# Title

stata.com

**recode —** Recode categorical variables



## Syntax

*Basic syntax*

```
recode varlist (rule) [(rule) ...] [, generate(newvar)]
```

*Full syntax*

```
recode varlist (erule) [(erule) ...] [if] [in] [, options]
```

where the most common forms for *rule* are

| *rule* | Example | Meaning |
|---|---|---|
| `# = #` | `3 = 1` | 3 recoded to 1 |
| `# # = #` | `2 . = 9` | 2 and . recoded to 9 |
| `#/# = #` | `1/5 = 4` | 1 through 5 recoded to 4 |
| `nonmissing = #` | `nonmiss = 8` | all other nonmissing to 8 |
| `missing = #` | `miss = 9` | all other missings to 9 |

where *erule* has the form

*element* [*element* ...] `=` *el* [`"`*label*`"`]

`nonmissing` `=` *el* [`"`*label*`"`]

`missing` `=` *el* [`"`*label*`"`]

`else` | `*` `=` *el* [`"`*label*`"`]

*element* has the form

*el* | *el*`/`*el*

and *el* is

`#` | `min` | `max`

The keyword rules `missing`, `nonmissing`, and `else` must be the last rules specified. `else` may not be combined with `missing` or `nonmissing`.

| *options* | Description |
|---|---|
| Options | |
| `generate(`*newvar*`)` | generate *newvar* containing transformed variables; default is to replace existing variables |
| `prefix(`*str*`)` | generate new variables with *str* prefix |
| `label(`*name*`)` | specify a name for the value label defined by the transformation rules |
| `copyrest` | copy out-of-sample values from original variables |
| `test` | test that rules are invoked and do not overlap |

# Menu

Data > Create or change data > Other variable-transformation commands > Recode categorical variable

# Description

`recode` changes the values of numeric variables according to the rules specified. Values that do not meet any of the conditions of the rules are left unchanged, unless an *otherwise* rule is specified.

A range *#1*`/`*#2* refers to *all* (real and integer) values between *#1* and *#2*, including the boundaries *#1* and *#2*. This interpretation of *#1*`/`*#2* differs from that in numlists.

`min` and `max` provide a convenient way to refer to the minimum and maximum for each variable in *varlist* and may be used in both the from-value and the to-value parts of the specification. Combined with `if` and `in`, the minimum and maximum are determined over the restricted dataset.

The keyword rules specify transformations for values not changed by the previous rules:

| | |
|---|---|
| `nonmissing` | all nonmissing values not changed by the rules |
| `missing` | all missing values (`.`, `.a`, `.b`, ..., `.z`) not changed by the rules |
| `else` | all nonmissing and missing values not changed by the rules |
| `*` | synonym for `else` |

`recode` provides a convenient way to define value labels for the generated variables during the definition of the transformation, reducing the risk of inconsistencies between the definition and value labeling of variables. Value labels may be defined for integer values and for the extended missing values (`.a`, `.b`, ..., `.z`), but not for noninteger values or for sysmiss (`.`).

Although this is not shown in the syntax diagram, the parentheses around the *rule*s and keyword clauses are optional if you transform only one variable and if you do not define value labels.

# Options

Options

`generate(`*newvar*`)` specifies the names of the variables that will contain the transformed variables. `into()` is a synonym for `generate()`. Values outside the range implied by `if` or `in` are set to missing (`.`), unless the `copyrest` option is specified.

If `generate()` is not specified, the input variables are overwritten; values outside the `if` or `in` range are not modified. Overwriting variables is dangerous (you cannot undo changes, value labels may be wrong, etc.), so we strongly recommend specifying `generate()`.

`prefix(`*str*`)` specifies that the recoded variables be returned in new variables formed by prefixing the names of the original variables with *str*.

`label(`*name*`)` specifies a name for the value label defined from the transformation rules. `label()` may be defined only with `generate()` (or its synonym, `into()`) and `prefix()`. If a variable is recoded, the label name defaults to *newvar* unless a label with that name already exists.

`copyrest` specifies that out-of-sample values be copied from the original variables. In line with other data management commands, `recode` defaults to setting *newvar* to missing (`.`) outside the observations selected by `if` *exp* and `in` *range*.

`test` specifies that Stata test whether rules are ever invoked or that rules overlap; for example, `(1/5=1) (3=2)`.

# Remarks and examples

stata.com

Remarks are presented under the following headings:



## Simple examples

Many users experienced with other statistical software use the `recode` command often, but easier and faster solutions in Stata are available. On the other hand, `recode` often provides simple ways to manipulate variables that are not easily accomplished otherwise. Therefore, we show other ways to perform a series of tasks with and without `recode`.

We want to change 1 to 2, leave all other values unchanged, and store the results in the new variable `nx`.

```
. recode x (1 = 2), gen(nx)
```

or

```
. gen nx = x
. replace nx = 2 if nx==1
```

or

```
. gen nx = cond(x==1,2,x)
```

We want to swap 1 and 2, saving them in `nx`.

```
. recode x (1 = 2) (2 = 1), gen(nx)
```

or

```
. gen nx = cond(x==1,2,cond(x==2,1,x))
```

We want to recode `item` by collapsing 1 and 2 into 1, 3 into 2, and 4 to 7 (boundaries included) into 3.

```
. recode item (1 2 = 1) (3 = 2) (4/7 = 3), gen(Ritem)
```

or

```
. gen Ritem = item
. replace Ritem = 1 if inlist(item,1,2)
. replace Ritem = 2 if item==3
. replace Ritem = 3 if inrange(item,4,7)
```

We want to change the "direction" of the $1, \ldots, 5$ valued variables `x1`, `x2`, `x3`, storing the transformed variables in `nx1`, `nx2`, and `nx3` (that is, we form new variable names by prefixing old variable names with an "n").

```
. recode x1 x2 x3 (1=5) (2=4) (3=3) (4=2) (5=1), pre(n) test
```

or

```
. gen nx1 = 6-x1
. gen nx2 = 6-x2
```

```
. gen nx3 = 6-x3
. forvalues i = 1/3 {
          generate nx‘i’ = 6-x‘i’
  }
```

In the categorical variable `religion`, we want to change 1, 3, and the real and integer numbers 3 through 5 into 6; we want to set 2, 8, and 10 to 3 and leave all other values unchanged.

```
. recode religion 1 3/5 = 6 2 8 10 = 3
```

or

```
. replace religion = 6 if religion==1 | inrange(religion,3,5)
. replace religion = 3 if inlist(religion,2,8,10)
```

This example illustrates two features of `recode` that were included for backward compatibility with previous versions of `recode` but that we do not recommend. First, we omitted the parentheses around the rules. This is allowed if you recode one variable and you do not plan to define value labels with `recode` (see below for an explanation of this feature). Personally, we find the syntax without parentheses hard to read, although we admit that we could have used blanks more sensibly. Because difficulties in reading may cause us to overlook errors, we recommend always including parentheses. Second, because we did not specify a `generate()` option, we overwrite the `religion` variable. This is often dangerous, especially for “original” variables in a dataset. We recommend that you always specify `generate()` unless you want to overwrite your data.

## Setting up value labels with recode

The `recode` command is most often used to transform categorical variables, which are many times value labeled. When a value-labeled variable is overwritten by `recode`, it may well be that the value label is no longer appropriate. Consequently, output that is labeled using these value labels may be misleading or wrong.

When `recode` creates one or more new variables with a new classification, you may want to put value labels on these new variables. It is possible to do this in three steps:

1. Create the new variables (`recode ..., gen()`).
2. Define the value label (`label define ...`).
3. Link the value label to the variables (`label value ...`).

Inconsistencies may emerge from mistakes between steps 1 and 2. Especially when you make a change to the recode 1, it is easy to forget to make a similar adjustment to the value label 2. Therefore, `recode` can perform steps 2 and 3 itself.

Consider recoding a series of items with values

1 = strongly agree
2 = agree
3 = neutral
4 = disagree
5 = strongly disagree

into three items:

1 = positive (= “strongly agree” or “agree”)
2 = neutral
3 = negative (= “strongly disagree” or “disagree”)

This is accomplished by typing

```
. recode item* (1 2 = 1 positive) (3 = 2 neutral) (4 5 = 3 negative), pre(R)
> label(Item3)
```

which is much simpler and safer than

```
. recode item1-item7 (1 2 = 1) (3 = 2) (4 5 = 3), pre(R)
. label define Item3  1 positive 2 neutral 3 negative
. forvalues i = 1/7 {
        label value Ritem`i' Item3
  }
```

## ▷ Example 1

As another example, let's recode vote (voting intentions) for 12 political parties in the Dutch parliament into left, center, and right parties. We then tabulate the original and new variables so that we can check that everything came out correctly.

```
. use http://www.stata-press.com/data/r13/recodexmpl
. label list pparty
pparty:
           1 pvda
           2 cda
           3 d66
           4 vvd
           5 groenlinks
           6 sgp
           7 rpf
           8 gpv
           9 aov
          10 unie55
          11 sp
          12 cd
. recode polpref (1 5 11 = 1 left) (2 3 = 2 center) (4 6/10 12 = 3 right),
> gen(polpref3)
(2020 differences between polpref and polpref3)
. tabulate polpref polpref3
```

| pol party choice if elections | RECODE of polpref (pol party choice if elections) | | | |
|---|---|---|---|---|
| | left | center | right | Total |
| pvda | 622 | 0 | 0 | 622 |
| cda | 0 | 525 | 0 | 525 |
| d66 | 0 | 634 | 0 | 634 |
| vvd | 0 | 0 | 930 | 930 |
| groenlinks | 199 | 0 | 0 | 199 |
| sgp | 0 | 0 | 54 | 54 |
| rpf | 0 | 0 | 63 | 63 |
| gpv | 0 | 0 | 30 | 30 |
| aov | 0 | 0 | 17 | 17 |
| unie55 | 0 | 0 | 23 | 23 |
| sp | 45 | 0 | 0 | 45 |
| cd | 0 | 0 | 25 | 25 |
| Total | 866 | 1,159 | 1,142 | 3,167 |

◁

## Referring to the minimum and maximum in rules

recode allows you to refer to the minimum and maximum of a variable in the transformation rules. The keywords min and max may be included as a from-value, as well as a to-value.

For example, we might divide age into age categories, storing in iage.

```
. recode age (0/9=1) (10/19=2) (20/29=3) (30/39=4) (40/49=5) (50/max=6),
> gen(iage)
```

or

```
. gen iage = 1 + irecode(age,9,19,29,39,49)
```

or

```
. gen iage = min(6, 1+int(age/10))
```

As another example, we could set all incomes less than 10,000 to 10,000 and those more than 200,000 to 200,000, storing the data in ninc.

```
. recode inc (min/10000 = 10000) (200000/max = 200000), gen(ninc)
```

or

```
. gen ninc = inc
. replace ninc = 10000 if ninc<10000
. replace ninc = 200000 if ninc>200000 & !missing(ninc)
```

or

```
. gen ninc = max(min(inc,200000),10000)
```

or

```
. gen ninc = clip(inc,10000,200000)
```

## Recoding missing values

You can also set up rules in terms of missing values, either as from-values or as to-values. Here recode mimics the functionality of mvdecode and mvencode (see [D] **mvencode**), although these specialized commands execute much faster.

Say that we want to change missing (.) to 9, storing the data in X:

```
. recode x (.=9), gen(X)
```

or

```
. gen X = cond(x==., 9, x)
```

or

```
. mvencode x, mv(.=9) gen(X)
```

We want to change 9 to .a and 8 to ., storing the data in z.

```
. recode x (9=.a) (8=.), gen(z)
```

or

```
. gen z = cond(x==9, .a, cond(x==8, ., x))
```

or

```
. mvdecode x, mv(9=.a, 8=.) gen(z)
```

## Recoding subsets of the data

We want to swap in x the values 1 and 2 only for those observations for which age>40, leaving all other values unchanged. We issue the command

```
. recode x (1=2) (2=1) if age>40, gen(y)
```

or

```
. gen y = cond(x==1,2,cond(x==2,1,x)) if age>40
```

We are in for a surprise. y is missing for observations that do not satisfy the if condition. This outcome is in accordance with how Stata's data manipulation commands usually work. However, it may not be what you intend. The copyrest option specifies that x be copied into y for all nonselected observations:

```
. recode x (1=2) (2=1) if age>40, gen(y) copy
```

or

```
. gen y = x
. recode y (1=2) (2=1) if age>40
```

or

```
. gen y = cond(age>40,cond(x==1,2,cond(x==2,1,x),x))
```

## Otherwise rules

In all our examples so far, recode had an implicit rule that specified that values that did not meet the conditions of any of the rules were to be left unchanged. recode also allows you to use an "otherwise rule" to specify how untransformed values are to be transformed. recode supports three kinds of otherwise conditions:

| | |
|---|---|
| nonmissing | all nonmissing not yet transformed |
| missing | all missing values not yet transformed |
| else | all values, missing or nonmissing, not yet transformed |

The otherwise rules are to be specified *after* the standard transformation rules. nonmissing and missing may be combined with each other, but not with else.

Consider a recode that swaps the values 1 and 2, transforms all other nonmissing values to 3, and transforms all missing values (that is, sysmiss and the extended missing values) to . (sysmiss). We could type

```
. recode x (1=2) (2=1) (nonmissing=3) (missing=.), gen(z)
```

or

```
. gen z = cond(x==1,2,cond(x==2,1,cond(!missing(x),3),.))
```

As a variation, if we had decided to recode all extended missing values to .a but to keep sysmiss . distinct at ., we could have typed

```
. recode x (1=2) (2=1) (.=.) (nonmissing=3) (missing=.a), gen(z)
```

## Test for overlapping rules

`recode` evaluates the rules from left to right. Once a value has been transformed, it will not be transformed again. Thus if rules "overlap", the first matching rule is applied, and further matches are ignored. A common form of overlapping is illustrated in the following example:

```
... (1/5 = 1) (5/10 = 2)
```

Here 5 occurs in the condition parts of both rules. Because rules are matched left to right, 5 matches the first rule, and the second rule will not be tested for 5, unless `recode` is instructed to test for rule overlap with the `test` option.

Other instances of overlapping rules usually arise because you mistyped the rules. For instance, you are recoding voting intentions for parties in elections into three groups of parties (left, center, right), and you type

```
... (1/5 = 1) ... (3 = 2)
```

Party 3 matches the conditions 1/5 and 3. Because `recode` applies the first matching rule, party 3 will be mapped into party category 1. The second matching rule is ignored. It is not clear what was wrong in this example. You may have included party 3 in the range 1/5 or mistyped 3 in the second rule. Either way, `recode` did not notice the problem and your data analysis is in jeopardy. The `test` option specifies that `recode` display a warning message if values are matched by more than one rule. With the `test` option specified, `recode` also tests whether all rules were applied at least once and displays a warning message otherwise. Rules that never matched any data may indicate that you mistyped a rule, although some conditions may not have applied to (a selection of) your data.

# Acknowledgment

This version of `recode` was written by Jeroen Weesie of the Department of Sociology at Utrecht University, The Netherlands.

# Also see

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

[D] **mvencode** — Change missing values to numeric values and vice versa