

# The old-style 'for' command

The old-style 'for' command is not documented in the current STATA help files or manual, but it provides really useful features that are not readily duplicated by the new-style facilities. Full documentation (from an older version of the STATA manual) follows this introduction.

The new facilities (foreach, forvalues, while) are far more elaborate and provide basic programming features, but also quite hard to use. If you are not ambitious to become a programmer I recommend that you stick to old-style facilities, which are simple and powerful. These are defined in the following pages in an extract from a long-superseded edition of the STATA manual.

At the end of the STATA documentation that follows, you will find some examples of applications particularly relevant to survey data users. These are set out in do-file format from which they can be copied and pasted into your own do-file as a basis to work on.

## Title

**for** — Repeat Stata command

## Syntax

Basic syntax

```
for listtype list : stata_cmd_containing_X
```

Full syntax

```
for [id in] listtype list [\ [id in] listtype list [\ ...]] [, dryrun noheader  
pause nostop] : stata_cmd [\ stata_cmd [\ ...]]
```

where *listtype* and *list* are

if <i>listtype</i> is	then	<i>list</i> is a
<u>varlist</u>		<u>varlist</u>
<u>newlist</u>		<u>new_varlist</u>
<u>numlist</u>		<u>numlist</u>
<u>anylist</u>		<u>list of words</u>

If *ids* are not specified, then

Elements of the	1st	list are used to substitute for	X	in <i>stata_cmd</i>
Elements of the	2nd	list are used to substitute for	Y	in <i>stata_cmd</i>
...	3rd	...	Z	...
...	4th	...	A	...
...	5th	...	B	...
...	6th	...	C	...
...	7th	...	D	...
...	8th	...	E	...
Elements of the	9th	list are used to substitute for	F	in <i>stata_cmd</i>

## Description

**for** repeats *stata\_cmd*. At each repetition, the members of *list* are substituted for the *ids* in *stata\_cmd*. More generally, at each repetition, the members of the first *list* are substituted for all occurrences of the first *id* in the *stata\_cmds*, members of the second *list* are substituted for all occurrences of the second *id* in the *stata\_cmds*, and so forth.

## Options

`dryrun` specifies that `stata_cmd` is not to be executed; `for` is merely to display the commands that would be run had `dryrun` not been specified.

`noheader` suppresses the display of the command before each repetition.

`pause` pauses output after each execution of `stata_cmd`. This may be useful, for example, when `for` is combined with the `graph` command.

`nostop` does not stop the repetitions if one of them results in an error.

## Remarks

### ▷ Example

Let's do some simple examples which demonstrate the use of the four possible *listtypes*: `varlist`, `newlist`, `numlist`, and `anylist`. First, for all variables that begin with the letter `m`, let's replace their values with their values divided by ten.

```
. for var m* : replace X = X/10
-> replace miles = miles/10
(100 real changes made)
-> replace minutes = minutes/10
(100 real changes made)
-> replace marks = marks/10
(100 real changes made)
-> replace myvar = myvar/10
(100 real changes made)
```

A word of caution about using the wildcard character (`*`): the specification might contain more variables than you intend. For example, if the dataset above had included a string variable called `maiden` containing maiden names, then `m*` would have included this variable and `for` would have attempted to divide the names by ten.

Next, we will generate ten new variables named `u1`, `u2`, ..., `u10` filled with uniform random numbers.

```
. for new u1-u10 : gen X = uniform()
-> gen u1 = uniform()
-> gen u2 = uniform()
(output omitted)
-> gen u9 = uniform()
-> gen u10 = uniform()
```

Now, let's count the number of times the values in the variable `freq` equal 1, 2, 3, 4, or 5.

```
. for num 1/5: count if freq==X
-> count if freq==1
14
-> count if freq==2
12
-> count if freq==3
18
```

```
-> count if freq==4
    36
-> count if freq==5
    20
```

Finally, we place a missing value ('.') in the variable *z* whenever the variable *y* is missing or equal to -1.

```
. for any . -1 : replace z = . if y==X
-> replace z = . if y==.
(16 real changes made, 16 to missing)
-> replace z = . if y==-1
(15 real changes made, 15 to missing)
```

If we specify the `noheader` option, the line that indicates which command is being executed would be omitted for each iteration:

```
. for any . -1 , noheader : replace z = . if y==X
(16 real changes made, 16 to missing)
(15 real changes made, 15 to missing)
```

Note that `noheader` is specified as an option to `for`, not as an option to `replace`. Typing

```
for any . -1 : replace z = . if y==X, noheader
```

would be an error. Second, note that `noheader` suppressed the printing of the headers, not the output from `replace`. If we want to suppress all the output, we need to specify `quietly` in front of the entire compound command (see [P] `quietly`); if we specify `quietly`, we do not need to specify `noheader`.

◀

In the examples above, elements from the list are substituted one at a time for the capital *X* and the command executed. *X* is the default *id*—the character marking where substitutions are to be made. Other *ids* can be specified by including *id in*.

## ▷ Example

What if we wished to append several similarly named Stata data files to our dataset? We could use `for` to make the task easier.

```
. for num 1/4 , dryrun : append using "my dir\fileX"
-> append using `"'my dir\file1"'`
-> append using `"'my dir\file2"'`
-> append using `"'my dir\file3"'`
-> append using `"'my dir\file4"'`
```

specifies that `file1`, `file2`, `file3`, and `file4` from the `my dir` directory are to be appended. Notice the use of the `dryrun` option. This indicates that we wish to preview what commands would be executed but not actually execute them. This is especially helpful if you wish to verify that you have specified your `for` command correctly.

We can accomplish the same thing with

```
. for FILENO in num 1/4 , dryrun : append using "my dir\fileFILENO"
(output omitted)
```

or

```
. for fno in num 1/4 , dryrun : append using "my dir\filefno"
(output omitted)
```

Be careful when specifying *ids* because all occurrences of *id* are substituted. The command

```
. for f in num 1/4 , dryrun : append using "my dir\filef"
-> append using `my dir\1ile1'
-> append using `my dir\2ile2'
-> append using `my dir\3ile3'
-> append using `my dir\4ile4'
```

ends up with filenames like *1ile1* instead of *file1*.

Some users prefer special characters for the *ids*.

```
. for @ in num 1/4 , dryrun : append using "my dir\file@"
(output omitted)
```

The choice of *id* is yours.

Note that if you refer to filenames containing backslash (`\`), then you must enclose the filename in double quotes; otherwise, `for` will think that the backslash is a separator for the `for` command.

◀

Multiple *lists* are processed concurrently, in parallel. The default *ids* are X, Y, Z, A, B, C, D, E, and F when you do not specify otherwise.

## ▷ Example

Let's generate four new variables—*x2*, *x3*, *x4*, and *x5*—that correspond to the 2nd, 3rd, 4th, and 5th powers of the variable *myvar*.

```
. for new x2-x5 \ num 2/5 : gen X = myvar^Y
-> gen x2 = myvar^2
-> gen x3 = myvar^3
-> gen x4 = myvar^4
-> gen x5 = myvar^5
```

Or, instead, we could have typed

```
. for VAR in new x2-x5 \ POWER in num 2/5 : gen VAR = myvar^POWER
(output omitted)
```

or even

```
. for num 2/5 : gen xX = myvar^X
(output omitted)
```

to produce the same result.

◀

Multiple commands are allowed with the `for` command. The backslash (`\`) is used to separate the commands.

## ▷ Example

Let's say that we want to perform several regressions and obtain predicted values after each one. With each new regression we add another of the variables we created in our last example. We can perform both the `regress` and the `predict` command in the same `for` command.

```
. for NUM in num 2/5 : quietly reg z m* x2-xNUM \ quietly predict predNUM
-> quietly reg z m* x2-x2
-> quietly predict pred2
-> quietly reg z m* x2-x3
-> quietly predict pred3
-> quietly reg z m* x2-x4
-> quietly predict pred4
-> quietly reg z m* x2-x5
-> quietly predict pred5
```

In fact, if we had not previously generated the `x` variables we could have included that step in the same `for` command.

```
. for num 2/5 : qui gen xX = myvar`X' \ qui reg z m* x2-xX \ qui predict predX
```

## □ Technical Note

If you set `version` to 5.0 or earlier (see [P] `version`), you will execute an older `for` command which has a different syntax. This ensures that old `do`-files, `ado`-files, and programs continue to work. □

## Methods and Formulas

`for` is implemented as an `ado`-file.

## Acknowledgment

We thank Patrick Royston of the MRC Clinical Trials Unit, London for his contributions to this command.

## References

- Cox, N. J. 1998a. ip26: Bivariate results for each pair of variables in a list. *Stata Technical Bulletin* 45: 17–19. Reprinted in *Stata Technical Bulletin Reprints*, vol. 8, pp. 78–81.
- . 1998b. ip27: Results for all possible combinations of arguments. *Stata Technical Bulletin* 45: 20–21. Reprinted in *Stata Technical Bulletin Reprints*, vol. 8, pp. 81–83.
- Royston, P. 1995. ip8: An enhanced `for` command. *Stata Technical Bulletin* 26: 12. Reprinted in *Stata Technical Bulletin Reprints*, vol. 5, p. 65.
- . 1996. ip8.1: An even more enhanced `for` command. *Stata Technical Bulletin* 30: 5–6. Reprinted in *Stata Technical Bulletin Reprints*, vol. 5, pp. 65–66.
- Weesie, J. 1997. ip19: Using expressions in Stata commands. *Stata Technical Bulletin* 39: 20–22. Reprinted in *Stata Technical Bulletin Reprints*, vol. 7, pp. 83–85.
- . 1999. gr36: An extension of `for`, useful for graphics commands. *Stata Technical Bulletin* 49: 8–10. Reprinted in *Stata Technical Bulletin Reprints*, vol. 9, pp. 92–95.

**Also See**

- Complementary:** [P] quietly
- Related:** [R] by,  
[P] foreach, [P] forvalues, [P] while
- Background:** [U] 14.4 varlists

**Useful applications of old-style loops**

Did you ever want to see whether a variable was equal to one of a series of values?

How about:

```
for any 0 8 9 : replace testvar = . if testvar == X
```

This alone can save hours of painful typing !

Or consider:

```
for X in var test1-test55 : quietly for Y in any 0 8 9 : replace X = . if X == Y
```

This would convert a series of SPSS variables with 0,8,9 missing data codes into variables with missing data specified correctly. **NOTE THAT YOU SHOULD CAREFULLY CHECK THAT THESE ARE IN FACT THE MISSING DATA CODES USED FOR EACH VARIABLE CONVERTED IN THIS WAY!** The carefully placed prefix “quietly” suppresses much of the output that could be rather voluminous. Note that with nested “for”s you must explicitly name the stand-in token, since otherwise only tokens from the last loop will be considered, which won’t do it.

Or consider:

```
for new mis1-mis5: gen X=0
for var mis1-mis5 \ var var21-var25 \ gen X = 1 if Y==.
```

This creates a set of missing data indicators that take on the value 1 if a particular variable is missing, and zero otherwise. Here, we separate concurrent variable lists by backslashes. Backslashes can also be used after the colon to separate commands that will be performed in sequence for each of the items specified before the colon, as follows:

```
for num 1/5: gen churchattX = 0 \ replace churchattX=1 if VAR282==X
```

This zeros a set of variables and then replaces one of those zeros with a 1, identifying which of the set of dummies corresponds to the value of VAR282 in each case in the dataset. This is an alternative to the use of “generate” with the “tab1” command, but gives better control over the choice and naming of dummy variables created.