Data Tricks

What's on the menu

- `egen` – especially for calculating group statistics at the individual level dataset.
- `collapse` – for aggregating into group-level datasets.
- `reshape` – to turn panel-data datasets from long format to wide format and vice versa.
- `[_n]` tricks – can help with relational stuff.

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)
```

“Vertical" `egen` functions – populating values to other observations in the group

`max()`, `min()` and other vertical functions allow us to populate values from individual observations to the group.

Say you have school-student dataset with parents' schooling and you want to know how many children in a school have fathers who dropped out of school.

First you need to count students for which the condition applies. One way to do it is this:

```
egen f_dropout_kids_only = count(_n) if feduc < 12, by(school)
```

But this will put missing values in the observations for which `f_educ >= 12`. So to populate the value to them we run:

```
egen f_dropout_kids = max(fdropoutkidsonly), by(school)
```

In this case, `min()` and `mean()` will also work the same.

Example: Using max to populate values

When there is a single unique nonmissing value but missing values in other observations

student_id	school	f_educ	…	f_dropout_ kids_only	f_dropout_ kids
1	1	10	…	3	3
2	1	16	…	.	3
3	1	11	…	3	3
4	1	11	…	3	3
5	1	14	…	.	3
6	1	16	…	.	3
7	2	20	…	.	1
8	2	18	…	.	1
9	2	16	…	.	1
10	2	16	…	.	1
11	2	14	…	.	1
12	2	10	…	1	1
13	2	15	…	.	1

“Vertical" `egen` functions – flagging problematic observations

A frequent use of populating values is when flagging inconsistencies in the dataset.

```
egen oldest_sibling_byear = min(birthyear), by(mother_id)
gen problematic = oldest_sibling_byear - byear > 30 ///
           if oldest_sibling_byear - byear != .
browse mother_id child_id birthyear if problematic
```

This will show you only the observations of those who are younger than the oldest sibling by more than 30 years. If you want to browse the all observations in families in which a problematic obs was found, you can populate the problematic flag to the rest of the family:

```
egen problematic_in_fam = max(problematic), by(mother_id)
browse mother_id child_id birthyear if problematic_in_fam
```

“Vertical" egen functions – tag()

`tag(<variables>)` takes a group of observations with the same values in `<variables>`, puts 1 in the first and 0 to all the rest.

```
egen distinct_name = tag(name)
su distinct_name
di "There are " r(sum) " unique names in dataset"
```

A more useful use is for counting unique values within group, like the number of X in each Y using a Z-based dataset:

```
// Number of teachers in a school, using a student-based dataset
egen tag_teacher_school = tag(teacher_id school_id)
egen teachers_in_school = sum(tag_teacher_school), by(school_id)

// Number of brands sold in a store in a day, using a transaction-based dataset
egen tag_brand_store_week = tag(brand_code store_id week)
egen brands_in_store_week = sum(tag_brand_store_week), by(store_id week)
```

The following slide will show graphically how to get the number of weeks “in" a gas station, using a station-week-fuel dataset:

Example: `egen tag()`

When there is a single unique nonmissing value but missing values in other observations

week	station	brand	…	station_week_tag	weeks_of_station
1	1	leaded	…	1	50
1	1	unleaded	…	0	50
2	1	leaded	…	1	50
2	1	unleaded	…	0	50
…	…	…	…	…	…
50	1	leaded	…	1	50
50	1	unleaded	…	0	50
2	2	leaded	…	1	41
2	2	unleaded	…	0	41
2	2	premium	…	0	41
5	2	leaded	…	1	41
5	2	unleaded	…	0	41
5	2	premium	…	0	41
…	…	…	…	…	…

Other "Vertical" `egen` functions

`group(<variables>)`
creates codes corresponding to distinct values of a variables. Just like tag, but instead of 1's and 0's within a group, all observations in a group get the same number (ascending with every group).

`rank(<variable>) [, by(<group variables>)]` –
puts the rank of the value of <variable> within the <group variable>'s values.

More on those in `help egen`

“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`.

`collapse`

- Vertical `egen` functions will calculate group-level statistics, but the dataset will stay at the individual level.
  - All individuals will share the same group-level statistic
- `collapse` allows easy aggregation from ij-level data to j-level data.
- Allows various statistics on the ij level for each observation at the j level.
- Basic syntax:

```
collapse [(statistic)] <varlist> [(statistic) <varlist> ...], by(group id)
```

- It's a good idea to `preserve` the data before collapsing if you want to go back to the individual level data (using `restore`)

`collapse` Example

- Suppose you have a dataset on federal contracts in which every row is a contract.
- You want to aggregate the total value of the contract by congressional district and year.

```
collapse (sum) value, by(district year)
```

- Suppose you also want the share of contracts given to businesses owned by women and minorities

```
collapse womanowned minorityowned (sum) value, by(district year)
```

  - The first statistic by default is `(mean)`

- Suppose you have a categorical variable containing the contract type. You want the share of contracts in each type (regardless of value)

```
xi i.contract_type, prefix(_CT) noomit
collapse _CT* (sum) value, by(district year)
```

`collapse` and Variable Names in the New Dataset

- Sometimes you want more than one statistic of the same variable.
  - You want both the mean and the standard deviation, or the mean and the median.
- By default, collapse will give the same variable name in the aggregated dataset.
- But if there is more than one statistic for a variable, can't have the same name for two variables.

```
collapse [(statistic)] newvarname=<orig_varname> [...] [(statistic) ...],
```

- Suppose we want both the total value and the average value per contract.

```
collapse (sum) total_usd=value (mean) avg_usd=value, by(district year)
```

`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)
```

reshape – last remarks

If the suffix that represents the member is not numeric (if we had kid_educA, kid_educB, ...), you need to add the `string` option.

When going from wide to long, the group identifying variable (i) should have only unique values (no duplicate ids).

The recurring j code must be a suffix. It can not be in the beginning or the middle of variable.

There are more advanced options. Feel free to experiment and check `help reshape`.

Referring to absolute and relative observation values

When one puts `[#]` right after a variable name, Stata interprets it as if one is referring to the value of the variable in the #th observation.

```
di "Revenue at obs 50 is " price[50] * quantity[50]
gen same_R_as_50 = (price*quantity == price[50]*quantity[50])
```

Using absolute observation numbers is not very common, unless your dataset has a matrix flavor where obs numbers have meaning.

It is tempting to use it when you want to replace a specific value in some observation, but you can't use the `[#]` to refer to an observation you want to *assign* values to:

```
replace price[50] = price[49] // ERROR! Bad!
replace price     = price[49] in 50 // That's the way I like it.
```

Referring to absolute and relative observation values

A more frequent use is referal to relative observation values. For example, taking the value of the previous observation.

```
sort famid birthyear
gen years_since_last_sib = birthyear - birthyear[_n-1] ///
        if fam_id == fam_id[_n-1]

gen cumulative_tax = tax
replace cumulative_tax = cumulative_tax[_n-1] + tax in 2/l
```

- _n is the current observation's number.
- Specifying `[_n]` after a variable name is equivalent to not specifying it. You need it for other observations next to this one: `` [_n+`k'] ``

Remember:

```
replace price[_n] = price[_n-1] // ERROR! Not good.
replace price     = price[_n-1] // This is the proper way
```