

**Description**

`egen` creates `newvar` of the optionally specified storage type equal to `fcn(arguments)`. Here `fcn()` is a function specifically written for `egen`, as documented below or as written by users. Only `egen` functions may be used with `egen`, and conversely, only `egen` may be used to run `egen` functions.

Depending on `fcn()`, `arguments`, if present, refers to an expression, `varlist`, or a `numlist`, and the `options` are similarly `fcn` dependent. Explicit subscripting (using `_N` and `_n`), which is commonly used with `generate`, should not be used with `egen`; see [U] 13.7 Explicit subscripting.

**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)
2  egen — Extensions to generate

## Syntax

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

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

where depending on the fcn, arguments refers to an expression, varlist, or numlist, and the options are also fcn dependent, and where fcn is

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).

cocat(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(%) 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(#,#,...,#) | group(#) } [icodes label]

may not be combined with by. It creates a new categorical variable coded with the left-hand ends of the grouping intervals specified in the at() option, which expects an ascending numlist.

at(#,#,...,#) supplies the breaks for the groups, in ascending order. The list of breakpoints may be simply a list of numbers separated by commas but can also include the syntax a(b)c, meaning from a to c in steps of size b. 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(). If no breaks are specified, the command expects the group() option.

group(#) specifies the number of equal frequency grouping intervals to be used in the absence of breaks. 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.
egen — Extensions to generate

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) (allows by varlist:)
creates a constant (within varlist) containing the interquartile range of exp. 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)  
returns the mean absolute deviation from the mean (within varlist) of exp.

mean(exp)  
creates a constant (within varlist) containing the mean of exp.

median(exp)  
creates a constant (within varlist) containing the median of exp. Also see pctile().

min(exp) [ , missing]  
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]  
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, and the missing option will treat missing values as categories. minmode returns the lowest value, and maxmode returns the highest value. nummode(#) will return the #th mode, counting from the lowest up. nummode() may not be combined with by. Missing values are excluded from determination of the mode unless missing is specified. Even so, the value of the mode is recorded for observations for which the values of varname are missing unless they are explicitly excluded, that is, by if varname < . or if varname != "".

mtr(year income)  
may not be combined with by. It returns the U.S. marginal income tax rate for a married couple with taxable income income in year year, where 1930 ≤ year ≤ 2019. year and income may be specified as variable names or constants; for example, mtr(1993 faminc), mtr(surveyyr 28000), or mtr(surveyyr faminc). A blank or comma may be used to separate income from year.

pc(exp) [ , prop]  
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(#)]  
creates a constant (within varlist) containing the #th percentile of exp. If p(#) is not specified, 50 is assumed, meaning medians. Also see median().

rank(exp) [ , field|track|unique]  
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.

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.
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.

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. Where none of the variables exist, newvar is set to missing.

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

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)—this is the value used by rowmean() for the denominator in the mean calculation.

String variables may not be specified unless the strok option is also specified. If strok is specified, string variables will be counted as containing missing values when they contain "". Numeric variables will be counted as containing missing when their value is “≥ .”.

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

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.

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

seq() [, from(#) to(#) block(#)]
(allow 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.

\[
\text{skew}(\text{exp}) \quad (\text{allows by varlist:})
\]

returns the skewness (within `varlist`) of `exp`.

\[
\text{std}(\text{exp}) \quad [\text{, mean(#)} \text{ sd(#)}] \quad (\text{allows by varlist:})
\]

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`).

\[
\text{tag}(\text{varlist}) \quad [\text{, missing}]
\]

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.

\[
\text{total}(\text{exp}) \quad [\text{, missing}]
\]

(allow 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**

Remarks are presented under the following headings:

- **Summary statistics**
- **Generating patterns**
- **Marking differences among variables**
- **Ranks**
- **Standardized variables**
- **Row functions**
- **Categorical and integer variables**
- **String variables**
- **U.S. marginal income tax rate**

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

Technical note

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
+-----+--------+---------+
<table>
<thead>
<tr>
<th>a</th>
<th>sum1</th>
<th>sum2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>
+-----+--------+---------+

Technical note

The definitions and formulas used by these 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 fifth 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. It is possible that missing is the most common value in a variable (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 values are by default excluded from determination of modes. If you wish to include them, use the missing option.

In contrast, egen mode = mode(varname) allows the generation of nonmissing modes for observations for which varname is missing. This allows use of the mode as one simple means of imputing categorical variables. If you want the mode to be missing whenever varname is missing, you can specify if varname < . or if varname != "" or, most generally, if !missing(varname).

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).

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 3: 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)
```

<table>
<thead>
<tr>
<th>i</th>
<th>w</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>100</td>
<td>22</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td>99</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td>98</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
<td>97</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td>96</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6.</td>
<td>6</td>
<td>95</td>
<td>-3</td>
<td>3</td>
</tr>
<tr>
<td>7.</td>
<td>7</td>
<td>94</td>
<td>-8</td>
<td>4</td>
</tr>
<tr>
<td>8.</td>
<td>8</td>
<td>93</td>
<td>-13</td>
<td>4</td>
</tr>
<tr>
<td>9.</td>
<td>9</td>
<td>92</td>
<td>-18</td>
<td>5</td>
</tr>
<tr>
<td>10.</td>
<td>10</td>
<td>91</td>
<td>-23</td>
<td>5</td>
</tr>
<tr>
<td>11.</td>
<td>11</td>
<td>90</td>
<td>-28</td>
<td>6</td>
</tr>
<tr>
<td>12.</td>
<td>12</td>
<td>89</td>
<td>-33</td>
<td>6</td>
</tr>
</tbody>
</table>
Example 4: Patterns produced by fill()

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

```stata
. 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)
   +-----+-----+-----+-----+
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-3</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>40</td>
</tr>
</tbody>
</table>
   +-----+-----+-----+-----+
   | 5   | 0   | -3  | 50  |
   | 1   | 8   | 0   | 10  |
   | 7   | 0   | 1   | 3   |
   | 0   | 3   | 6   | 30  |
   +-----+-----+-----+-----+
   | 1   | 8   | -3  | 40  |
   | 1   | 8   | -3  | 40  |
   | 1   | 8   | -3  | 40  |
   | 1   | 8   | -3  | 40  |
   +-----+-----+-----+-----+
```

Example 5: 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.

```stata
. 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)
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>3.</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>4.</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>5.</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>6.</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>7.</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>8.</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>9.</td>
<td>9</td>
<td>5</td>
<td>3</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>10.</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11.</td>
<td>11</td>
<td>6</td>
<td>5</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12.</td>
<td>12</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

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 6: `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
```

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

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

Ranks

Example 7: `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
. 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*

<table>
<thead>
<tr>
<th>mpg</th>
<th>rank</th>
<th>rank_r</th>
<th>rank_f</th>
<th>rank_t</th>
<th>rank_u</th>
<th>rank_ur</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>9</td>
<td>9</td>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>6.5</td>
<td>4.5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>6.5</td>
<td>4.5</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>8.5</td>
<td>2.5</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>8.5</td>
<td>2.5</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>26</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Standardized variables

Example 8: 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
(40)=50

<table>
<thead>
<tr>
<th>age</th>
<th>stdage</th>
</tr>
</thead>
</table>
age    | 1.0000 |
stdage | 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
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>50</td>
<td>29.54</td>
<td>1.693445</td>
<td>24.2</td>
<td>34.7</td>
</tr>
<tr>
<td>newage1</td>
<td>50</td>
<td>1.28e-08</td>
<td>2</td>
<td>-6.306671</td>
<td>6.094089</td>
</tr>
<tr>
<td>newage2</td>
<td>50</td>
<td>2</td>
<td>4</td>
<td>-10.61334</td>
<td>14.18818</td>
</tr>
<tr>
<td>newage3</td>
<td>50</td>
<td>2</td>
<td>1</td>
<td>-1.153336</td>
<td>5.047044</td>
</tr>
</tbody>
</table>

```
correlate age newage1-newage3
```

```
(obs=50)
```

<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>newage1</th>
<th>newage2</th>
<th>newage3</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>newage1</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>newage2</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>newage3</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

---

**Row functions**

Example 9: `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
```

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>hsum</th>
<th>vsum</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>.</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>4</td>
<td>.</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>3.</td>
<td>7</td>
<td>8</td>
<td>.</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>4.</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>33</td>
<td>63</td>
</tr>
</tbody>
</table>
Example 10: 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 \texttt{r(mean)} and the standard deviation in \texttt{r(sd)} (see \texttt{[R] summarize}). \texttt{egen}'s \texttt{rowmean()} function creates the means of observations across variables. \texttt{rowmedian()} creates the medians of observations across variables. \texttt{rowpctile()} returns the \#th percentile of the variables specified in \texttt{varlist}. \texttt{rowsd()} creates the standard deviations of observations across variables. \texttt{rownonmiss()} creates a count of the number of nonmissing observations, the denominator of the \texttt{rowmean()} calculation:

\begin{verbatim}
. 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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

Example 11: rowmiss()

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

\begin{verbatim}
. 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
\end{verbatim}

<table>
<thead>
<tr>
<th></th>
<th>price</th>
<th>weight</th>
<th>mpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buick</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Electra</td>
<td>4,080</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>Cad. Eldorado</td>
<td>14,500</td>
<td>3,900</td>
<td></td>
</tr>
<tr>
<td>Olds Starfire</td>
<td>4,195</td>
<td>.</td>
<td>24</td>
</tr>
<tr>
<td>Pont. Phoenix</td>
<td>.</td>
<td>3,420</td>
<td></td>
</tr>
</tbody>
</table>
Example 12: 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).

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>.</td>
<td>-6</td>
<td>.</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>.</td>
<td>-5</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>.</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>.</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>.</td>
<td>7</td>
<td>7</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>.</td>
<td>5</td>
<td>-1</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Categorical and integer variables

Example 13: 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:

```stata
. 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:

```stata
. 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 14: *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 15: *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)

<table>
<thead>
<tr>
<th>race</th>
<th>sex</th>
<th>racesex</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Female</td>
<td>1</td>
</tr>
<tr>
<td>White</td>
<td>Male</td>
<td>2</td>
</tr>
<tr>
<td>Black</td>
<td>Female</td>
<td>3</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>.</td>
<td>Female</td>
<td>.</td>
</tr>
<tr>
<td>Black</td>
<td></td>
<td>.</td>
</tr>
</tbody>
</table>
```

*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)
```

<table>
<thead>
<tr>
<th>race</th>
<th>sex</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Female</td>
<td>1</td>
</tr>
<tr>
<td>White</td>
<td>Male</td>
<td>2</td>
</tr>
<tr>
<td>Black</td>
<td>Female</td>
<td>3</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>4</td>
</tr>
<tr>
<td>.</td>
<td>Female</td>
<td>6</td>
</tr>
<tr>
<td>Black</td>
<td>.</td>
<td>5</td>
</tr>
</tbody>
</table>

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)
```

<table>
<thead>
<tr>
<th>race</th>
<th>sex</th>
<th>rs3</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Female</td>
<td>White Female</td>
</tr>
<tr>
<td>White</td>
<td>Male</td>
<td>White Male</td>
</tr>
<tr>
<td>Black</td>
<td>Female</td>
<td>Black Female</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>Black Male</td>
</tr>
<tr>
<td>Black</td>
<td>Male</td>
<td>Black Male</td>
</tr>
<tr>
<td>.</td>
<td>Female</td>
<td>. Female</td>
</tr>
<tr>
<td>Black</td>
<td>.</td>
<td>Black .</td>
</tr>
</tbody>
</table>

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 16: `concat()`

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

```stata
. 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,

```stata
. 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

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

Noninteger numerical values can cause difficulties, but

```stata
. egen newstr = concat(n1 n2), format(%9.3f) p(" ")
```
specifies the use of format `%9.3f`. This is equivalent to

```stata
. 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 17: `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.

<table>
<thead>
<tr>
<th></th>
<th>head</th>
<th>tail</th>
<th>last</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;frog&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot; &quot;</td>
<td>&quot;frog&quot;</td>
</tr>
<tr>
<td>&quot;frog toad&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad&quot;</td>
<td>&quot;toad&quot;</td>
</tr>
<tr>
<td>&quot;frog toad newt&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad newt&quot;</td>
<td>&quot;newt&quot;</td>
</tr>
<tr>
<td>&quot;frog toad newt&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad newt&quot;</td>
<td>&quot;newt&quot;</td>
</tr>
<tr>
<td>&quot;frog toad newt&quot;</td>
<td>&quot;frog&quot;</td>
<td>&quot;toad newt&quot;</td>
<td>&quot;newt&quot;</td>
</tr>
</tbody>
</table>
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.

U.S. marginal income tax rate

mtr(year income) (Schmidt 1993, 1994) returns the U.S. marginal income tax rate for a married couple with taxable income income in year year, where 1930 ≤ year ≤ 2019.

Example 18: mtr()

Schmidt (1993) examines the change in the progressivity of the U.S. tax schedule over the period from 1930 to 1990. As a measure of progressivity, he calculates the difference in the marginal tax rates at the 75th and 25th percentiles of income, using a dataset of percentiles of taxable income developed by Hakkio, Rush, and Schmidt (1996). (Certain aspects of the income distribution are imputed in these data.) A subset of the data contains the following:

```
. describe
Contains data from income1.dta
Observations: 61
Variables: 4 12 Feb 2021 03:33

<table>
<thead>
<tr>
<th>Variable</th>
<th>Storage type</th>
<th>Display format</th>
<th>Value label</th>
<th>Variable label</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>float</td>
<td>%9.0g</td>
<td>Year</td>
<td></td>
</tr>
<tr>
<td>inc25</td>
<td>float</td>
<td>%9.0g</td>
<td>25th percentile</td>
<td></td>
</tr>
<tr>
<td>inc50</td>
<td>float</td>
<td>%9.0g</td>
<td>50th percentile</td>
<td></td>
</tr>
<tr>
<td>inc75</td>
<td>float</td>
<td>%9.0g</td>
<td>75th percentile</td>
<td></td>
</tr>
</tbody>
</table>
```

Sorted by:
Given the series for income and the four-digit year, we can generate the marginal tax rates corresponding to the 25th and 75th percentiles of income:

\[
\text{. egen mtr25 = mtr(year inc25)}
\]
\[
\text{. egen mtr75 = mtr(year inc75)}
\]

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>61</td>
<td>1960</td>
<td>17.75293</td>
<td>1930</td>
<td>1990</td>
</tr>
<tr>
<td>inc25</td>
<td>61</td>
<td>6948.272</td>
<td>6891.921</td>
<td>819.4</td>
<td>27227.35</td>
</tr>
<tr>
<td>inc50</td>
<td>61</td>
<td>11645.15</td>
<td>11550.71</td>
<td>1373.29</td>
<td>45632.43</td>
</tr>
<tr>
<td>inc75</td>
<td>61</td>
<td>18166.43</td>
<td>18019.1</td>
<td>2142.33</td>
<td>71186.58</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Obs</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtr25</td>
<td>61</td>
<td>.1664898</td>
<td>.0677949</td>
<td>.01125</td>
<td>.23</td>
</tr>
<tr>
<td>mtr75</td>
<td>61</td>
<td>.2442053</td>
<td>.1148427</td>
<td>.01125</td>
<td>.424625</td>
</tr>
</tbody>
</table>

### Methods and formulas

Stata users have written many extra functions for `egen`. Type `net search egen` to locate Internet sources of programs.

### Acknowledgments

The `mtr()` function of `egen` was written by Timothy J. Schmidt formerly with the Federal Reserve Bank of Kansas City.

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


---

Also see

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

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

[U] **13.3 Functions**