Title

stata.com

foreach — Loop over items

Syntax Description Remarks and examples Also see

Syntax

```
foreach lname { in | of listtype } list {
        Stata commands referring to `lname'
}
```

Allowed are

```
foreach lname in any_list {

foreach lname of local lmacname {

foreach lname of global gmacname {

foreach lname of varlist varlist {

foreach lname of newlist newvarlist {

foreach lname of numlist numlist {
```

Braces must be specified with `foreach`, and

1. the open brace must appear on the same line as `foreach`;
2. nothing may follow the open brace except, of course, comments; the first command to be executed must appear on a new line;
3. the close brace must appear on a line by itself.

Description

`foreach` repeatedly sets local macro *lname* to each element of the list and executes the commands enclosed in braces. The loop is executed zero or more times; it is executed zero times if the list is null or empty. Also see [P] **forvalues**, which is the fastest way to loop over consecutive values, such as looping over numbers from 1 to k.

`foreach` *lname* `in` *list* `{...}` allows a general list. Elements are separated from each other by one or more blanks.

`foreach` *lname* `of local` *list* `{...}` and `foreach` *lname* `of global` *list* `{...}` obtain the list from the indicated place. This method of using `foreach` produces the fastest executing code.

`foreach` *lname* `of varlist` *list* `{`...`}`, `foreach` *lname* `of newlist` *list* `{`...`}`, and `foreach` *lname* `of numlist` *list* `{`...`}` are much like `foreach` *lname* `in` *list* `{`...`}`, except that the *list* is given the appropriate interpretation. For instance,

```
foreach x in mpg weight-turn {
        ...
}
```

has two elements, `mpg` and `weight-turn`, so the loop will be executed twice.

```
foreach x of varlist mpg weight-turn {
        ...
}
```

has four elements, `mpg`, `weight`, `length`, and `turn`, because *list* was given the interpretation of a varlist.

`foreach` *lname* `of varlist` *list* `{`...`}` gives *list* the interpretation of a varlist. The *list* is expanded according to standard variable abbreviation rules, and the existence of the variables is confirmed.

`foreach` *lname* `of newlist` *list* `{`...`}` indicates that the *list* is to be interpreted as new variable names; see **[U] 11.4.2 Lists of new variables**. A check is performed to see that the named variables could be created, but they are not automatically created.

`foreach` *lname* `of numlist` *list* `{`...`}` indicates a number list and allows standard number-list notation; see **[U] 11.1.8 numlist**.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

`foreach` has many forms, but it is just one command, and what it means is

```
foreach value of a list of things, set x equal to each and {
        execute these instructions once per value
        and in the loop we can refer to `x' to refer to the value
}
```

and this is coded

```
foreach x ... {
        ... `x' ...
}
```

We use the name `x` for illustration; you may use whatever name you like. The list itself can come from a variety of places and can be given a variety of interpretations, but `foreach x in` is easiest to understand:

```
foreach x in a b mpg 2 3 2.2 {
        ... `x' ...
}
```

The list is `a`, `b`, `mpg`, `2`, `3`, and `2.2`, and appears right in the command. In some programming instances, you might know the list ahead of time, but often what you know is that you want to do the loop for each value of the list contained in a macro, for instance, ‘`varlist`’. Then you could code

```
foreach x in ‘varlist’ {
        ... ‘x’ ...
}
```

but your code will execute more quickly if you code

```
foreach x of local varlist {
        ... ‘x’ ...
}
```

Both work, but the second is quicker to execute. In the first, Stata has to expand the macro and substitute it into the command line, whereupon `foreach` must then pull back the elements one at a time and store them. In the second, all of that is already done, and `foreach` can just grab the local macro `varlist`.

The two forms we have just shown,

```
foreach x in ... {
        ... ‘x’ ...
}
```

and

```
foreach x of local ... {
        ... ‘x’ ...
}
```

are the two ways `foreach` is most commonly used. The other forms are for special occasions.

In the event that you have something that you want to be given the interpretation of a varlist, newvarlist, or numlist before it is interpreted as a list, you can code

```
foreach x of varlist mpg weight-turn g* {
        ... ‘x’ ...
}
```

or

```
foreach x of newlist id values1-values9 {
        ... ‘x’ ...
}
```

or

```
foreach x of numlist 1/3 5 6/10 {
        ... ‘x’ ...
}
```

Just as with `foreach x in ...`, you put the list right on the command line, and, if you have the list in a macro, you can put ‘*macroname*’ on the command line.

If you have the list in a macro, you have no alternative but to code ‘*macroname*’; there is no special `foreach x of local` *macroname* variant for varlist, newvarlist, and numlist because, in those cases, `foreach x of local` *macroname* itself is probably sufficient. If you have the list in a macro, then how did it get there? Well, it probably was something that the user typed and that your program has already parsed. Then the list has already been expanded, and treating the list as a general list is adequate; it need not be given the special interpretation again, at least as far as `foreach` is concerned.

▷ Example 1: Using foreach, interactively

`foreach` is generally used in programs, but it may be used interactively, and for illustration we will use it that way. Three files are appended to the dataset in memory. The dataset currently in memory and each of the three files has only one string observation.

```
. list

                          x
  1.         data in memory

. foreach file in this.dta that.dta theother.dta {
  2.         append using "‘file’"
  3. }

. list

                          x
  1.         data in memory
  2.     data from this.dta
  3.     data from that.dta
  4. data from theother.dta
```

Quotes may be used to allow elements with blanks.

```
. foreach name in "Annette Fett" "Ashley Poole" "Marsha Martinez" {
  2.         display length("‘name’") " characters long -- ‘name’"
  3. }
12 characters long -- Annette Fett
12 characters long -- Ashley Poole
15 characters long -- Marsha Martinez
```

◁

foreach ... of local and foreach ... of global

`foreach` *lname* `of local` *lmacname* obtains the blank-separated list (which may contain quotes) from local macro *lmacname*. For example,

```
foreach file of local flist {
        ...
}
```

produces the same results as typing

```
foreach file in ‘flist’ {
        ...
}
```

except that `foreach file of local flist` is faster, uses less memory, and allows the list to be modified in the body of the loop.

If the contents of `flist` are modified in the body of `foreach file in ‘flist’`, `foreach` will not notice, and the original list will be used. The contents of `flist` may, however, be modified in `foreach file of local flist`, but only to add new elements onto the end.

`foreach` *lname* `of global` *gmacname* is the same as `foreach` *lname* `in $`*gmacname*, with the same three caveats as to speed, memory use, and modification in the loop body.

▷ Example 2: Looping over the elements of local and global macros

```
. local grains "rice wheat flax"

. foreach x of local grains {
  2.         display "`x'"
  3. }
rice
wheat
flax

. global money "Dollar Lira Pound"

. foreach y of global money {
  2.         display "`y'"
  3. }
Dollar
Lira
Pound
```

◁

foreach ... of varlist

`foreach` *lname* `of varlist` *varlist* allows specifying an existing variable list.

▷ Example 3: Looping over existing variables

```
. foreach var of varlist pri-rep t* {
  2.         quietly summarize `var'
  3.         summarize `var' if `var' > r(mean)
  4. }
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       price |        22    9814.364    3022.929      6229      15906
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         mpg |        31    26.67742    4.628802        22         41
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       rep78 |        29     4.37931     .493804         4          5
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       trunk |        40        17.1    2.351214        14         23
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        turn |        41    43.07317    2.412367        40         51
```

◁

`foreach` *lname* `of varlist` *varlist* can be useful interactively but is rarely used in programming contexts. You can code

```
syntax [varlist] ...
foreach var of varlist `varlist' {
        ...
}
```

but that is not as efficient as coding

```
syntax [varlist] ...
foreach var of local varlist {
        ...
}
```

because ‘varlist’ has already been expanded by the syntax command according to the macro rules.

❑ Technical note

```
syntax [varlist] ...
foreach var of local varlist {
        ...
}
```

is also preferable to

```
syntax [varlist] ...
tokenize ‘varlist’
while "‘1’" != "" {
        ...
        macro shift
}
```

or

```
syntax [varlist] ...
tokenize ‘varlist’
local i = 1
while "‘‘i’’" != "" {
        ...
        local i = ‘i’ + 1
}
```

because it is not only more readable but also faster.

❑

foreach ... of newlist

newlist signifies to foreach that the list is composed of new variables. foreach verifies that the list contains valid new variable names, but it does not create the variables. For instance,

```
. foreach var of newlist z1-z4 {
  2.         gen ‘var’ = runiform()
  3. }
```

would create variables z1, z2, z3, and z4.

foreach ... of numlist

foreach *lname* of numlist *numlist* provides a method of looping through a list of numbers. Standard number-list notation is allowed; see [U] **11.1.8 numlist**. For instance,

```
. foreach num of numlist 1/4 8 103 {
  2.         display ‘num’
  3. }
1
2
3
4
8
103
```

If you wish to loop over many equally spaced values, do not code, for instance,

```
foreach x in 1/1000 {
        ...
}
```

Instead, code

```
forvalues x = 1/1000 {
        ...
}
```

`foreach` must store the list of elements, whereas `forvalues` obtains the elements one at a time by calculation; see [P] **forvalues**.

Use of foreach with continue

The *lname* in `foreach` is defined only in the loop body. If you code

```
foreach x ... {
        // loop body, ‘x’ is defined
}
// ‘x’ is now undefined, meaning it contains ""
```

‘`x`’ is defined only within the loop body, which is the case even if you use `continue, break` (see [P] **continue**) to exit the loop early:

```
foreach x ... {
        ...
        if ... {
                continue, break
        }
}
// ‘x’ is still undefined, even if continue, break is executed
```

If you later need the value of ‘`x`’, code

```
foreach x ... {
        ...
        if ... {
                local lastx ‘"‘x’"’
                continue, break
        }
}
// ‘lastx’ defined
```

The unprocessed list elements

The macro ‘`ferest()`’ may be used in the body of the `foreach` loop to obtain the unprocessed list elements.

▷ Example 4

```
. foreach x in alpha "one two" three four {
  2.         display
  3.         display `"       x is |`x'|"'
  4.         display `"ferest() is |`ferest()'|"'
  5. }

       x is |alpha|
ferest() is |"one two" three four|

       x is |one two|
ferest() is |three four|

       x is |three|
ferest() is |four|

       x is |four|
ferest() is ||
```

◁

`ferest()` is available only within the body of the loop; outside that, `ferest()` evaluates to "". Thus you might code

```
foreach x ... {
        ...
        if ... {
                local lastx `"`x'"'
                local rest `"`ferest()'"'
                continue, break
        }
}
// `lastx' and `rest' are defined
```

Also see

[P] **continue** — Break out of loops

[P] **forvalues** — Loop over consecutive values

[P] **if** — if programming command

[P] **levelsof** — Levels of variable

[P] **while** — Looping

[U] **18 Programming Stata**

[U] **18.3 Macros**