Word

Suppose you run the following regressions and you want to export your results as a regression table:

```
eststo clear
eststo:  reg c_act coartemprice
eststo:  reg c_act coartemprice b_*
```

Simply typing `esttab` after storing the results of these regressions will produce a decent-looking regression table in Stata's output window:

```
. esttab
```

	(1) c_act	(2) c_act
coartemprice	-0.000451***	-0.000413***
	(-4.41)	(-3.64)
b_h_edu		0.0117
		(1.93)
b_knowledg~t		0.0591
		(1.35)
b_hh_size		0.00765
		(0.89)
b_acres		0.0102
		(1.23)
b_dist_km		0.0495*
		(2.23)
b_h_age_im~d		-0.00378*
		(-2.42)
b_h_age_mi~g		-0.169
		(-1.51)
_cons	0.423***	0.307**
	(15.72)	(3.04)
N	575	460

t statistics in parentheses
* p<0.05, ** p<0.01, *** p<0.001

If you want to export this table to a word document, you can use the command

```
esttab using myregtable.rtf, replace
```

which will save a rich text format (rtf) file in your working directory, which you can then open using word.

Publication-Ready Tables

You can clean up your table by labeling your variables using the `label var` command (note: you must do this **before** you run your regressions). When variable names appear in otherwise finished tables, readers often have a hard time knowing what the variables are: names like `b_dist_km` or `txpostxfem` may not be immediately self-eplanatory. And they do not look very professional. If your variables are labeled, you can invoke `esttab`'s `label` option to use labels rather than variable names.

Variable labels should be short, so that they do not wrap over multiple lines in your table. They should also be self-expanatory. You can include additional information in the table notes when necessary.

If you have categorical variables that can be replaced with easier-to-interpret dummy variables, this might be a good opportunity to transform them. So, for example, you can include a rural dummy labeled "Rural" instead of a `region` variable labeled "Region of residence: urban = 1, rural = 2". This will make it easier for readers to immediately interpret your regression coefficients.

Having relabeled the data, you can use `esttab` to generate a new version of your regression table. If the text of your variable labels wraps onto a second line, you can make the first column of the table wider using `esttab`'s `varwidth()` option. In the example below, I used `varwidth(28)` You can also set the width of your columns of coefficient estimates using the `modelwidth()` option. I used `modelwidth(16)`. It's not clear what the units are, but higher numbers lead to wider columns. This gives you a fairly professional looking table:

```
. esttab, label varwidth(28) modelwidth(16) replace
```

	(1) Treated with ACT	(2) Treated with ACT
Randomized Price	-0.000451***	-0.000413***
	(-4.41)	(-3.64)
Education		0.0117
		(1.93)
Malaria Knowledge		0.0591
		(1.35)
Household Size		0.00765
		(0.89)
Acres of Land		0.0102
		(1.23)
Distance to Health Center		0.0495*
		(2.23)
Age		-0.00378*
		(-2.42)
Age Data Missing		-0.169
		(-1.51)
Constant	0.423***	0.307**
	(15.72)	(3.04)
Observations	575	460

```
t statistics in parentheses
* p<0.05, ** p<0.01, *** p<0.001
```

Before publishing this table, you'd want to find out why your two regression specifications include different numbers of observations. Otherwise, you won't know whether any difference in the coefficient of interest between Column 1 and Column 2 results from adding the controls or changing your analysis sample. You should always make sure that the columns in your regression tables contain identical numbers of observations (unless you are varying the sample intentionally, for example if you were looking at treatment effects on women in one column and treatment effects on men in another column).

You might also notice that the coefficient estimates associated with with `coartemprice` variable are very long - the begin with three zeroes after the decimal point. To correct this, simply divide the `coartemprice` variable by 100 (or even 1000) before running your regressions. This will not alter any of your other

coefficients, but it will rescale the coefficients on price so that they fit into the table more easily. (Of course, this changes the interpretation slightly: the coefficient would then indicate the change in your outcome variable resulting from a 100 shilling increase in the price of coartem rather than a one shilling price increase.)

Additional esttab Options

Economists typically report standard errors rather than t-statistics in parentheses. You can achieve this by invoking `esttab`'s `se` option. If you want to specify how many digits to report, you can do this adding a number in partheses after either `b` (for the coefficient estimates) or `se` (for the standard errors). So, if for some reason you wished to report coefficient estimates to two decimal places and standard errors to three, you would use the command

```
esttab, b(2) se(3)
```

You can change the headings of the columns reporting regression coefficients with `esttab`'s `mtitles()` option. By default, columns of coefficients will be numbered, and either the name of the dependent variable or its label will appear as well. However, when the outcome variable does not differ across columns, it often makes more sense to use more informative columns headings. For example, if your first specification was OLS and your second specification was probit, you might want to label your columns accordingly:

```
esttab, mtitles(OLS Probit)
```

If you wish to use multi-word column headings, you can put each one in quotes. For example:

```
esttab, mtitles("Rural Areas" "Urban Areas")
```

You may not always want to report the coefficients on your control variables - particularly when your specification includes a large number of fixed effects. The `esttab` option `keep()` allows you to provide a list (in parentheses) of the variables that you want to appear in your table. Alternatively, you can use the option `drop()` to indicate which variables should be surpressed. The option `indicate()` drops a specific set of variables from the table (like `drop()`) but also creates an additional row at the bottom of the table indicating which specifications include the dropped variables. For example:

```
eststo clear
eststo:  reg c_act coartemprice
eststo:  reg c_act coartemprice b_*
esttab, label indicate(Baseline Controls = b_*) replace
```

produces the table below.

```
. esttab, label indicate(Baseline Controls = b_*) varwidth(28) modelwidth(16)
```

	(1) Treated with ACT	(2) Treated with ACT
Price (1000s of Shillings)	-0.397***	-0.413***
	(-3.43)	(-3.64)
Constant	0.398***	0.307**
	(13.40)	(3.04)
Baseline Controls	No	Yes
Observations	460	460

t statistics in parentheses
* p<0.05, ** p<0.01, *** p<0.001

You can also use the `note()` option to add any relevant information in the table notes. You will typically want to indicate what specification you are using (i.e. OLS or logit), what level you are clustering you standard errors at (or, if you are not clustering, that they are robust), and which controls are included in which specifications, if that is not obvious from the body of your table.

Exporting Regression Results to LaTeX or Excel

The `esttab` command can also be used to export regression results to Excel or LaTeX. To export to Excel, just name your new file as a csv file rather than an rtf file:

```
esttab using myregtable.csv, replace
```

A nice thing about exporting to Excel is that you can make additional modifications (for example, to format borders) using the `putexcel` command.

`esttab` can also export your table to LaTeX - all you need to do is give your new file a name that ends in tex. For example, if you export your regression results using the code

```
esttab using myregtable.tex, label b(2) se(2) nostar replace ///
   title(Regression table\label{tab1})
```

you can compile a pdf of the table in overleaf or any other LaTeX compiler with the LaTeX code:

```
\documentclass[12pt]{article}
\begin{document}
\input{myregtable.tex}
\end{document}
```

A do File

A do file containing the code used in these examples is available here.

27 Commands everyone should know

Contents

27.1 41 commands

Putting aside the statistical commands that might particularly interest you, here are 41 commands that everyone should know:

Getting help	**[U] 4 Stata's help and search facilities**
`help`, `net search`, `search`	
Keeping Stata up to date	
`ado`, `net`, `update`	**[U] 28 Using the Internet to keep up to date**
`adoupdate`	[R] **adoupdate**
Operating system interface	
`pwd`, `cd`	[D] **cd**
Using and saving data from disk	
`save`	[D] **save**
`use`	[D] **use**
`append`, `merge`	**[U] 22 Combining datasets**
`compress`	[D] **compress**
Inputting data into Stata	**[U] 21 Entering and importing data**
`import`	[D] **import**
`edit`	[D] **edit**
Basic data reporting	
`describe`	[D] **describe**
`codebook`	[D] **codebook**
`list`	[D] **list**
`browse`	[D] **edit**
`count`	[D] **count**
`inspect`	[D] **inspect**
`table`	[R] **table**
`tabulate`	[R] **tabulate oneway** and [R] **tabulate twoway**
`summarize`	[R] **summarize**

Data manipulation	[U] 13 Functions and expressions
generate, replace	[D] generate
egen	[D] egen
rename	[D] rename, [D] rename group
clear	[D] clear
drop, keep	[D] drop
sort	[D] sort
encode, decode	[D] encode
order	[D] order
by	[U] 11.5 by varlist: construct
reshape	[D] reshape
Keeping track of your work	
log	[U] 15 Saving and printing output—log files
notes	[D] notes
Convenience	
display	[R] display

27.2 The by construct

If you do not understand the by *varlist*: construct, _n, and _N, and their interaction, and if you process data where observations are related, you are missing out on something. See

[U] 13.7 Explicit subscripting
[U] 11.5 by varlist: construct

Say that you have a dataset with multiple observations per person, and you want the average value of each person's blood pressure (bp) for the day. You could

```
. egen avgbp = mean(bp), by(person)
```

but you could also

```
. by person, sort: gen avgbp = sum(bp)/_N
. by person: replace avgbp = avgbp[_N]
```

Yes, typing two commands is more work than typing just one, but understanding the two-command construct is the key to generating more complicated things that no one ever thought about adding to egen.

Say that your dataset also contains time recording when each observation was made. If you want to add the total time the person is under observation (last time minus first time) to each observation, type

```
. by person (time), sort: gen ttl = time[_N]-time[1]
```

Or, suppose you want to add how long it has been since the person was last observed to each observation:

```
. by person (time), sort: gen howlong = time - time[_n-1]
```

If instead you wanted how long it would be until the next observation, type

```
. by person (time), sort: gen whennext = time[_n+1] - time
```

by *varlist*:, _n, and _N are often the solution to difficult calculations.