Title

stata.com

egen — Extensions to generate

Description

`egen` creates a new variable of the optionally specified storage type equal to the given function based on arguments of that function. The functions are specifically written for `egen`, as documented below or as written by users.

Quick start

Generate `newv1` for distinct groups of `v1` and `v2`, and create and apply value label `mylabel`

```
egen newv1 = group(v1 v2), label(mylabel)
```

Generate `newv2` equal to the minimum of `v1`, `v2`, and `v3` for each observation

```
egen newv2 = rowmin(v1 v2 v3)
```

Generate `newv3` equal to the overall sum of `v1`

```
egen newv3 = total(v1)
```

As above, but calculate total within each level of `catvar`

```
egen newv3 = total(v1), by(catvar)
```

Generate `newv4` equal to the number of nonmissing numeric values across `v1`, `v2`, and `v3` for each observation

```
egen newv4 = rownonmiss(v1 v2 v3)
```

As above, but allow string values

```
egen newv4 = rownonmiss(v1 v2 v3), strok
```

Generate `newv5` as the concatenation of numeric `v1` and string `v4` separated by a space

```
egen newv5 = concat(v1 v4), punct(" ")
```

Menu

Data > Create or change data > Create new variable (extended)

Syntax

```
egen [type] newvar = fcn(arguments) [if] [in] [ , options]
```

by is allowed with some of the egen functions, as noted below.

Depending on *fcn*, *arguments* refers to an expression, *varlist*, or *numlist*, and the *options* are also *fcn* dependent. *fcn* and its dependencies are listed below.

anycount(*varlist*), values(*integer numlist*)
may not be combined with by. It returns the number of variables in *varlist* for which values are equal to any integer value in a supplied *numlist*. Values for any observations excluded by either if or in are set to 0 (not missing). Also see anyvalue(*varname*) and anymatch(*varlist*).

anymatch(*varlist*), values(*integer numlist*)
may not be combined with by. It is 1 if any variable in *varlist* is equal to any integer value in a supplied *numlist* and 0 otherwise. Values for any observations excluded by either if or in are set to 0 (not missing). Also see anyvalue(*varname*) and anycount(*varlist*).

anyvalue(*varname*) , values(*integer numlist*)
may not be combined with by. It takes the value of *varname* if *varname* is equal to any integer value in a supplied *numlist* and is missing otherwise. Also see anymatch(*varlist*) and anycount(*varlist*).

concat(*varlist*) [, format(*%fmt*) decode maxlength(#) punct(*pchars*)]
may not be combined with by. It concatenates *varlist* to produce a string variable. Values of string variables are unchanged. Values of numeric variables are converted to string, as is, or are converted using a numeric format under the format(*%fmt*) option or decoded under the decode option, in which case maxlength() may also be used to control the maximum label length used. By default, variables are added end to end: punct(*pchars*) may be used to specify punctuation, such as a space, punct(" "), or a comma, punct(,).

count(*exp*) (allows by *varlist*:)
creates a constant (within *varlist*) containing the number of nonmissing observations of *exp*. Also see rownonmiss() and rowmiss().

cut(*varname*), { at(*numlist*) | group(#) } [icodes label]
may not be combined with by. Either at() or group() must be specified. When at() is specified, it creates a new categorical variable coded with the left-hand ends of the grouping intervals specified in the at() option. When group() is specified, groups of roughly equal frequencies are created.

at(*numlist*) with *numlist* in ascending order supplies the breaks for the groups. *newvar* is set to missing for observations with *varname* less than the first number specified in at() and for observations with *varname* greater than or equal to the last number specified in at().

group(#) specifies the number of equal-frequency grouping intervals when breaks are not specified. Specifying this option automatically invokes icodes.

icodes requests that the codes 0, 1, 2, etc., be used in place of the left-hand ends of the intervals.

label requests that the integer-coded values of the grouped variable be labeled with the left-hand ends of the grouping intervals. Specifying this option automatically invokes icodes.

diff(*varlist*)
may not be combined with by. It creates an indicator variable equal to 1 if the variables in *varlist* are not equal and 0 otherwise.

`ends(`*strvar*`)` [`, punct(`*pchars*`) trim` [`head` | `last` | `tail`]]
may not be combined with `by`. It gives the first "word" or head (with the `head` option), the last "word" (with the `last` option), or the remainder or tail (with the `tail` option) from string variable *strvar*.

`head`, `last`, and `tail` are determined by the occurrence of *pchars*, which is by default one space (" ").

The head is whatever precedes the first occurrence of *pchars*, or the whole of the string if it does not occur. For example, the head of "frog toad" is "frog" and that of "frog" is "frog". With `punct(,)`, the head of "frog,toad" is "frog".

The last word is whatever follows the last occurrence of *pchars* or is the whole of the string if a space does not occur. The last word of "frog toad newt" is "newt" and that of "frog" is "frog". With `punct(,)`, the last word of "frog,toad" is "toad".

The remainder or tail is whatever follows the first occurrence of *pchars*, which will be the empty string `""` if *pchars* does not occur. The tail of "frog toad newt" is "toad newt" and that of "frog" is `""`. With `punct(,)`, the tail of "frog,toad" is "toad".

The `trim` option trims any leading or trailing spaces.

`fill(`*numlist*`)`
may not be combined with `by`. It creates a variable of ascending or descending numbers or complex repeating patterns. *numlist* must contain at least two numbers and may be specified using standard *numlist* notation; see **[U] 11.1.8 numlist**. `if` and `in` are not allowed with `fill()`.

`group(`*varlist*`)` [`, missing autotype label` [`(`*lblname* [`, replace truncate(#)`] `)`]]
may not be combined with `by`. It creates one variable taking on values 1, 2, ... for the groups formed by *varlist*. *varlist* may contain numeric variables, string variables, or a combination of the two. The order of the groups is that of the sort order of *varlist*.

`missing` indicates that missing values in *varlist* (either `.` or `""`) are to be treated like any other value when assigning groups. By default, any observation with a missing value is assigned to the group with *newvar* equal to missing (`.`).

`autotype` specifies that *newvar* be the smallest *type* possible to hold the integers generated. The resulting *type* will be `byte`, `int`, `long`, or `double`.

`label` or `label(`*lblname*`)` creates a value label for *newvar*. The integers in *newvar* are labeled with the values of *varlist* or their value labels, if they exist. `label(`*lblname*`)` specifies *lblname* as the name of the value label. If `label` alone is specified, the name of the value label is *newvar*. `label(..., replace)` allows an existing value label to be redefined. `label(..., truncate(#))` truncates the values contributed to the label from each variable in *varlist* to the length specified by the integer argument #.

`iqr(`*exp*`)` [`, autotype`] (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the interquartile range of *exp*. `autotype` specifies that *newvar* be the smallest *type* possible to hold the result. The resulting *type* will be `byte`, `int`, `long`, or `double`. Also see `pctile()`.

`kurt(`*exp*`)` (allows `by` *varlist*`:`)
returns the kurtosis (within *varlist*) of *exp*.

`mad(`*exp*`)` (allows `by` *varlist*`:`)
returns the median absolute deviation from the median (within *varlist*) of *exp*.

`max(`*exp*`)` [`, missing`] (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the maximum value of *exp*. `missing` indicates that missing values be treated like other values.

`mdev(`*exp*`)` (allows `by` *varlist*`:`)
returns the mean absolute deviation from the mean (within *varlist*) of *exp*.

`mean(`*exp*`)` (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the mean of *exp*.

`median(`*exp*`)` [`, autotype`] (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the median of *exp*. `autotype` specifies that *newvar* be the smallest *type* possible to hold the result. The resulting *type* will be `byte`, `int`, `long`, or `double`. Also see `pctile()`.

`min(`*exp*`)` [`, missing`] (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the minimum value of *exp*. `missing` indicates that missing values be treated like other values.

`mode(`*varname*`)` [`, minmode maxmode nummode(`*integer*`) missing`] (allows `by` *varlist*`:`)
produces the mode (within *varlist*) for *varname*, which may be numeric or string. The mode is the value occurring most frequently. If two or more modes exist or if *varname* contains all missing values, the mode produced will be a missing value. To avoid this, the `minmode`, `maxmode`, or `nummode()` option may be used to specify choices for selecting among the multiple modes. `minmode` returns the lowest value, and `maxmode` returns the highest value. `nummode(#)` returns the #th mode, counting from the lowest up. `missing` indicates that missing values be treated like other values.

`pc(`*exp*`)` [`, prop`] (allows `by` *varlist*`:`)
returns *exp* (within *varlist*) scaled to be a percentage of the total, between 0 and 100. The `prop` option returns *exp* scaled to be a proportion of the total, between 0 and 1.

`pctile(`*exp*`)` [`, p(#) autotype`] (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the #th percentile of *exp*. If `p(#)` is not specified, 50 is assumed, meaning medians. `autotype` specifies that *newvar* be the smallest *type* possible to hold the result. The resulting *type* will be `byte`, `int`, `long`, or `double`. Also see `median()`.

`rank(`*exp*`)` [`, field` | `track` | `unique`] (allows `by` *varlist*`:`)
creates ranks (within *varlist*) of *exp*; by default, equal observations are assigned the average rank. The `field` option calculates the field rank of *exp*: the highest value is ranked 1, and there is no correction for ties. That is, the field rank is 1 + the number of values that are higher. The `track` option calculates the track rank of *exp*: the lowest value is ranked 1, and there is no correction for ties. That is, the track rank is 1 + the number of values that are lower. The `unique` option calculates the unique rank of *exp*: values are ranked 1, ..., #, and values and ties are broken arbitrarily. Two values that are tied for second are ranked 2 and 3.

`rowfirst(`*varlist*`)`
may not be combined with `by`. It gives the first nonmissing value in *varlist* for each observation (row). If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation.

`rowlast(`*varlist*`)`
may not be combined with `by`. It gives the last nonmissing value in *varlist* for each observation (row). If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation.

`rowmax(`*varlist*`)`
may not be combined with `by`. It gives the maximum value (ignoring missing values) in *varlist* for each observation (row). If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation.

`rowmean(`*varlist*`)`
may not be combined with `by`. It creates the (row) means of the variables in *varlist*, ignoring missing values. For example, if three variables are specified and, in some observations, one of the variables is missing, in those observations *newvar* will contain the mean of the two variables that do exist. Other observations will contain the mean of all three variables. If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation.

`rowmedian(`*varlist*`)`
may not be combined with `by`. It gives the (row) median of the variables in *varlist*, ignoring missing values. If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation. Also see `rowpctile()`.

`rowmin(`*varlist*`)`
may not be combined with `by`. It gives the minimum value in *varlist* for each observation (row). If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation.

`rowmiss(`*varlist*`)`
may not be combined with `by`. It gives the number of missing values in *varlist* for each observation (row).

`rownonmiss(`*varlist*`)` [`, strok`]
may not be combined with `by`. It gives the number of nonmissing values in *varlist* for each observation (row).

String variables may not be specified unless the `strok` option is also specified. When `strok` is specified, *varlist* may contain a mixture of string and numeric variables.

`rowpctile(`*varlist*`)` [`, p(`#`)`]
may not be combined with `by`. It gives the #th percentile of the variables in *varlist*, ignoring missing values. If `p()` is not specified, `p(50)` is assumed, meaning medians. If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation. Also see `rowmedian()`.

`rowsd(`*varlist*`)`
may not be combined with `by`. It creates the (row) standard deviations of the variables in *varlist*, ignoring missing values. If all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation.

`rowtotal(`*varlist*`)` [`, missing`]
may not be combined with `by`. It creates the (row) sum of the variables in *varlist*, treating missing values as 0. If `missing` is specified and all values in *varlist* are missing for an observation, *newvar* is set to missing for that observation.

`sd(`*exp*`)` (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the standard deviation of *exp*. Also see `mean()`.

`seq()` [`, from(`#`) to(`#`) block(`#`)`] (allows `by` *varlist*`:`)
returns integer sequences. Values start from `from()` (default 1) and increase to `to()` (the default is the maximum number of values) in blocks (default size 1). If `to()` is less than the maximum number, sequences restart at `from()`. Numbering may also be separate within groups defined by *varlist* or decreasing if `to()` is less than `from()`. Sequences depend on the sort order of observations, following three rules: 1) observations excluded by `if` or `in` are not

counted; 2) observations are sorted by *varlist*, if specified; and 3) otherwise, the order is that when called. No *arguments* are specified.

`skew(`*exp*`)` (allows `by` *varlist*`:`)
returns the skewness (within *varlist*) of *exp*.

`std(`*exp*`)` [`, mean(#) sd(#)`] (allows `by` *varlist*`:`)
creates the standardized values (within *varlist*) of *exp*. The options specify the desired mean and standard deviation. The default is `mean(0)` and `sd(1)`, producing a variable with mean 0 and standard deviation 1 (within each group defined by *varlist*).

`tag(`*varlist*`)` [`, missing`]
may not be combined with `by`. It tags just one observation in each distinct group defined by *varlist*. When all observations in a group have the same value for a summary variable calculated for the group, it will be sufficient to use just one value for many purposes. The result will be 1 if the observation is tagged and never missing, and 0 otherwise. Values for any observations excluded by either `if` or `in` are set to 0 (not missing). Hence, if `tag` is the variable produced by `egen tag = tag(`*varlist*`)`, the idiom `if tag` is always safe. `missing` specifies that missing values of *varlist* may be included.

`total(`*exp*`)` [`, missing`] (allows `by` *varlist*`:`)
creates a constant (within *varlist*) containing the sum of *exp* treating missing as 0. If `missing` is specified and all values in *exp* are missing, *newvar* is set to missing. Also see `mean()`.

Remarks and examples

stata.com

Remarks are presented under the following headings:

See Mitchell (2020) for numerous examples using `egen`.

Summary statistics

The functions `count()`, `iqr()`, `kurt()`, `mad()`, `max()`, `mdev()`, `mean()`, `median()`, `min()`, `mode()`, `pc()`, `pctile()`, `sd()`, `skew()`, and `total()` create variables containing summary statistics. These functions take a `by ...:` prefix and, if specified, calculate the summary statistics within each by-group.

▷ Example 1: Without the by prefix

Without the `by` prefix, the result produced by these functions is a constant for every observation in the data. For instance, we have data on cholesterol levels (`chol`) and wish to have a variable that, for each patient, records the deviation from the average across all patients:

```
. use https://www.stata-press.com/data/r17/egenxmpl
. egen avg = mean(chol)
. generate deviation = chol - avg
```

◁

▷ Example 2: With the by prefix

These functions are most useful when the by prefix is specified. For instance, assume that our dataset includes dcode, a hospital–patient diagnostic code, and los, the number of days that the patient remained in the hospital. We wish to obtain the deviation in length of stay from the median for all patients having the same diagnostic code:

```
. use https://www.stata-press.com/data/r17/egenxmpl2, clear
. by dcode, sort: egen medstay = median(los)
. generate deltalos = los - medstay
```

◁

▷ Example 3: sum() function and egen total()

Distinguish carefully between Stata's sum() function and egen's total() function. Stata's sum() function creates the running sum, whereas egen's total() function creates a constant equal to the overall sum, for example,

```
. clear
. set obs 5
Number of observations (_N) was 0, now 5.
. generate a = _n
. generate sum1 = sum(a)
. egen sum2 = total(a)
. list
```

	a	sum1	sum2
1.	1	1	15
2.	2	3	15
3.	3	6	15
4.	4	10	15
5.	5	15	15

◁

Definitions of egen summary functions

The definitions and formulas used by egen summary functions are the same as those used by summarize; see [R] **summarize**. For comparison with summarize, mean() and sd() correspond to the mean and standard deviation. total() is the numerator of the mean, and count() is its denominator. min() and max() correspond to the minimum and maximum. median()—or, equally well, pctile() with p(50)—is the median. pctile() with p(5) refers to the 5th percentile, and so on. iqr() is the difference between the 75th and 25th percentiles.

The mode is the most common value of a dataset, whether it contains numeric or string variables. It is perhaps most useful for categorical variables (whether defined by integers or strings) or for other integer-valued values, but mode() can be applied to variables of any type. Nevertheless, the modes of continuous (or nearly continuous) variables are perhaps better estimated either from inspection of a graph of a frequency distribution or from the results of some density estimation (see [R] **kdensity**).

Missing values need special attention. `egen` *newvar* `=` `mode(`*varname*`)` calculates the mode of all nonmissing observations, and the variable *newvar* containing the mode is filled in for all observations, even those for which *varname* is missing (except for observations excluded using an `if` or `in` statement). This allows use of `mode()` as a simple way to impute categorical variables.

Missing values are by default excluded from the determination of modes (whether missing is defined by the period [`.`] or extended missing values [`.a`, `.b`, ..., `.z`] for numeric variables or the empty string [`""`] for string variables). However, missing may be the most common value in a variable, and you want `mode()` to report this value as the mode. To include missing values as possible values for the mode, use the `missing` option. See *Missing values* below for more on missing values.

`mad()` and `mdev()` produce alternative measures of spread. The median absolute deviation from the median and even the mean deviation will both be more resistant than the standard deviation to heavy tails or outliers, in particular from distributions with heavier tails than the normal or Gaussian. The first measure was named the MAD by Andrews et al. (1972) but was already known to K. F. Gauss in 1816, according to Hampel et al. (1986). For more historical and statistical details, see David (1998) and Wilcox (2003, 72–73).

Missing values

Missing values in the argument to `egen` functions (typically, *varname*, an expression, or *varlist*) are generally handled in one of three ways. Functions that calculate a single statistic for *varname* or an expression (for example, `mean()` and `total()`) fill in the result for all observations, including those for which *varname* or the expression is missing.

Functions that calculate results that potentially differ observation by observation (for example, `group()` and `rank()`) generally generate missing values for the result for observations where *varname* or the expression is missing.

Functions that take *varlist* (for example, `rowmean()`) generally generate a missing value for the result only when every variable in *varlist* is missing for that observation.

▷ Example 4: How missing values are handled

Here's an example of how `mean()` handles missing values.

```
. use https://www.stata-press.com/data/r17/egenxmpl1, clear
. egen y = mean(x)
. list x y
```

	x	y
1.	0	3
2.	5	3
3.	2	3
4.	5	3
5.	3	3
6.	.	3
7.	.a	3

The result `y` is filled in for all observations, including the 6th and 7th observations where `x` is missing. If you do not want this behavior, you can explicitly exclude missing values using an `if` statement.

```
. egen z = mean(x) if !missing(x)
(2 missing values generated)
. list x z
```

	x	z
1.	0	3
2.	5	3
3.	2	3
4.	5	3
5.	3	3
6.	.	.
7.	.a	.

Other functions, such as `group()`, by default exclude missing values. If you want to treat missing values just like other values and let them be part of the enumerated groups as well, use the `missing` option.

```
. egen g1 = group(x)
(2 missing values generated)
. egen g2 = group(x), missing
. list x g1 g2
```

	x	g1	g2
1.	0	1	1
2.	5	4	4
3.	2	2	2
4.	5	4	4
5.	3	3	3
6.	.	.	5
7.	.a	.	6

With the `missing` option, the missing values "`.`" and "`.a`" are placed in two distinct groups, the 5th and 6th groups, in the result `g2`.

Here's an example of how `rowmean()` and `rowtotal()` handle missing values.

```
. egen m = rowmean(x1 x2 x3 x4)
(1 missing value generated)
. egen t1 = rowtotal(x1 x2 x3 x4)
. egen t2 = rowtotal(x1 x2 x3 x4), missing
(1 missing value generated)
. list x1 x2 x3 x4 m t1 t2
```

	x1	x2	x3	x4	m	t1	t2
1.	2	6	4	8	5	20	20
2.	9	.	0	3	4	12	12
3.	.	.a	.b	2	2	2	2
4.	.	.a	3	6	4.5	9	9
5.	4	5	5	2	4	16	16
6.	7	8	4	5	6	24	24
7.	.b	.a	.	.	.	0	.

rowmean() uses all the nonmissing values to calculate the mean of a row, ignoring any missing values. In the first row, all four variables are nonmissing, so the result is the mean of these four values. In the second row, three variables are nonmissing, and the result is the mean of these three values. In the third row, only one variable is nonmissing, and the result is simply the mean of this one value, that is, the value itself.

rowtotal() is similar to rowmean(), except that by default the total is 0 when all four variables are missing. See the 7th observation in this example. The result t1 is 0 in this case. If you want rowtotal() to behave like rowmean(), use the missing option. The result t2 is produced with this option, and you can see it is missing for the 7th observation, just like the rowmean() result.

Several egen functions have a missing option. See *Syntax* for the description of what missing does with each function that has this option—or better yet create a simple example, and run the function with and without the missing option.

◁

Generating patterns

To create a sequence of numbers, simply "show" the fill() function how the sequence should look. It must be a linear progression to produce the expected results. Stata does not understand geometric progressions. To produce repeating patterns, you present fill() with the pattern twice in the *numlist*.

▷ Example 5: Sequences produced by fill()

Here are some examples of ascending and descending sequences produced by fill():

```
. clear
. set obs 12
Number of observations (_N) was 0, now 12.
. egen i = fill(1 2)
. egen w = fill(100 99)
. egen x = fill(22 17)
. egen y = fill(1 1 2 2)
. egen z = fill(8 8 8 7 7 7)
. list, sep(4)
```

	i	w	x	y	z
1.	1	100	22	1	8
2.	2	99	17	1	8
3.	3	98	12	2	8
4.	4	97	7	2	7
5.	5	96	2	3	7
6.	6	95	-3	3	7
7.	7	94	-8	4	6
8.	8	93	-13	4	6
9.	9	92	-18	5	6
10.	10	91	-23	5	5
11.	11	90	-28	6	5
12.	12	89	-33	6	5

◁

Example 6: Patterns produced by fill()

Here are examples of patterns produced by `fill()`:

```
. clear
. set obs 12
Number of observations (_N) was 0, now 12.
. egen a = fill(0 0 1 0 0 1)
. egen b = fill(1 3 8 1 3 8)
. egen c = fill(-3(3)6 -3(3)6)
. egen d = fill(10 20 to 50   10 20 to 50)
. list, sep(4)
```

	a	b	c	d
1.	0	1	-3	10
2.	0	3	0	20
3.	1	8	3	30
4.	0	1	6	40
5.	0	3	-3	50
6.	1	8	0	10
7.	0	1	3	20
8.	0	3	6	30
9.	1	8	-3	40
10.	0	1	0	50
11.	0	3	3	10
12.	1	8	6	20

Example 7: seq()

`seq()` creates a new variable containing one or more sequences of integers. It is useful mainly for quickly creating observation identifiers or automatically numbering levels of factors or categorical variables.

```
. clear
. set obs 12
```

In the simplest case,

```
. egen a = seq()
```

is just equivalent to the common idiom

```
. generate a = _n
```

`a` may also be obtained from

```
. range a 1 _N
```

(the actual value of `_N` may also be used).

In more complicated cases, `seq()` with option calls is equivalent to calls to the versatile functions `int` and `mod`.

```
. egen b = seq(), b(2)
```

produces integers in blocks of 2, whereas

```
. egen c = seq(), t(6)
```

restarts the sequence after 6 is reached.

```
. egen d = seq(), f(10) t(12)
```

shows that sequences may start with integers other than 1, and

```
. egen e = seq(), f(3) t(1)
```

shows that they may decrease.

The results of these commands are shown by

```
. list, sep(4)
```

	a	b	c	d	e
1.	1	1	1	10	3
2.	2	1	2	11	2
3.	3	2	3	12	1
4.	4	2	4	10	3
5.	5	3	5	11	2
6.	6	3	6	12	1
7.	7	4	1	10	3
8.	8	4	2	11	2
9.	9	5	3	12	1
10.	10	5	4	10	3
11.	11	6	5	11	2
12.	12	6	6	12	1

All of these sequences could have been generated in one line with `generate` and with the use of the `int` and `mod` functions. The variables `b` through `e` are obtained with

```
. gen b = 1 + int((_n - 1)/2)
. gen c = 1 + mod(_n - 1, 6)
. gen d = 10 + mod(_n - 1, 3)
. gen e = 3 - mod(_n - 1, 3)
```

Nevertheless, `seq()` may save users from puzzling out such solutions or from typing in the needed values.

In general, the sequences produced depend on the sort order of observations, following three rules:

1. observations excluded by `if` or `in` are not counted;
2. observations are sorted by *varlist*, if specified; and
3. otherwise, the order is that specified when `seq()` is called.

◁

The `fill()` and `seq()` functions are alternatives. In essence, `fill()` requires a minimal example that indicates the kind of sequence required, whereas `seq()` requires that the rule be specified through options. There are sequences that `fill()` can produce that `seq()` cannot, and vice versa. `fill()` cannot be combined with `if` or `in`, in contrast to `seq()`, which can.

Marking differences among variables

▷ Example 8: diff()

We have three measures of respondents' income obtained from different sources. We wish to create the variable `differ` equal to 1 for disagreements:

```
. use https://www.stata-press.com/data/r17/egenxmpl3, clear
. egen byte differ = diff(inc*)
. list if differ==1
```

	inc1	inc2	inc3	id	differ
10.	42,491	41,491	41,491	110	1
11.	26,075	25,075	25,075	111	1
12.	26,283	25,283	25,283	112	1
78.	41,780	41,780	41,880	178	1
100.	25,687	26,687	25,687	200	1
101.	25,359	26,359	25,359	201	1
102.	25,969	26,969	25,969	202	1
103.	25,339	26,339	25,339	203	1
104.	25,296	26,296	25,296	204	1
105.	41,800	41,000	41,000	205	1
134.	26,233	26,233	26,133	234	1

Rather than typing `diff(inc*)`, we could have typed `diff(inc1 inc2 inc3)`.

◁

Ranks

▷ Example 9: rank()

Most applications of `rank()` will be to one variable, but the argument *exp* can be more general, namely, an expression. In particular, `rank(-`*varname*`)` reverses ranks from those obtained by `rank(`*varname*`)`.

The default ranking and those obtained by using one of the `track`, `field`, and `unique` options differ principally in their treatment of ties. The default is to assign the same rank to tied values such that the sum of the ranks is preserved. The `track` option assigns the same rank but resembles the convention in track events; thus, if one person had the lowest time and three persons tied for second-lowest time, their ranks would be 1, 2, 2, and 2, and the next person(s) would have rank 5. The `field` option acts similarly except that the highest is assigned rank 1, as in field events in which the greatest distance or height wins. The `unique` option breaks ties arbitrarily: its most obvious use is assigning ranks for a graph of ordered values. See also `group()` for another kind of "ranking".

```
. use https://www.stata-press.com/data/r17/auto, clear
(1978 automobile data)
. keep in 1/10
(64 observations deleted)
. egen rank = rank(mpg)
. egen rank_r = rank(-mpg)
. egen rank_f = rank(mpg), field
```

```
. egen rank_t = rank(mpg), track
. egen rank_u = rank(mpg), unique
. egen rank_ur = rank(-mpg), unique
. sort rank_u
. list mpg rank*
```

	mpg	rank	rank_r	rank_f	rank_t	rank_u	rank_ur
1.	15	1	10	10	1	1	10
2.	16	2	9	9	2	2	9
3.	17	3	8	8	3	3	8
4.	18	4	7	7	4	4	7
5.	19	5	6	6	5	5	6
6.	20	6.5	4.5	4	6	6	5
7.	20	6.5	4.5	4	6	7	4
8.	22	8.5	2.5	2	8	8	3
9.	22	8.5	2.5	2	8	9	2
10.	26	10	1	1	10	10	1

Standardized variables

▷ Example 10: std()

We have a variable called `age` recording the median age in the 50 states. We wish to create the standardized value of age and verify the calculation:

```
. use https://www.stata-press.com/data/r17/states1, clear
(State data)
. egen stdage = std(age)
. summarize age stdage
```

Variable	Obs	Mean	Std. dev.	Min	Max
age	50	29.54	1.693445	24.2	34.7
stdage	50	6.41e-09	1	-3.153336	3.047044

```
. correlate age stdage
(obs=50)
```

	age	stdage
age	1.0000	
stdage	1.0000	1.0000

`summarize` shows that the new variable has a mean of approximately zero; 10^{-9} is the precision of a `float` and is close enough to zero for all practical purposes. If we wanted, we could have typed `egen double stdage = std(age)`, making `stdage` a double-precision variable, and the mean would have been 10^{-16}. In any case, `summarize` also shows that the standard deviation is 1. `correlate` shows that the new variable and the original variable are perfectly correlated.

We may optionally specify the mean and standard deviation for the new variable. For instance,

```
. egen newage1 = std(age), sd(2)
. egen newage2 = std(age), mean(2) sd(4)
. egen newage3 = std(age), mean(2)
. summarize age newage1-newage3
```

Variable	Obs	Mean	Std. dev.	Min	Max
age	50	29.54	1.693445	24.2	34.7
newage1	50	1.28e-08	2	-6.306671	6.094089
newage2	50	2	4	-10.61334	14.18818
newage3	50	2	1	-1.153336	5.047044

```
. correlate age newage1-newage3
(obs=50)
```

	age	newage1	newage2	newage3
age	1.0000			
newage1	1.0000	1.0000		
newage2	1.0000	1.0000	1.0000	
newage3	1.0000	1.0000	1.0000	1.0000

◁

Row functions

▷ Example 11: rowtotal()

`generate`'s `sum()` function creates the vertical, running sum of its argument, whereas `egen`'s `total()` function creates a constant equal to the overall sum. `egen`'s `rowtotal()` function, however, creates the horizontal sum of its arguments. They all treat missing as zero. However, if the `missing` option is specified with `total()` or `rowtotal()`, then *newvar* will contain missing values if all values of *exp* or *varlist* are missing.

```
. use https://www.stata-press.com/data/r17/egenxmpl4, clear
. egen hsum = rowtotal(a b c)
. generate vsum = sum(hsum)
. egen sum = total(hsum)
. list
```

	a	b	c	hsum	vsum	sum
1.	.	2	3	5	5	63
2.	4	.	6	10	15	63
3.	7	8	.	15	30	63
4.	10	11	12	33	63	63

◁

Example 12: rowmean(), rowmedian(), rowpctile(), rowsd(), and rownonmiss()

`summarize` displays the mean and standard deviation of a variable across observations; program writers can access the mean in `r(mean)` and the standard deviation in `r(sd)` (see [R] **summarize**). `egen`'s `rowmean()` function creates the means of observations across variables. `rowmedian()` creates the medians of observations across variables. `rowpctile()` returns the #th percentile of the variables specified in *varlist*. `rowsd()` creates the standard deviations of observations across variables. `rownonmiss()` creates a count of the number of nonmissing observations, the denominator of the `rowmean()` calculation:

```
. use https://www.stata-press.com/data/r17/egenxmpl4, clear
. egen avg = rowmean(a b c)
. egen median = rowmedian(a b c)
. egen pct25 = rowpctile(a b c), p(25)
. egen std = rowsd(a b c)
. egen n = rownonmiss(a b c)
. list
```

	a	b	c	avg	median	pct25	std	n
1.	.	2	3	2.5	2.5	2	.7071068	2
2.	4	.	6	5	5	4	1.414214	2
3.	7	8	.	7.5	7.5	7	.7071068	2
4.	10	11	12	11	11	10	1	3

Example 13: rowmiss()

`rowmiss()` returns $k -$ `rownonmiss()`, where k is the number of variables specified. `rowmiss()` can be especially useful for finding casewise-deleted observations caused by missing values.

```
. use https://www.stata-press.com/data/r17/auto3, clear
(1978 automobile data)
. correlate price weight mpg
(obs=70)
```

	price	weight	mpg
price	1.0000		
weight	0.5309	1.0000	
mpg	-0.4478	-0.7985	1.0000

```
. egen excluded = rowmiss(price weight mpg)
. list make price weight mpg if excluded~=0
```

	make	price	weight	mpg
5.	Buick Electra	.	4,080	15
12.	Cad. Eldorado	14,500	3,900	.
40.	Olds Starfire	4,195	.	24
51.	Pont. Phoenix	.	3,420	.

▷ Example 14: rowmin(), rowmax(), rowfirst(), and rowlast()

`rowmin()`, `rowmax()`, `rowfirst()`, and `rowlast()` return the minimum, maximum, first, or last nonmissing value, respectively, for the specified variables within an observation (row).

```
. use https://www.stata-press.com/data/r17/egenxmpl5, clear
. egen min = rowmin(x y z)
(1 missing value generated)
. egen max = rowmax(x y z)
(1 missing value generated)
. egen first = rowfirst(x y z)
(1 missing value generated)
. egen last = rowlast(x y z)
(1 missing value generated)
. list, sep(4)
```

	x	y	z	min	max	first	last
1.	-1	2	3	-1	3	-1	3
2.	.	-6	.	-6	-6	-6	-6
3.	7	.	-5	-5	7	7	-5
4.	.	.	.	.	.	.	.
5.	4	.	.	4	4	4	4
6.	.	.	8	8	8	8	8
7.	.	3	7	3	7	3	7
8.	5	-1	6	-1	6	5	6

◁

Categorical and integer variables

▷ Example 15: anyvalue(), anymatch(), and anycount()

`anyvalue()`, `anymatch()`, and `anycount()` are for categorical or other variables taking integer values. If we define a subset of values specified by an integer *numlist* (see **[U] 11.1.8 numlist**), `anyvalue()` extracts the subset, leaving every other value missing; `anymatch()` defines an indicator variable (1 if in subset, 0 otherwise); and `anycount()` counts occurrences of the subset across a set of variables. Therefore, with just one variable, `anymatch(`*varname*`)` and `anycount(`*varname*`)` are equivalent.

With the `auto` dataset, we can generate a variable containing the high values of `rep78` and a variable indicating whether `rep78` has a high value:

```
. use https://www.stata-press.com/data/r17/auto, clear
(1978 automobile data)
. egen hirep = anyvalue(rep78), v(3/5)
(15 missing values generated)
. egen ishirep = anymatch(rep78), v(3/5)
```

Here it is easy to produce the same results with official Stata commands:

```
. generate hirep = rep78 if inlist(rep78,3,4,5)
. generate byte ishirep = inlist(rep78,3,4,5)
```

However, as the specification becomes more complicated or involves several variables, the egen functions may be more convenient.

◁

▷ Example 16: group( )

group() maps the distinct groups of a varlist to a categorical variable that takes on integer values from 1 to the total number of groups. order of the groups is that of the sort order of *varlist*. The *varlist* may be of numeric variables, string variables, or a mixture of the two. The resulting variable can be useful for many purposes, including stepping through the distinct groups easily and systematically and cleaning up an untidy ordering. Suppose that the actual (and arbitrary) codes present in the data are 1, 2, 4, and 7, but we desire equally spaced numbers, as when the codes will be values on one axis of a graph. group() maps these to 1, 2, 3, and 4.

We have a variable agegrp that takes on the values 24, 40, 50, and 65, corresponding to age groups 18–24, 25–40, 41–50, and 51 and above. Perhaps we created this coding using the recode() function (see [U] **13.3 Functions** and [U] **26 Working with categorical data and factor variables**) from another age-in-years variable:

```
. generate agegrp=recode(age,24,40,50,65)
```

We now want to change the codes to 1, 2, 3, and 4:

```
. egen agegrp2 = group(agegrp)
```

◁

▷ Example 17: group( ) with missing values

We have two categorical variables, race and sex, which may be string or numeric. We want to use ir (see [R] **Epitab**) to create a Mantel–Haenszel weighted estimate of the incidence rate. ir, however, allows only one variable to be specified in its by() option. We type

```
. use https://www.stata-press.com/data/r17/egenxmpl6, clear
. egen racesex = group(race sex)
(2 missing values generated)
. ir deaths smokes pyears, by(racesex)
  (output omitted)
```

The new numeric variable, racesex, will be missing wherever race or sex is missing (meaning . for numeric variables and "" for string variables), so missing values will be handled correctly. When we list some of the data, we see

```
. list race sex racesex in 1/7, sep(0)
```

	race	sex	racesex
1.	White	Female	1
2.	White	Male	2
3.	Black	Female	3
4.	Black	Male	4
5.	Black	Male	4
6.	.	Female	.
7.	Black	.	.

group() began by putting the data in the order of the grouping variables and then assigned the numeric codes. Observations 6 and 7 were assigned to racesex = . because, in one case, race was not known, and in the other, sex was not known. (These observations were not used by ir.)

If we wanted the unknown groups to be treated just as any other category, we could have typed

```
. egen rs2 = group(race sex), missing
. list race sex rs2 in 1/7, sep(0)
```

	race	sex	rs2
1.	White	Female	1
2.	White	Male	2
3.	Black	Female	3
4.	Black	Male	4
5.	Black	Male	4
6.	.	Female	6
7.	Black	.	5

The resulting variable from `group()` does not have value labels. Therefore, the values carry no indication of meaning. Interpretation requires comparison with the original *varlist*. To get value labels, we specify the option `label`.

```
. egen rs3 = group(race sex), missing label
. list race sex rs3 in 1/7, sep(0)
```

	race	sex	rs3
1.	White	Female	White Female
2.	White	Male	White Male
3.	Black	Female	Black Female
4.	Black	Male	Black Male
5.	Black	Male	Black Male
6.	.	Female	. Female
7.	Black	.	Black .

The numeric values of the generated variable `rs3` are the same as `rs2`, but `rs3` has a value label that indicates the categories of `race` and `sex` that define the groups. The value label created by `group()` uses the actual values of the categorical variables or their value labels, if they exist. In this case, the categorical variables `race` and `sex` are numeric variables with value labels, so their value labels were used to create the value label for `rs3`. ◁

String variables

Concatenation of string variables is provided in Stata. In context, Stata understands the addition symbol `+` as specifying concatenation or adding strings end to end. `"soft" + "ware"` produces `"software"`, and given string variables `s1` and `s2`, `s1 + s2` indicates their concatenation.

The complications that may arise in practice include wanting 1) to concatenate the string versions of numeric variables and 2) to concatenate variables, together with some separator such as a space or a comma. Given numeric variables `n1` and `n2`,

```
. generate newstr = s1 + string(n1) + string(n2) + s2
```

shows how numeric values may be converted to their string equivalents before concatenation, and

```
. generate newstr = s1 + " " + s2 + " " + s3
```

shows how spaces may be added between variables. Stata will automatically assign the most appropriate data type for the new string variables.

▷ Example 18: concat()

`concat()` allows us to do everything in one line concisely.

```
. egen newstr = concat(s1 n1 n2 s2)
```

carries with it an implicit instruction to convert numeric values to their string equivalents, and the appropriate string data type is worked out within `concat()` by Stata's automatic promotion. Moreover,

```
. egen newstr = concat(s1 s2 s3), p(" ")
```

specifies that spaces be used as separators. (The default is to have no separation of concatenated strings.)

As an example of punctuation other than a space, consider

```
. egen fullname = concat(surname forename), p(", ")
```

Noninteger numerical values can cause difficulties, but

```
. egen newstr = concat(n1 n2), format(%9.3f) p(" ")
```

specifies the use of format `%9.3f`. This is equivalent to

```
. generate str1 newstr = ""
. replace newstr = string(n1,"%9.3f") + " " + string(n2,"%9.3f")
```

See [FN] **String functions** for more about `string()`.

◁

As a final flourish, the `decode` option instructs `concat()` to use value labels. With that option, the `maxlength()` option may also be used. For more details about `decode`, see [D] **encode**. Unlike the `decode` command, however, `concat()` uses `string(`*varname*`)`, not `""`, whenever values of *varname* are not associated with value labels, and the `format()` option, whenever specified, applies to this use of `string()`.

▷ Example 19: ends()

The `ends(`*strvar*`)` function is used for subdividing strings. The approach is to find specified separators by using the `strpos()` string function and then to extract what is desired, which either precedes or follows the separators, using the `substr()` string function.

By default, substrings are considered to be separated by individual spaces, so we will give definitions in those terms and then generalize.

The head of the string is whatever precedes the first space or is the whole of the string if no space occurs. This could also be called the first "word". The tail of the string is whatever follows the first space. This could be nothing or one or more words. The last word in the string is whatever follows the last space or is the whole of the string if no space occurs.

To clarify, let's look at some examples. The quotation marks here just mark the limits of each string and are not part of the strings.

	head	tail	last
"frog"	"frog"	""	"frog"
"frog toad"	"frog"	"toad"	"toad"
"frog toad newt"	"frog"	"toad newt"	"newt"
"frog toad newt"	"frog"	" toad newt"	"newt"
"frog toad newt"	"frog"	"toad newt"	"newt"

The main subtlety is that these functions are literal, so the tail of `"frog  toad newt"`, in which two spaces follow `"frog"`, includes the second of those spaces, and is thus `" toad newt"`. Therefore, you may prefer to use the `trim` option to trim the result of any leading or trailing spaces, producing `"toad newt"` in this instance.

The `punct(`*pchars*`)` option may be used to specify separators other than spaces. The general definitions of the `head`, `tail`, and `last` options are therefore interpreted in terms of whatever separator has been specified; that is, they are relative to the first or last occurrence of the separator in the string value. Thus, with `punct(,)` and the string `"Darwin, Charles Robert"`, the head is `"Darwin"`, and the tail and the last are both `" Charles Robert"`. Note again the leading space in this example, which may be trimmed with `trim`. The punctuation (here the comma, "`,`") is discarded, just as it is with one space.

pchars, the argument of `punct()`, will usually, but not always, be one character. If two or more characters are specified, these must occur together; for example, `punct(:;)` would mean that words are separated by a colon followed by a semicolon (that is, `:;`). It is not implied, in particular, that the colon and semicolon are alternatives. To do that, you would have to modify the programs presented here or resort to first principles by using `split`; see [D] **split**.

With personal names, the `head` or `last` option might be applied to extract surnames if strings were similar to `"Darwin, Charles Robert"` or `"Charles Robert Darwin"`, with the surname coming first or last. What then happens with surnames like `"von Neumann"` or `"de la Mare"`? `"von Neumann, John"` is no problem, if the comma is specified as a separator, but the `last` option is not intelligent enough to handle `"Walter de la Mare"` properly.

◁

Acknowledgments

The `cut()` function was written by David Clayton (retired) of the Cambridge Institute for Medical Research and Michael Hills (retired) of the London School of Hygiene and Tropical Medicine.

Many of the other `egen` functions were written by Nicholas J. Cox of the Department of Geography at Durham University, UK, and coeditor of the *Stata Journal* and author of *Speaking Stata Graphics*.

References

Andrews, D. F., P. J. Bickel, F. R. Hampel, P. J. Huber, W. H. Rogers, and J. W. Tukey. 1972. *Robust Estimates of Location: Survey and Advances*. Princeton, NJ: Princeton University Press.

Cappellari, L., and S. P. Jenkins. 2006. Calculation of multivariate normal probabilities by simulation, with applications to maximum simulated likelihood estimation. *Stata Journal* 6: 156–189.

Cox, N. J. 2009. Speaking Stata: Rowwise. *Stata Journal* 9: 137–157.

——. 2014. Speaking Stata: Self and others. *Stata Journal* 14: 432–444.

——. 2020. Speaking Stata: More ways for rowwise. *Stata Journal* 20: 481–488.

——. 2021. Speaking Stata: Ordering or ranking groups of observations. *Stata Journal* 21: 818–837.

Cox, N. J., and C. B. Schechter. 2018. Speaking Stata: Seven steps for vexatious string variables. *Stata Journal* 18: 981–994.

David, H. A. 1998. Early sample measures of variability. *Statistical Science* 13: 368–377. https://doi.org/10.1214/ss/1028905831.

Gallup, J. L. 2019. Grade functions. *Stata Journal* 19: 459–476.

Hampel, F. R., E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel. 1986. *Robust Statistics: The Approach Based on Influence Functions*. New York: Wiley.

Huber, C. 2014. How to simulate multilevel/longitudinal data. *The Stata Blog: Not Elsewhere Classified.* http://blog.stata.com/2014/07/18/how-to-simulate-multilevellongitudinal-data/.

Kohler, U., and J. Zeh. 2012. Apportionment methods. *Stata Journal* 12: 375–392.

Mitchell, M. N. 2020. *Data Management Using Stata: A Practical Handbook*. 2nd ed. College Station, TX: Stata Press.

Pinzon, E. 2015. Fixed effects or random effects: The Mundlak approach. *The Stata Blog: Not Elsewhere Classified.* http://blog.stata.com/2015/10/29/fixed-effects-or-random-effects-the-mundlak-approach/.

Rios-Avila, F. 2020. Recentered influence functions (RIFs) in Stata: RIF regression and RIF decomposition. *Stata Journal* 20: 51–94.

Salas Pauliac, C. H. 2013. group2: Generating the finest partition that is coarser than two given partitions. *Stata Journal* 13: 867–875.

Weiss, M. 2009. Stata tip 80: Constructing a group variable with specified group sizes. *Stata Journal* 9: 640–642.

Wilcox, R. R. 2003. *Applying Contemporary Statistical Techniques*. San Diego, CA: Academic Press.

Also see

[D] **collapse** — Make dataset of summary statistics

[D] **generate** — Create or change contents of variable

[U] 13.3 Functions