Title

stata.com

rename group — Rename groups of variables

Syntax

Rename a single variable

`rename` *old new* [`,` *options*$_1$]

Rename groups of variables

`rename (`*old*$_1$ *old*$_2$ ...`) (`*new*$_1$ *new*$_2$ ...`)` [`,` *options*$_1$]

Change the case of groups of variable names

`rename` *old*$_1$ *old*$_2$ ..., { `upper` | `lower` | `proper` } [*options*$_2$]

where *old* and *new* specify the existing and the new variable names. The rules for specifying them are

1. `rename stat status`: Renames `stat` to `status`.

   Rule 1: This is the same `rename` command documented in [D] **rename**, with which you are familiar.

2. `rename (stat inc) (status income)`: Renames `stat` to `status` and `inc` to `income`.

   Rule 2: Use parentheses to specify multiple variables for *old* and *new*.

3. `rename (v1 v2) (v2 v1)`: Swaps `v1` and `v2`.

   Rule 3: Variable names may be interchanged.

4. `rename (a b c) (b c a)`: Swaps names. Renames `a` to `b`, `b` to `c`, and `c` to `a`.

   Rule 4: There is no limit to how many names may be interchanged.

5. `rename (a b c) (c b a)`: Renames `a` to `c` and `c` to `a`, but leaves `b` as is.

   Rule 5: Renaming variables to themselves is allowed.

6. `rename jan* *1`: Renames all variables starting with `jan` to instead end with `1`, for example, `janstat` to `stat1`, `janinc` to `inc1`, etc.

   Rule 6.1: `*` in *old* selects the variables to be renamed. `*` means that zero or more characters go here.

   Rule 6.2: `*` in *new* corresponds with `*` in *old* and stands for the text that `*` in *old* matched.

   `*` in *new* or *old* is called a wildcard character, or just a wildcard.

   `rename jan* *`: Removes prefix `jan`.

   `rename *jan *`: Removes suffix `jan`.

7. `rename jan? ?1`: Renames all variables starting with `jan` and ending in one character by removing `jan` and adding `1` to the end; for example, `jans` is renamed to `s1`, but `janstat` remains unchanged. `?` means that exactly one character goes here, just as `*` means that zero or more characters go here.

   Rule 7: `?` means exactly one character, `??` means exactly two characters, etc.

8. `rename *jan* **`: Removes prefix, midfix, and suffix `jan`, for example, `janstat` to `stat`, `injanstat` to `instat`, and `subjan` to `sub`.

   Rule 8: You may specify more than one wildcard in *old* and in *new*. They correspond in the order given.

   `rename jan*s* *s*1`: Renames all variables that start with `jan` and contain `s` to instead end in `1`, dropping the `jan`, for example, `janstat` to `stat1` and `janest` to `est1`, but not `janinc` to `inc1`.

9. `rename *jan* *`: Removes `jan` and whatever follows from variable names, thereby renaming `statjan` to `stat`, `incjan71` to `inc`, ...

   Rule 9: You may specify more wildcards in *old* than in *new*.

10. `rename *jan* .*`: Removes `jan` and whatever precedes it from variable names, thereby renaming `midjaninc` to `inc`, ...

    Rule 10: Wildcard `.` (dot) in *new* skips over the corresponding wildcard in *old*.

11. `rename *pop jan=`: Adds prefix `jan` to all variables ending in `pop`, for example, `age1pop` to `janage1pop`, ...

    `rename (status bp time) admit=`: Renames `status` to `admitstatus`, `bp` to `admitbp`, and `time` to `admittime`.

    `rename` *whatever* `pre=`: Adds prefix `pre` to all variables selected by *whatever*, however *whatever* is specified.

    Rule 11: Wildcard `=` in *new* specifies the original variable name.

    `rename` *whatever* `=jan`: Adds suffix `jan` to all variables selected by *whatever*.

    `rename` *whatever* `pre=fix`: Adds prefix `pre` and suffix `fix` to all variables selected by *whatever*.

12. `rename v# stat#`: Renames `v1` to `stat1`, `v2` to `stat2`, ..., `v10` to `stat10`, ...

    Rule 12.1: `#` is like `*` but for digits. `#` in *old* selects one or more digits.

    Rule 12.2: `#` in *new* copies the digits just as they appear in the corresponding *old*.

13. `rename v(#) stat(#)`: Renames `v1` to `stat1`, `v2` to `stat2`, ..., but does not rename `v10`, ...

    Rule 13.1: `(#)` in *old* selects exactly one digit. Similarly, `(##)` selects exactly two digits, and so on, up to ten `#` symbols.

    Rule 13.2: `(#)` in *new* means reformat to one or more digits. Similarly, `(##)` reformats to two or more digits, and so on, up to ten `#` symbols.

    `rename v(##) stat(##)`: Renames `v01` to `stat01`, `v02` to `stat02`, ..., `v10` to `stat10`, ..., but does not rename `v0`, `v1`, `v2`, ..., `v9`, `v100`, ...

14. `rename v# v(##)`: Renames `v1` to `v01`, `v2` to `v02`, ..., `v10` to `v10`, `v11` to `v11`, ..., `v100` to `v100`, `v101` to `v101`, ...

    Rule 14: You may combine `#`, `(#)`, `(##)`, ... in *old* with any of `#`, `(#)`, `(##)`, ... in *new*.

    `rename v(##) v(#)`: Renames `v01` to `v1`, `v02` to `v2`, ..., `v10` to `v10`, ..., but does not rename `v001`, etc.

    `rename stat(##) stat_20(##)`: Renames `stat10` to `stat_2010`, `stat11` to `stat_2011`, ..., but does not rename `stat1`, `stat2`, ...

    `rename stat(#) to stat_200(#)`: Renames `stat1` to `stat_2001`, `stat2` to `stat_2002`, ..., but does not rename `stat10` or `stat_2010`.

15. `rename v# (a b c)`: Renames `v1` to `a`, `v10` to `b`, and `v2` to `c` if variables `v1`, `v10`, `v2` appear in that order in the data. Because three variables were specified in *new*, `v#` in *old* must select three variables or `rename` will issue an error.

    Rule 15.1: You may mix syntaxes. Note that the explicit and implied numbers of variables must agree.

    `rename v# (a b c), sort`: Renames (for instance) `v1` to `a`, `v2` to `b`, and `v10` to `c`.

    Rule 15.2: The `sort` option places the variables selected by *old* in order and does so smartly. In the case where `#`, `(#)`, `(##)`, ... appear in *old*, `sort` places the variables in numeric order.

    `rename v* (a b c), sort`: Renames (for instance) `valpha` to `a`, `vbeta` to `b`, and `vgamma` to `c` regardless of the order of the variables in the data.

    Rule 15.3: In the case where `*` or `?` appears in *old*, `sort` places the variables in alphabetical order.

16. `rename v# v#, renumber`: Renames (for instance) `v9` to `v1`, `v10` to `v2`, `v8` to `v3`, ..., assuming that variables `v9`, `v10`, `v8`, ... appear in that order in the data.

    Rule 16.1: The `renumber` option resequences the numbers.

    `rename v# v#, renumber sort`: Renames (for instance) `v8` to `v1`, `v9` to `v2`, `v10` to `v3`, ... Concerning option `sort`, see rule 15.2 above.

    `rename v# v#, renumber(10) sort`: Renames (for instance) `v8` to `v10`, `v9` to `v11`, `v10` to `v12`, ...

    Rule 16.2: The `renumber(#)` option allows you to specify the starting value.

17. `rename v* v#, renumber`: Renames (for instance) `valpha` to `v1`, `vgamma` to `v2`, `vbeta` to `v3`, ..., assuming variables `valpha`, `vgamma`, `vbeta`, ... appear in that order in the data.

    Rule 17: `#` in *new* may correspond to `*`, `?`, `#`, `(#)`, `(##)`, ... in *old*.

    `rename v* v#, renumber sort`: Renames (for instance) `valpha` to `v1`, `vbeta` to `v2`, `vgamma` to `v3`, ... Also see rule 15.3 above concerning the `sort` option.

    `rename *stat stat#, renumber`: Renames, for instance, `janstat` to `stat1`, `febstat` to `stat2`, ... Note that `#` in *new* corresponds to `*` in *old*, just as in the previous example.

    `rename *stat stat(##), renumber`: Renames, for instance, `janstat` to `stat01`, `febstat` to `stat02`, ...

    `rename *stat stat#, renumber(0)`: Renames, for instance, `janstat` to `stat0`, `febstat` to `stat1`, ...

`rename *stat stat#, renumber sort`: Renames, for instance, `aprstat` to `stat1`, `augstat` to `stat2`, ...

18. `rename (a b c) v#, addnumber`: Renames `a` to `v1`, `b` to `v2`, and `c` to `v3`.

    Rule 18: The `addnumber` option allows you to add numbering. More formally, if you specify `addnumber`, you may specify one more wildcard in *new* than is specified in *old*, and that extra wildcard must be `#`, `(#)`, `(##)`, ...

19. `rename a(#)(#) a(#)[2](#)[1]`: Renames `a12` to `a21`, `a13` to `a31`, `a14` to `a41`, ..., `a21` to `a12`, ...

    Rule 19.1: You may specify explicit subscripts with wildcards in *new* to make explicit its matching wildcard in *old*. Subscripts are specified in square brackets after a wildcard in *new*. The number refers to the number of the wildcard in *old*.

    `rename *stat* *[2]stat*[1]`: Swaps prefixes and suffixes; it renames `bpstata` to `astatbp`, `rstater` to `erstatr`, etc.

    `rename *stat* *[2]stat*`: Does the same as above; it swaps prefixes and suffixes.

    Rule 19.2: After specifying a subscripted wildcard, subsequent unsubscripted wildcards correspond to the same wildcards in *old* as they would if you had removed the subscripted wildcards altogether.

    `rename v#a# v#_#[1]_a#[2]`: Renames `v1a1` to `v1_1_a1`, `v1a2` to `v1_1_a2`, ..., `v2a1` to `v2_2_a1`, ...

    Rule 19.3: Using subscripts, you may refer to the same wildcard in *old* more than once.

    Subscripts are commonly used to interchange suffixes at the ends of variable names. For instance, you have districts and schools within them, and many of the variable names in your data match `*_#_#`. The first number records district and the second records school within district. To reverse the ordering, you type `rename *_#_# *_#[3]_#[2]`. When specifying subscripts, you refer to them by the position number in the original name. For example, our original name was `*_#_#` so `[1]` refers to `*`, `[2]` refers to the first `#`, and `[3]` refers to the last `#`.

In summary, the pattern specifiers are

Specifier	Meaning in *old*
`*`	0 or more characters
`?`	1 character exactly
`#`	1 or more digits
`(#)`	1 digit exactly
`(##)`	2 digits exactly
`(###)`	3 digits exactly
...	
`(##########)`	10 digits exactly

Specifier	May correspond in *old* with	Meaning in *new*
`*`	`*`, `?`, `#`, `(#)`, ...	copies matched text
`?`	`?`	copies a character
`#`	`#`, `(#)`, ...	copies a number as is
`(#)`	`#`, `(#)`, ...	reformats to 1 or more digits
`(##)`	`#`, `(#)`, ...	reformats to 2 or more digits
...		
`(##########)`	`#`, `(#)`, ...	reformats to 10 digits
`.`	`*`, `?`, `#`, `(#)`, ...	skip
`=`	*nothing*	copies entire variable name

Specifier `#` in any of its guises may also correspond with `*` or `?` if the `renumber` option is specified.

The options are as follows:

options$_1$	Description
`addnumber`	add sequential numbering to end
`addnumber(#)`	`addnumber`, starting at *#*
`renumber`	renumber sequentially
`renumber(#)`	`renumber`, starting at *#*
`sort`	sort before numbering
`dryrun`	do not rename, but instead produce a report
`r`	store variable names in `r()` for programming use

These options correspond to the first and second syntaxes.

options$_2$	Description
`upper`	uppercase variable names (UPPERCASE)
`lower`	lowercase variable names (lowercase)
`proper`	propercase variable names (Propercase)
`dryrun`	do not rename, but instead produce a report
`r`	store variable names in `r()` for programming use

These options correspond to the third syntax. One of `upper`, `lower`, or `proper` must be specified.

Menu

Data > Data utilities > Rename groups of variables

Description

`rename` changes the names of existing variables to the new names specified. See [D] **rename** for the base `rename` syntax. Documented here is the advanced syntax for renaming groups of variables.

Options for renaming variables

`addnumber` and `addnumber(#)` specify to add a sequence number to the variable names. See item 18 of *Syntax*. If # is not specified, the sequence number begins with 1.

`renumber` and `renumber(#)` specify to replace existing numbers or text in a set of variable names with a sequence number. See items 16 and 17 of *Syntax*. If # is not specified, the sequence number begins with 1.

`sort` specifies that the existing names be placed in order before the renaming is performed. See item 15 of *Syntax* for details. This ordering matters only when `addnumber` or `renumber` is also specified or when specifying a list of variable names for *old* or *new*.

`dryrun` specifies that the requested renaming not be performed but instead that a table be displayed showing the old and new variable names. It is often a good idea to specify this option before actually renaming the variables.

`r` is a programmer's option that requests that old and new variable names be stored in `r()`. This option may be specified with or without `dryrun`.

Options for changing the case of groups of variable names

`upper`, `lower`, and `proper` specify how the variables are to be renamed. `upper` specifies that variable names be changed to uppercase; `lower`, to lowercase; and `proper`, to having the first letter capitalized and the remaining letters in lowercase. One of these three options must be specified.

`dryrun` and `r` are the same options as documented directly above.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Advice
Explanation
** matches 0 or more characters; use ?* to match 1 or more*
** is greedy*
 # is greedier

Advice

1. Read [D] **rename** before reading this entry.
2. Read items 1–19 (the Rules) under *Syntax* above before reading the rest of these remarks.
3. Specify the `dryrun` option when using complicated patterns. `dryrun` presents a table of the old and new variable names rather than actually renaming the variables, so you can check that the patterns you have specified produce the desired result.

Explanation

The `rename` command has three syntaxes; see *Syntax*. See [D] **rename** for details on the first syntax, renaming a single variable. The remaining two syntaxes are for renaming groups of variables and for changing the case of groups of variables. These two syntaxes are the ones we will focus on for the remainder of this manual entry. Here they are again:

`rename (`*old*$_1$ *old*$_2$ ...`) (`*new*$_1$ *new*$_2$ ...`)`

`rename` *old*$_1$ *old*$_2$ ..., { `upper` | `lower` | `proper` }

The second syntax shown above merely changes the case of variables, such as `MPG` or `mpg` or `Mpg`. For instance, to rename all variables to be lowercase, type

```
rename *, lower
```

The first syntax shown above is more daunting and more powerful. The first syntax has two styles, with and without parentheses:

```
rename (bp_0 bp_1) (bp_1 bp_0)
rename pop*80 pop_*_1980
```

You can combine the two styles whenever it is convenient.

```
rename v* (mpg weight displacement)
rename (mpg weight displacement) v#, addnumber
rename (bp_0 bp_1 pop*80) (bp_1 bp_0 pop_*_1980)
```

We summarize all of this by simply writing the syntax as

`rename` *old new*, ...

and referring to *old* and *new*.

Wildcards play different but related roles in *old* and *new*. When you type

```
rename pop*80 pop_*_1980
```

the wildcard (`*` in this case) in *old* specifies which variables are to be renamed, and in *new* the wildcard stands for the text that appears in the variables to be renamed. In this case, there is just one wildcard, but sometimes there are more.

In *old*, `*` means zero or more characters go here. Specifying `pop*80` means find all variables that begin with `pop` and end in `80`. Say that doing so results in three variables being found: `poplt2080`, `pop204080`, and `pop41plus80`. To understand how `*` is interpreted in *new*, it is useful to write the three found variables like this:

pop*80	=	pop	+	*	+	80
poplt2080	=	pop	+	lt20	+	80
pop204080	=	pop	+	2040	+	80
pop41plus80	=	pop	+	41plus	+	80

`*` in *new* refers to what was found by `*` in *old*. So the new pattern `pop_*_1980` will assemble the following new variable names for each of the old names:

old variable	* is	→	pop_*_1980 is
poplt2080	lt20	→	pop_lt20_1980
pop204080	2040	→	pop_2040_1980
pop41plus80	41plus	→	pop_41plus_1980

Thus typing `rename pop*80 pop_*_1980` is equivalent to typing

```
rename poplt2080    pop_lt20_1980
rename pop204080    pop_2040_1980
rename pop41plus80 pop_41plus_1980
```

There are three basic wildcard characters for specification in *old*, and they filter the variables to be renamed:

`*`	0 or more characters go here
`?`	exactly 1 character goes here
`#`	number goes here (this one comes in 11 flavors!)

The generic `#` listed above collects all the digits. The other 10 flavors are `(#)`, which means exactly 1 digit goes here; `(##)`, which means exactly 2 digits go here; and so on, up to exactly 10 digits go here.

All the above, the $3 + 10 = 13$ wildcard characters, can appear in *new*, where each has a different but related meaning:

`*`	copy corresponding text from *old* as is
`?`	copy corresponding character from *old*
`#`	copy corresponding number from *old* as is
`(#)`	reformat corresponding number from *old* to 1 or more digits
`(##)`	reformat corresponding number from *old* to 2 or more digits
...	

In addition, *new* allows two special wildcard characters of its own:

`=`	copy the entire original variable name
`.`	skip the corresponding text in *old*

With the above information and the definitions of the options, you can derive on your own the first eighteen rules given in *Syntax*. The nineteenth rule concerns subscripting. In *new*, you can specify explicitly to which wildcard in *old* you are referring. You can type

```
rename pop*80 pop_*_1980
```

or you can type

```
rename pop*80 pop_*[1]_1980
```

thus making it explicit that the `*` in *new* is referring to the text matched by the first wildcard in *old*. That `*` corresponds to `*` is hardly surprising, especially when there is only one `*` in *old*, so let's complicate the example:

```
rename v*_* outcome_*_*
```

You can type that command, or you can type

```
rename v*_* outcome_*[1]_*[2]
```

More importantly, you can specify the subscripts in whatever order you wish, so you could type

```
rename v*_* outcome_*[2]_*[1]
```

That command would interchange the text in *old* matched by the two wildcards.

* matches 0 or more characters; use ?* to match 1 or more

`l*a` in *old* matches `louisiana` and it matches `la` because `*` means zero or more characters. What if you want to match `louisiana` and `lymphoma` but not `la`?

For instance, say you have from–to variables named `from*to*` and from variables named `from*`. The problem is that variable `fromtoledo` would match `from*to*`. To avoid that, rather than describing the from–to pattern `from*to*`, you use `from?*to?*`. Thus you could type

```
rename from?*to?* from_?*_to_?*
```

`?*` is not a secret wildcard we have yet to tell you about—it is merely the two wildcards `?` and `*` in sequence. `?` means exactly one character goes here, and `*` means zero or more characters go here, so `?*` means one or more characters go here. In the same way, `??*` means two or more characters go here, and so on.

* is greedy

Consider the existing variable `assessment` and pattern `*s*` in *old*. Clearly, `*s*` matches `assessment`, but how? That is, among these possibilities,

```
assessment  =  *        s    *
              ------------------------
              a      + s + sessment
              as     + s + essment
              asse   + s + sment
              asses  + s + ment
```

which one is true? We need to know the answer to know what each of the corresponding wildcards in *new* will mean. The answer is that `*` is greedy, and the pattern is matched from left to right. As we move through the variable name from left to right, at each step `*` takes the most characters possible, subject to the pattern working out.

```
                 *        s    *
              ------------------------
assessment  =  asses  + s + ment
```

Thus the first `*` in *new* would stand for `asses` and the second would stand for `ment`.

The "subject to the pattern working out" part is important. Variable `sunglasses` would be broken out by `*s*` as

```
                 *           s    *
              ------------------------
sunglasses  =  sunglasse + s + nothing
```

But by `*s?*`, the breakout would be

```
                 *         s    ?    *
              ------------------------
sunglasses  =  sunglas + s + e + s
```

is greedier

Wildcard # in *old* is greedier than *, which means that when * and # are up against each other, # wins.

Consider the pattern *# and the variable name v1234. Given that * is greedy and that the # specifies one or more digits, the possible solutions are

```
v1234 =  *       #
         ---------------
         v123  +  4
         v12   +  34
         v1    +  234
         v     +  1234
```

The solution chosen by rename is the last one, v + 1234. Thus you can type

```
rename *# period_#[2]
```

without concern that some digits might be lost.

Stored results

rename stores nothing in r() by default. If the r option is specified, then rename stores the following in r():

Scalar
- r(n) — number of variables to be renamed

Macros
- r(oldnames) — original variable names
- r(newnames) — new variable names

Variables that are renamed to themselves are omitted from the recorded lists.

Also see

[D] **rename** — Rename variable

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

[D] **varmanage** — Manage variable labels, formats, and other properties