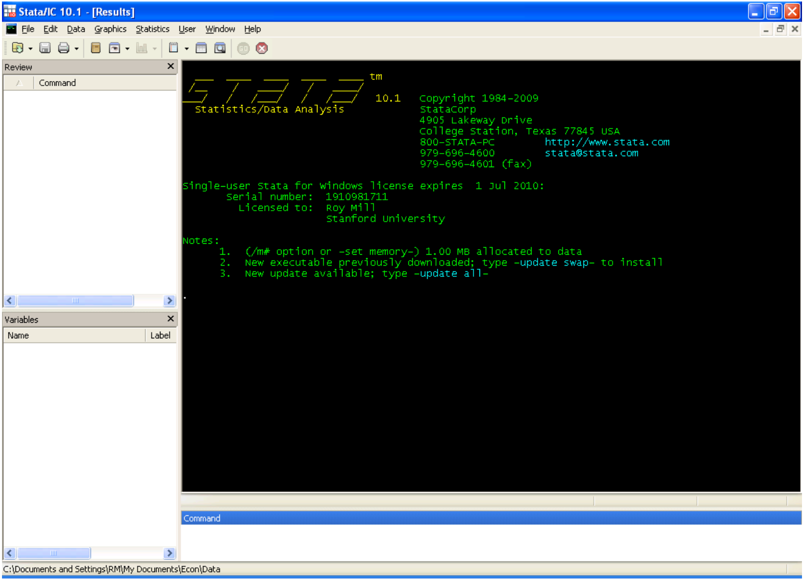
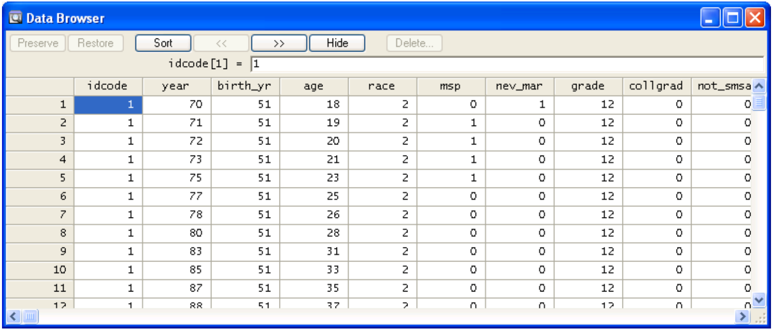
INTRODUCTION TO STATA

Stata - love at first sight?



Datasets

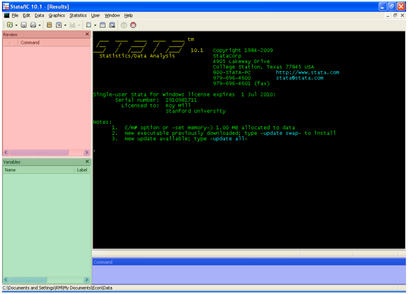
Datasets are the objects of statistical analysis. They contain a matrix of which rows represent different observations (draws of random variables) and the columns are the variables.
Each cell contains the value of the variable for the observation in question:



	idcode	year	birth_yr	age	race	msp	nev_mar	grade	collgrad	not_smsa
1	1	70	51	18	2	0	1	12	0	0
2	1	71	51	19	2	1	0	12	0	0
3	1	72	51	20	2	1	0	12	0	0
4	1	73	51	21	2	1	0	12	0	0
5	1	75	51	23	2	1	0	12	0	0
6	1	77	51	25	2	0	0	12	0	0
7	1	78	51	26	2	0	0	12	0	0
8	1	80	51	28	2	0	0	12	0	0
9	1	83	51	31	2	0	0	12	0	0
10	1	85	51	33	2	0	0	12	0	0
11	1	87	51	35	2	0	0	12	0	0
12	1	88	51	37	2	0	0	12	0	0

Main windows

- Results (black) - text output of the commands you run
- Command (blue) - allows to enter commands to run
- Review (red) - shows previously run commands
- Variables (green) - shows the variables in the loaded dataset



Menu and Upper bar

-  - Open a *data* file
-  - Save dataset in memory to a file on disk
-   - Open data editor (left), or data browser (right), for dataset in memory
-  - Start a new do-file in a new do-file editor window
-  - Stop execution of a command

From the menu you can easily call forms that will run some commands for you. For example: “Data → Describe Data → Summary Statistics” will open a form and then run the `summarize` command accordingly.

Commands and Syntax Conventions

Commands in Stata usually take the following form:

```
<command name> [... something ...] [if] [in] [, options]
```

A few conventions:

- Angle brackets - ⟨⟩ - mean that you *must* put something in their place. In other words, they are *mandatory*
- Square brackets - [] - mean that you *may* put something in their place. In other words, they are *optional*
- General syntax will be in blue color, specific examples in green

For example:

```
use mydataset.dta, clear
drop if male==1
save mydataset_females.dta, replace
```

Use and Save

Stata works with a single dataset in memory. It can work with multiple files, but not at the same time. You will need to load the dataset to the internal memory from the disk and you can save it back to the disk after you are done.

To load a dataset into the memory we run the `use` command:

```
use <file path> [, clear]
```

- file path - required path to the dta file you want to load.
- clear - Ignore existing dataset in memory, even if unsaved.

Use and Save

Now, after messing with the file, we might want to save it on file for later use.

```
save <file path> [, replace]
```

- file path - required path to the dta file you want to save.
- replace - If a filename by that name in this folder exists, replace it with the dataset in memory.

Notes:

- The `clear` and `replace` options will appear in the future and will have the same use: `clear` will overwrite the current dataset in memory and `replace` will overwrite the file on disk.
- If you will use the icons in the upper bar for loading and saving datasets, Stata will actually run the the `use` and `save` commands.

Use and Save

Examples:

```
use "C:\Documents and Settings\RM\Papers\RawData.dta", clear


// Alternatively...
cd "C:\Documents and Settings\RM\Papers"
use RawData, clear


// (... do some stuff ...)

// Now save it to the "data" subfolder
save data\GoodData, replace
```

Sniffing Around - What's in the Data

So we loaded a dataset and we want to learn more about what's there. Here are a few commands worthy of notice:

- describe - Lists the variables names, labels and formats
- list - Lists the observations' matrix: each observation with its values for each of the variables.
- browse - Like list, but opens up a window and is more convenient
- tabulate - Reports a histogram of a variable or joint histogram of two variables.
- summarize - For each variable requested, reports the number of observations with non-missing values, the mean, standard deviation, and other summary statistics.

Examples and their Output

```
describe idcode year birth_yr
```

```
              storage  display     value
variable name   type   format      label      variable label
------------------------------------------------------------
idcode          int    %8.0g                 NLS ID
year            byte   %8.0g                 interview year
birth_yr        byte   %8.0g                 birth year
```

```
tab mothers_brothers fathers_brothers
```

```
mothers_br |                  fathers_brothers
    others |         0          1          2          3          4 |     Total
-----------+-------------------------------------------------------+----------
         0 |       132        138         74         24          7 |       375
         1 |       113        131         69         26          9 |       348
         2 |        72         87         37          7          2 |       205
         3 |        21         18         12          4          1 |        56
         4 |         7          5          2          2          0 |        16
-----------+-------------------------------------------------------+----------
     Total |       345        379        194         63         19 |     1,000
```

Examples and their Output

```
su south race age
```

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       south |     28526    .4095562    .4917605          0          1
        race |     28534    1.303392    .4822773          1          3
         age |     28510    29.04511    6.700584         14         46
```

```
su tenure, detail
```

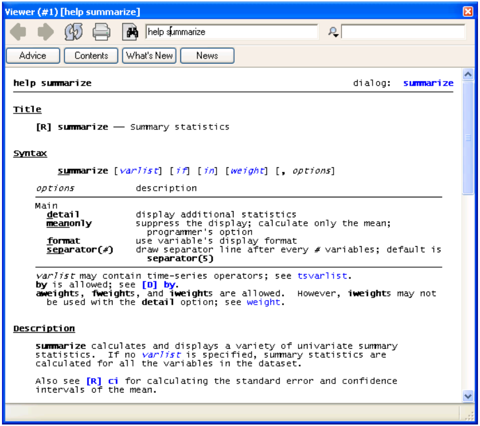
```
                    job tenure, in years
-------------------------------------------------------------
      Percentiles      Smallest
 1%            0              0
 5%     .0833333              0
10%     .1666667              0       Obs               28101
25%           .5              0       Sum of Wgt.       28101

50%     1.666667                      Mean           3.123836
                        Largest       Std. Dev.      3.751409
75%     4.166667       23.33333
90%     8.416667           24.5       Variance       14.07307
95%     11.41667          24.75       Skewness       1.939685
99%     16.91667       25.91667       Kurtosis       6.901501
```

Note: su is short for summarize

Commands’ Help Files

Each command should come with an accompanying help file. To learn more about additional options, other features, or to troubleshoot, the first place to look at is the help file:

```
help <command-name>
```



Commands' Help Files

Main parts of a usual help file (by the order I usually read them):

- Syntax - How to specify your command
- Description - Tells you what the command does generally
- Examples (at the bottom) - Shows you specific examples of how to run the command. Sometimes with an explanation.
- Options - The same command with different options can do totally different things. Skim through the options and look for the good ones.

Tips:

- Don't be afraid to experiment. Your data is saved on file, so you can always load it back if you made a mistake.
- Error messages looks scary, but don't let them fail you. READ them and try to understand them.
  - Remember: Errors don't mean that you are stupid, they mean that Stata is stupid.

Basic Data Manipulation - generate

To add a new variable we use the `gen` (short for `generate`) command:

```
gen <new-variable-name> = <expression> [if] [in]
```

For example:

```
gen four = 4
```

will create a variable (=column) that will contain the number 4 for all observations (=rows).

```
gen age_sq = age^2
```

will create a variable that will contain the square of the value in the age variable for the same row.

Conditions in Stata

Sometimes we want to apply a command only to some observations, not all.
We need to tell Stata what distinguishes those observations, so we construct a logical condition:

`male == 1` `age >= 21` `4 > 60`

- Stata evaluates the condition and turns it to 1, if the statement is true, or 0 if the statement is false.
- For example, since $(4 > 60)$ is not true, Stata will treat the expression $(4 > 60)$ as if it was 0.
- The other conditions involve variable names. They will be invoked as part of a command. Stata will apply the command only to observations for which the values inside the specified variables make the statement true.
  - For example, this is how we ask Stata to run `summarize` on females only:

```
su income if male == 0
```

Conditions in Stata

Note that some of the observations - those for which male == 1 - will not be processed by the `summarize` command.

We can combine multiple conditions with AND, OR and NOT operators:

```
(male == 1 & age >= 21) | (male == 0 & age >= 50)
```

will be true for males aged 21 and above or females aged 50 and above.
Adding the ! operator before a condition will negate it:

```
!(age >= 21)
```

will be true for people strictly younger than 21.

Conditions in Stata

Lastly, do not forget operators precedence:

$$6+5\times 2 \neq (6+5)\times 2$$

The same goes for | and & (like + and × respectively):

```
male == 1 & age > 21 | age < 10
```

will be true for males older than 21 and *all* children under 10.

```
male == 1 & (age >= 21 | age <= 10)
```

will be true for males only that are either older than 21 or younger than 10.

Note: missing values (.) are bigger than any value:
(`.  > 400000` is true)

For a complete list of logical as well as other operators, see
`help operator`

Back to Data Manipulation

`gen` can also take a condition. Observations for which the condition is false will have a missing value in the new variable:

```
gen age_sq_males = age^2 if male == 1
```

`age_sq_males` will contain the square of age for males and a missing value for females.

Note the difference between the assignment = and the comparison ==

One can also use the evaluation of a condition as the value to put into the new variable:

```
gen really_old = age > 22
```

Remember, the expression `age > 22` will be translated to either 1 or 0 according to whether age is bigger than 22 or not.

Back to Data Manipulation

Question: Which of the next three commands is best?

```
gen dropout = 1 if schooling < 12

gen dropout = schooling < 12

gen dropout = schooling < 12 if schooling != .
```

All commands will make those with `schooling < 12` have the value 1 in `dropout`. But what about the other ones?

- First line will assign missing values to all other observations
- Second line will assign zeroes to all other observations
- Third line will assign missing values to all observations that have a missing value in `schooling` and zeroes to the rest

So you will probably want to use the third line rather than the first two.

Meet `replace`, `gen`'s sister

Just like gen, but for existing variables instead of new ones, use replace (other statistical packages such as SAS don't even have this distinction between generating and replacing):

```
replace <variable-name> = <expression> [if] [in]
```

For example:

```
replace four = 5
```

will change the values in the variable `four` to now be 5.

```
gen     actual_price = discount_price if discount == 1
replace actual_price = full_price if discount == 0
```

will first create a variable that will contain the value from `discount_price` for all observations in which `discount` contains 1, then replace puts the value of `full_price` into `actual_price` if `discount` contains 0.

Do-files

Until now we used the command window to type in commands and run them one-by-one. This was working interactively. What if you have many commands to run?

A .do file is a text file that contains a batch of commands each written as a separate line. This way you can save your commands for:

- later review, improvement and additional work
- collaborating with your colleagues - they can continue what you started

Comments

In a do file you can also explain what you are doing by adding comments. Here are a few ways to write comments:

```
/* Multi-line comments can be written easily like this
   I can continue babbling on and on
   until I have nothing more to say about this program */

* One line comments that start at the beginning of the line
* can be written by putting a * at the beginning of the line

replace a = b    // One line comments that don't necessarily
                 // begin at the beginning of a line can be
                 // written by those double-slashes. Everything to
                 // the right of a double-slash is a comment.
```

Long lines in do-files

As you noticed, each Stata command takes one line. Once you hit the return, or enter, key, Stata runs the command. This is also true for do-files.
But what if you have a really long command?

```
su income mot_educ fat_educ school age agesq south bigcity tenure comm
```

You can break the line with `///`:

```
su income mot_educ fat_educ school age agesq south bigcity ///
 tenure commute kids_u5 kids_18 tot_kids siblings ///
 if male & professional
```

Another way is to write `#delimit;` at the beginning of the do file and then end each command with a semicolon (;):

```
#delimit;
su income mot_educ fat_educ school age agesq south bigcity
 tenure commute kids_u5 kids_18 tot_kids siblings
 if male & professional;
tab age;
```

Log files

One last file you can save your work to is your log file. Unlike the do file, a log file will also save the text output resulted by your commands.

Whatever appeared in the results (big black) window, from when a log file was opened until it was closed, will be saved to the requested log file.

A log file is used to see what your program have done. Unlike the do file that will be edited and improved by you, the log file is automatically created by your program.

```
// Open a log file
log using <log-file-name> [, append replace text]

/* ... */

// Close current log file
log close
```

Log files

- <u>replace</u> - overwrite the file on disk if it already exists
- <u>append</u> - add the output to the end of the existing file if one exists - otherwise open a new one
- <u>text</u> - save the output in text format. In some cases Stata's default is to save it in a text-like format of its own called SMCL.

Here's a tip:
If a log is already open (usually after the last run ended tragically with an error), opening a log will create yet another error

To solve that, add the following line right before the `log using` line:

```
cap log close
```

File Types Summary

	data files	do files	log files
What's in it?	Observations and variables	A batch of commands	Text output from your commands
How do you open it?	`use` command mainly	Any text file editor	`log` commands
What is it good for?	Saving your data	Saving sequence of actions on the data	Recording commands you ran and their output
File extensions	`.dta`	`.do`	`.log`

What is a Macro?

A Macro is a string (= a sequence of characters) that we name and can refer to.

One type of a macro is the local macro (local = can not be referred to outside the program):

```
// Define local
local <macro name> = <expression>

// Refer to local
[...] `<macro name>'
```

Note the back-quote and quote signs: ` is the character usually on the upper left corner of the main part of your keyboard (where ~ is). ' is the usual single-quote sign you're using.

Globals

```
// Define the global and assign an expression to it
global <macro name> = <expression>

// Refer to the global
[...] ${<macro name>}
```

Example:

```
global a = 4

// Refer to the global
di "Four is ${a}"   // will print: Four is 4
```

Strings and Macros

In Stata, we put string expressions between double-quotes. For example:

```
gen     girl = 1 if sex == "female"
replace girl = 0 if sex == "male"
```

If we don't put double quotes in a string expression, Stata will look for a variable with that name. We need to tell Stata the word is a value instead of part of the command. This is why we need the double quotes.

Same goes for macros:

```
local greeting = "Hello world!"
di "`greeting'"      // will be run as: di "Hello world!"

local mycommand = "tab"
`mycommand' age female   // will be run as: tab age female
```

First loop: `forvalues`

Loops are lines of code that can be run more than one time. Each time is called an **iteration** and in `for` loops there is also an index that is changing each iteration (the index is actually a local).

In the case of `forvalues`, the index is incremented each iteration:

```
// Define the loop
forvalues <index name> = <starting value>/<ending value> {
   // Commands to run each iteration
   // ... more commands ...
}
```

The loop will put <starting value> into <index name>, then run the commands until it reaches the closing }. Then it will go back, increase the value of <index name> by one and run the commands again, until it is done with the commands for the <ending value>.

First loop: forvalues

For example:

```
forvalues i = 1/3 {
   di "Iteration #`i'"
}
```

Will print:

```
Iteration #1
Iteration #2
Iteration #3
```

```
forvalues i = 7(7)21 {
   replace age = 0 in `i'
}
```

Will set the value of the variable age to 0 for observations 7, 14 and 21.

What is it good for? Part 1

Imagine you have three different specifications:

$$y_t = \beta_0 + \beta_1 x_t + \varepsilon_t$$

$$y_t = \beta_0 + \beta_1 x_t + \beta_2 age_t + \beta_3 agesq_t + \beta_4 educ_t + \eta_t$$

$$y_t = \beta_0 + \beta_1 x_t + \beta_2 age_t + \beta_3 agesq_t + \beta_4 educ_t + \beta_5 mo\_educ_t + \beta_6 fa\_educ_t + u_t$$

```
local spec1 ""
local spec2 "age agesq educ"
local spec3 "‘spec2’ mo_educ fa_educ"

forvalues i = 1/3 {
   reg y x ‘spec‘i’’
}
```

Still not convinced? You're right. This example, as it is now, is longer than just writing three lines of regressions. But hold on...

foreach

When you want to iterate on other lists - not just an arithmetic sequence of numbers - you will want to use `foreach`.

The simplest form to use `foreach` is:

```
foreach <index name> in <list separated by space> {
   // Commands to run each iteration
}
```

For example:

```
foreach i in 3 15 17 39 {
   di "I am number `i'"
}

foreach dep_var in income consumption health_score {
   reg `dep_var' educ age agesq
}
```

Even though we didn't put double-quotes on the values, since they are inside a foreach loop with the `in` word, Stata knows to treat them as values

foreach

If you want to loop over values that have space in them, use the double-quotes:

```
foreach fullname in "Roy Mill" "John Doe" Elvis Presley Madonna {
   di "Hello `fullname'"
}
```

Will print:

```
Hello Roy Mill
Hello John Doe
Hello Elvis
Hello Presley
Hello Madonna
```

`foreach` and variables lists

When you iterate over variables' names it's better to put **`of varlist`** instead of **`in`**:

```
foreach <index name> of varlist <varlist> {
   // Commands to run each iteration
}
```

This way:

- Stata will check that each element of the variables list is actually a variable (avoid typos)
- You will be able to use wildcards

```
foreach mother_vars of varlist mother_* {
   // This loop will go over all variables that begin with mother_
}
foreach setvar of varlist set?_score {
  // This loop will go over all variables that have one character
  // where the ? is. For example set1_score, set2_score, ...
  // (but not set14_score)
}
```

What is it good for? Part 2

Remember our three specifications? Now imagine we want to run them on three different samples: males, females and both. Here is one way to do that:

```
local spec1 ""
local spec2 "age agesq educ"
local spec3 "‘spec2’ mo_educ fa_educ"

foreach sampleCond in "if male == 1" "if male == 0" "" {
  forvalues i = 1/3 {
   reg y x ‘spec‘i’’ ‘sampleCond’
  }
}
```

What is it good for? Part 2

This loop is equivalent to running:

```
reg y x if male == 1
reg y x age agesq educ if male == 1
reg y x age agesq educ mo_educ fa_educ if male == 1
reg y x if male == 0
reg y x age agesq educ if male == 0
reg y x age agesq educ mo_educ fa_educ if male == 0
reg y x
reg y x age agesq educ
reg y x age agesq educ mo_educ fa_educ
```

Now imagine you want to change the standard errors to robust, or add another control variable to the second specification. How much work will you need for the loops version and how much for the this version? And wait until you will need to post the results to a table.

Getting values returned by commands

```
su union
```

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       union |     19238    .2344319    .4236542          0          1
```

```
return list
```

```
scalars:
                  r(N) =  19238
              r(sum_w) =  19238
               r(mean) =  .2344318536230377
                r(Var) =  .179482889232214
                 r(sd) =  .4236542095060711
                r(min) =  0
                r(max) =  1
                r(sum) =  4510
```

Getting values returned by commands

```
reg union age south c_city

ereturn list

scalars:
                  e(N) =  19226
               e(df_m) =  3
               e(df_r) =  19222
                  e(F) =  159.8965961359796
[...]
               e(ll_0) =  -10764.76183026799

macros:
            e(cmdline) : "regress union age south c_city"
              e(title) : "Linear regression"
                e(vce) : "ols"
[...]
          e(estat_cmd) : "regress_estat"

matrices:
                  e(b) :  1 x 4
                  e(V) :  4 x 4
```

Getting values returned by commands

```
// Show coefficients after reg
matrix list e(b)

e(b)[1,4]
           age       south      c_city       _cons
y1   .00242889  -.11547906   .06972916   .18224179


// Show coefficients' variance-covariance matrix
matrix covmat = e(V)
matrix list covmat

symmetric covmat[4,4]
               age       south      c_city       _cons
   age   2.399e-07
 south  -9.321e-08   .00003756
c_city   2.739e-07  -6.147e-07   .00004085
 _cons  -7.578e-06  -.00001244  -.00002226   .00025953
```

Extensions 1 - _variables

Besides the _b[] and _se[] we have other special variables that start with _:

- `_n` refers to the observation number:

```
sort school
gen     id_in_school = 1
replace id_in_school = id_in_school[_n-1] + 1 ///
                     if school == school[_n-1]
```

- `_cons` refers to the constant term in a regression - in the `_b[]` or `_se[]` context.
- `_N` contains the total number of observations in the dataset

The relevant help file is `help _variables`.

Extensions 3 - `while` and `if`

<u>`if`</u> - up until now we used `if` as an argument of commands, to let them know which observations to work on. `if` can also be used to control the flow of the program - especially inside loops.

```
if <condition> {
    // Commands
}
[else if <condition> {
    // Commands
}]
[else {
    // Commands
}]
```

<u>`while`</u> - in addition to `foreach` and `forvalues` sometimes we don't know in advance how many iterations we will need. We just need to loop as long as some condition holds (for example, as long as we haven't reached convergence).

Extensions 3 - `while` and `if`

But be careful with `while`s, because if the condition will not be satisfied, you will enter a never-ending loop.

Usually, it's preferable to use some maximum number of iterations, in case there is some probability the usual condition will not work:

```
local converged = 0
local iter = 0
local max_iter = 800
while (!`converged' & `iter' < `max_iter') {
    // Commands that do something and check whether convergence
    // was achieved. If convergence was achieved it does
    // local converged = 1

    local iter = `iter' + 1
}
```

Extensions 3 - `while` and `if`

But then, if we're already counting iterations, we might as well do it all with forvalues:

```
local converged = 0
local max_iter = 800
forvalues iter = 1/`max_iter' {
    // Commands that do something and check whether convergence
    // was achieved. If convergence was achieved it does
    // local converged = 1
    if `converged' {
        continue, break
    }
}
```

`continue`, without the `break` option, stops the execution of the *current* iteration and goes on to the next iteration. With the `break` option it exits the loop altogether.

egen

`egen` is a “super-command". It generates new variables and serves as an extension to the `generate` command.

Very useful in panel data and in any other hierarichal data:

- Student-level data with class-, school- and/or city-level variables.
- Any other individual level data with some observations grouped by some identifier.
  - But uses extend to non-group-related tasks too.

egen – Syntax

The syntax is generally pretty simple:

```
egen <new varname> = <function>(<expression>) [, ... by (<varlist>)]
```

Another way to do the same thing:

```
bysort <variables>: egen <new variable> = <function>(<expression>) [, ... ]
```

- The function we specify in <function> will determine what egen will do. Each function is like a different command even though they all begin with `egen`

We will now go over main functions.

“Vertical" `egen` functions – `mean()`

To create a variable containing the mean of another variable we can do:

```
summarize gpa
gen meangpa = r(mean)
```

But if you want to create a variable containing the mean of another **within the group** of each observation, it will be much harder without **egen**.

```
egen meangpainyear = mean(gpa), by(year)
```

Example: your dataset is such that you have both year and cohort and you want to get the GPAs demeaned of the cohort-year mean GPA (for the class of 2012 in year 2010):

```
egen mean_gpa_in_cohort_year = mean(gpa), by(year cohort)
gen gpa_demeaned = gpa - mean_gpa_in_cohort_year
drop mean_gpa_in_cohort_year
```

“Vertical" `egen` functions – `sum()`, `min()`, `max()`

A function that works the same way but gives you the sum instead of the mean is called...

```
egen total_tax_in_county_year = sum(tax), by(state county year)
```

And when you want the minimum or maximum within a group:

```
// For a dataset with children that can be grouped to families.
egen youngest_sibling_age = min(age), by(familyid)

// For a dataset of basketball statistics per team, player, game.
egen highest_score_player = max(points), by(playerid)
egen highest_score_team = max(points), by(teamid)
egen highest_score_player_team = max(points), by(playerid teamid)
```

Example: Transferring a variable to the [0,1] range

Using a within-group maximum and minimum

```
gen norm_wage_f = (wage - min_wage_f) /
                  (max_wage_f - min_wage_f)
```

The denominator will be 0 if all employees get the same wage

```
egen min_wage_f = min(wage),
     by(firm)
```

```
replace norm_wage_f = .5
   if norm_wage_f == .
```

```
egen max_wage_f = max(wage),
     by(firm)
```

employee	firm	wage	max_w…	min_w…	norm_wage…	(norm_wage… after replace)
…	1	…	…	…	…	…
5	2	10,000	15,000	7,000	.375	.375
6	2	7,000	15,000	7,000	0	0
7	2	15,000	15,000	7,000	1	1
8	3	10,000	10,000	10,000	.	.5
9	3	10,000	10,000	10,000	.	.5
10	3	10,000	10,000	10,000	.	.5
…	4	…	…	…	…	…

“Vertical" `egen` functions – `count()`

`count()` will put the number of **nonmissing** values in the variable.

```
egen studentsinclass = count(studentid), by(school grade class)
```

If you’re interested in counting the number of observations, regardless of missing values, try to count `_cons` or `_n`. Every observation has `_cons==1` and `_n` is the obs’ number.

```
egen studentsinclass = count(_n), by(school grade class)
```

“Horizontal" egen functions

We sometimes want to do the sum, mean, count, min and max across variables for each observation, rather than across observations for each variable.

```
egen hours = rowtotal(hoursday hoursnight)

// suppose each is a judge score
egen disagreement = rowsd(evaluation1 evaluation2 evaluation3)

// suppose each is dummy for attendance at day
egen full_attendance = rowmin(mon tue wed thu fri)

// suppose each has gas quality or missing value
egen sampled_pumps = rownonmiss(leaded unleaded premium)
```

Two reasons for preferring `egen rowtotal()` and `egen rowmean()` over the simple `gen` with the respective formula:

- `egen` ignores missing values. If you specify two or more variables and some of them are missing, the sum or mean will be calculated only for the nonmissing values.
- `egen` can get varlists – for example: `evaluation_*` or `mon-fri`.

`reshape`

Suppose you have observations in a two-dimensional dataset. For example, “panel" data with state and year. Alternatively, think about a survey per household with recurring questions for each of the household members.

Some of the variables – X_i are common to all observations of the same group i (state area in the panel data, household income in the survey). Others – X_{ij} – are changing with members j within the group i.

Two ways to structure a dataset matrix:

Form	Each obs is	Member-level variables (X_{ij})
Wide	Group (i)	Appear $\max(j)$ times
Long	Group-member (i, j)	Appear just once

reshape

Wide form:

i	X_{ij}			
fam_id	kid_educ1	kid_educ2	kid_educ3	kid_educ4
A	8	6	.	.
B	3	.	.	.
C	14	10	8	6

Long form:

i	j	X_{ij}
fam_id	kid_id	kid_educ
A	1	8
A	2	6
B	1	3
C	1	14
C	2	10
C	3	8
C	4	6

reshape

Panel commands usually work with long forms. Wide forms are ugly and inefficient. However, you sometimes get your data in wide form. Especially if it's a questionnaire dataset

`reshape` allows you to go from wide to long form or the other way around. The simple syntax:

```
reshape <long|wide> <stubnames>, i(<group-identifying-vars>) [j(<member-identif
```

Where `stubname` is the part of the variable that is not changing between members. In our case: `kid_educ`.

Examples:

```
// From wide to long
reshape long kideduc, i(famid) j(kidid)

// From long to wide
reshape wide kideduc, i(famid) j(kidid)
```