How to face lists with `fortitude`

Nicholas J. Cox
University of Durham, UK
n.j.cox@durham.ac.uk

expanded version forthcoming in
Stata Journal 2(2), 2002

Synopsis

Three commands in official Stata

`foreach`

`forvalues`

and `for`

provide structures for cycling through lists of values — variable names, numbers, arbitrary text — and repeating commands using members of those lists in turn.

All may be used interactively.

The aim here is to explain and compare them, giving a variety of examples.

Why learn these commands?

Have you ever typed a series of very similar Stata commands and wondered whether there was a quicker way?

Working your way through a list is very common in data management and analysis.

We want to do that with speed and system.

Stata has structures for defining problems in which you cycle through lists.

They can be used interactively, as well as in programs.

There is just one hill to climb first, the idea of a local macro.

Local macros

A character string given a special name

A macro in Stata is just a character string given a special name.

```
local rhsvars "trunk weight length
turn displacement"
```

assigns the name `rhsvars` to the character string

```
"trunk weight length turn
displacement"
```

The quotes here, `" "`, delimit the string: i.e. they are not part of the string.

`" "` often are not necessary, but they are recommended for beginners.

(If you did want to include `"` in string, you need so-called compound double quotes.)

The most obvious reason for defining a macro is to refer to it later and save yourself typing.

In this example, we have names of variables from the auto data.

Refer to macro by using single quotes ‘ ’, say

`regress mpg ‘rhsvars’`

N.B. ‘ differs from ’.

How Stata processes macro names: substitution first

We need to understand a little of how Stata interprets a command line.

All macro names are substituted by their contents *before* Stata attempts to execute any command.

Call this the *substitution first* rule.

Whenever you type a command, Stata has two main things to do.

1. To receive your command and to translate into its own terms.
2. To try to execute your command.

1. includes substitution (expansion) of any macro names used.

In this example, Stata substitutes the macro name and now sees

```
regress mpg trunk weight length turn
displacement
```

2. should now be straightforward with auto data in memory.

The main pitfall is getting the local macro name slightly wrong.

◇ Stata is case-sensitive: compare `‘RHSVARS’` and `‘rhsvars’`.

◇ Stata makes no attempt to correct your spelling.

◇ Stata does not allow you to abbreviate macro names.

◇ Referring to a non-existent macro is not, in itself, an error. A macro name which refers to nothing is substituted by nothing, that is, the empty string `""`.

```
regress mpg ‘rhsvar’
```

would be seen by Stata as

```
regress mpg
```

Macros will fade away at the end of a session.

You can re-define macros at will. Suppose you wanted to add the variable name `gear_ratio` to `rhsvars`.

Over-write the old definition:

```
local rhsvars "trunk weight length
turn displacement gear_ratio"
```

or (better) amend the old definition by adding to it:

```
local rhsvars "`rhsvars' gear_ratio"
```

This is another example of the substitution first rule. Stata first sees

```
local rhsvars "trunk weight length
turn displacement gear_ratio"
```

and then executes the command, which happens to re-define the macro.

The second way is better because

◇ Stata will be far faster than you and less error-prone.

◇ It is a natural and helpful way of writing any operation in which we accumulate results step by step.

A key type of example is

```
local results "`results'`new' "
```

What does ‘local’ mean?

Local macros are visible only within the Stata program in which they are defined: within

◇ an interactive session,

◇ a program defined as such,

◇ a `do` file, or

◇ a set of commands in Stata’s `do` file editor.

Your local macro `rhsvars` and some other program’s `rhsvars` are different entities.

Otherwise you would need to look inside every program you used and check for possible incompatibilities!

Global macros

There are macros in Stata which are visible everywhere, irrespective of what program is running. These are called global and defined similarly:

```
global rhsvars "trunk weight length
turn displacement gear_ratio"
```

They are referred to in a slightly different way:

```
regress mpg $rhsvars
```

What we are discussing needs only local macros.

Macros can contain numeric characters

Although we have defined macros as character strings, when those strings in Stata are numeric characters, we can think of such macros as having numeric values.

Macros really are just strings: it is just that the rest of Stata is happy to treat their contents as numeric whenever that makes sense.

For example, given

```
local i "1"
```

Stata sets the local macro `i` to the character `"1"`, which happens to be numeric.

Now suppose we want to increment by 1. Most natural to us, and perfectly acceptable to Stata, is to write this as an evaluation:

```
local i = `i' + 1
```

This is a new syntax compared
with examples so far. Following the
substitution first rule, Stata sees

```
local i = 1 + 1
```

It then evaluates the expression and redefines the macro `i` as its result, namely the number 2, which it treats as the numeric character `"2"`.

The evaluation is nothing to do with the macro: it is part of the rest of Stata. That from context takes you to want `+` to be addition.

Within a different example

```
local i = "1" + "1"
```

the `" "` insist to Stata that the macros are to be treated as strings. From context `+` is taken to mean concatenation.
`i` will now contain `"11"`.

```
local i "`i' + 1"
```

is different again. Given `i` of `"1"`, `i` becomes `"1 + 1"`. Without the equals sign, no evaluation will take place.

Similarly in

```
local i = "`i' + 1"
```

`+` is treated as just another character, not an operator. Evaluation makes no difference to the contents of the string.

`foreach`, `forvalues`, `for`

The Stata commands `foreach`, `forvalues` and `for` make up a trio.

`for` was introduced in Stata 3.1 (1993), with redesigns in 5.0 (1997) and 6.0 (1999).

Begin with `foreach` and `forvalues`, introduced in 7.0 (2001).

`foreach`: **first syntax**

Suppose that we wish to `generate` a series of powers of a variable. The slow but sure way

```
gen y_2 = y^2
gen y_3 = y^3
gen y_4 = y^4
```

cries out for a simpler structure.

```
foreach i in 2 3 4 {
          gen y_`i' = y^`i'
}
```

This is an example of a first syntax with `foreach`.

`foreach` *macro* `in` *list_of_values* `{`

one or more statements defined
in terms of that macro name

`}`

The structure encapsulates several features:

◇ A macro name must be given in the first part.

◇ A list must be specified immediately after. The keyword `in` specifies that you are going to spell out all the *in*dividual elements of the list. (!!!)

◇ One or more statements, at least one of which refers to the macro name, must be given within braces. Spacing is at choice, so long as statements are on separate lines.

◇ The structure automatically defines the macro in turn as each member of the list and then substitutes the contents in the commands within braces.

◇ The controlling macro disappears at the end of the structure.

Another problem is producing various transformations of several variables.

```
foreach x in list_of_variables {
        gen log`x' = log(`x')
        gen sqrt`x' = sqrt(`x')
        gen rec`x' = 1 / `x'
}
```

However, Stata allows several ways of giving abbreviated variable lists. Such features can be exploited within `foreach` by using the second syntax.

`foreach`: second syntax

`foreach` *macro* `of` *listtype list_of_values* `{`
            one or more statements defined
            in terms of that macro name
`}`

The keyword `of` specifies that you are going to give a list *of* the type to be named. (!!!)

In last example, *listtype* is `varlist`:

```
foreach x of varlist * {
          cap gen log`x' = log(`x')
          cap gen sqrt`x' = sqrt(`x')
          cap gen rec`x' = 1 / `x'
}
```

`capture` (abbreviation `cap`) is a device to catch the occasions when a command will not work: digest the output (including the error) and carry on regardless.

Why might `generate` commands not work?
◇ Any attempt to transform string variables will fail.
◇ Variable names near the 32 character limit are problematic.

We might want to carry on regardless, but still get an informative message:

```
foreach x of varlist * {
          cap gen log`x' = log(`x')
          if _rc { di "`x': " _rc }
}
```

`capture` puts return code in `_rc`.
Here we use the command `if` and the fact that `if _rc` is equivalent to `if _rc ~= 0`.

`foreach` allows other types of list: within a `local` or a `global`, a `newlist` (list of new variable names) or a `numlist` (a list of numbers).

An earlier example could be written in terms of a `numlist`:

```
foreach i of num 2/4 {
        gen y_`i' = y^`i'
}
```

Three-letter abbreviations (TLAs) like `num` are permissible for *listtype*.

The second syntax is much more powerful and more useful than the first.

One very common application of `foreach` is producing univariate results for each of several variables.

`foreach` can be used as a wrapper to cycle through a *varlist* whenever only a single *varname* is acceptable.

For example, take normal probability plots:

```
foreach x of var varlist {
        qnorm `x'
        more
}
```

`more` ensures that each graph remains visible.

Do not confuse the two `foreach` syntaxes

The `in` and `of` syntaxes are distinct and should not be confused. In particular, it is legal in Stata to have a structure which begins something like

```
foreach q in numlist 1/3 {
```

Any kind of list may follow `in`, so Stata will not pick up that this is almost certainly an attempt at

```
foreach q of numlist 1/3 {
```

`forvalues`

`forvalues` is complementary to `foreach`. It can be thought of as an important special case of `foreach`, for cycling through certain types of *numlist*, but presented a little more directly.

`forval` *macro* `=` *range_of_values* `{`
one or more statements defined in terms of that macro name
`}`

Here *range_of_values* specifies a sequence of numbers and takes one of two main forms, exemplified by `1/9` and `10(10)80`.

`1/9` yields `1 2 3 4 5 6 7 8 9`.

`10(10)80` yields `10 20 30 40 50 60 70 80`.

For decreasing sequences, use a form like `25(-1)1`.

So for cycling over simple integer sequences, `forvalues` is an alternative to `foreach`. `foreach` must store the integers, giving `forvalues` an edge in speed.

For generating powers of a variable:

```
forval i = 2/4 {
        gen y_`i' = y^`i'
}
```

One key issue in assessing `qnorm` plots is how much variability would be expected even if parent distribution were normal. Suppose a normal plot was saved as a `gph` file by

`qnorm` *ourvar*`, saving(`*ourvar*`)`

We should compare this with a reference portfolio of plots for normal samples of the same size.

Say we create 24 random samples and their normal probability plots:

```
forval i = 1/24 {
        gen v`i' = invnorm(uniform())
        qnorm v`i', saving(v`i')
        local G "`G'v`i'"
}
```

As `i` goes from 1 to 24, at each step we get a new sample from $\mathrm{N}(0, 1)$.
We then use `qnorm` to draw a normal probability plot, and save graph image.

We also accumulate names of variables (and thus `gph` files) in local macro `G`. Then we can redraw the saved graphs:

```
graph using ourvar `G'
```

$1 + 24$ graphs will plot nicely as a 5×5 array.

This is not the only way to do it. You might prefer to use `foreach`:

```
foreach v of new v1-v24 {
          gen `v' = invnorm(uniform())
          qnorm `v', saving(`v')
          local G "`G'`v' "
}
```

Initialising before `foreach` or `forvalues`

In many problems, we need one step more: to initialise one or more things before we enter the loop.

For example, we might want to create a new variable, but the recipe for creation is too complicated for a single command.

Or we might want to populate a matrix with entries from separate calculations.

Initialising a variable

Such tasks often arise in cleaning up fairly large data sets containing string variables, say names of countries or companies or diseases.

Imagine a string variable indicating vacation destinations. Initial inspection reveals many near synonyms for Britain: Britain, Great Britain, UK, United Kingdom, and so forth. We decide to combine these all into Britain.

Constraint: we can only use `generate` once; thereafter changes must be through `replace`. It is often advisable therefore to put a `generate` statement outside the loop.

Simplifying our example a bit, we have code like

```
generate str1 Dest = ""
replace Dest = dest
foreach c in "Great Britain" "UK" {
    replace Dest =
        subinstr(Dest,"`c'","Britain",1)

}
```

(One linebreak here is made necessary by the font size chosen.)

Note also how strings with embedded spaces such as `"Great Britain"` need delimiting quotes.

Populating a matrix

Many bivariate commands produce single-number statistics which we might want to output as a two-way table. Many commands are not set up to do this automatically. `foreach` makes it possible to overcome that.

`ktau` takes a pair of variables *varname1* and *varname2*, and calculates Kendall's tau (here we focus on τ_a).

To grind through all the possibilities, we need two `foreach` loops, one nested inside the other. Here is our first stab, for an example with 10 variables:

```
foreach v of var price-gear {
        foreach w of var price-gear {
                ktau `v' `w'
        }
}
```

The nested loops look like a new idea, but the new idea is only a little one. Follow Stata as it goes through the structure.

The crucial rule is that the innermost loop is completed first.

For p variables in each list, all p^2 correlations are calculated. We made Stata calculate both `ktau` x y and `ktau` y x. This is wasteful by a factor of ~ 2, but the extra time is usually less than it would take to modify the code.

A more important detail is treatment of missing values. Say you prefer the same observations to be used throughout.
Use `egen` to count missing values across variables:

```
egen nmiss = rmiss(price-gear_ratio)
```

and then stipulate that τ_a is calculated `if nmiss == 0`.

To get results in a table, populate a matrix with correlations, and then `matrix list` takes care of displays.

Set up, before the `foreach` loops, a matrix of the right size:

```
matrix tau = J(10,10,10)
```

Here 10 as an impossible result for τ_a gives a check that code cycles through all the possibilities intended.

We need to pick up each result after `ktau`: the manual documents results temporarily accessible after a command is executed. In our case, we need `r(tau_a)`.

We need to cycle through rows and columns: initialise indexes `i` and `j` to 0 just before the corresponding `foreach` loop, and then increment by 1 every time we set `v` or `w`.

Putting the code together:

```
egen nmiss = rmiss(price-gear)
matrix tau = J(10,10,10)
local i = 0
foreach v of var price-gear {
     local i = `i' + 1
     local j = 0
     foreach w of var price-gear {
          local j = `j' + 1
          ktau `v' `w' if nmiss == 0
          mat tau[`i',`j'] = r(tau_a)
     }
}
matrix list tau, format(%4.3f)
```

We need intelligible labelling of rows and columns: `unab` 'unabbreviates' a variable list into a local macro.

```
unab vars : price-gear
matrix rownames tau = `vars'
matrix colnames tau = `vars'
matrix list tau, format(%4.3f)
```

`for`

`for` is the third of our commands for cycling through lists. While very useful for easy tasks, `for` can become awkward for slightly more difficult tasks.

Revisit some of our examples and see how they would be done with `for`:

Powers of a variable:

```
for any 2 3 4 : gen y_X = y^X
```

or (better)

```
for num 2/4 : gen y_X = y^X
```

Transformations of a variable:

for var *varlist* : gen logX = log(X) \
gen sqrtX = sqrt(X) \ gen recX = 1 / X

Normal probability plots:

for var *varlist* : qnorm X \ more

or

for var *varlist*, pause : qnorm X

The pattern underlying these examples is, with more details to come,

`for` *listtype list_of_values* `:` *one or more commands separated by backslashes*

Looking at examples shows both similarities and differences compared with `foreach` and `forvalues`.

◇ `for` has a notion of *listtype*, just like `foreach`: possible types are `varlist`, `newlist`, `numlist` and `anylist`. Specify arbitrary lists as type `anylist`. TLAs of each *listtype* are allowed.

◇ `for` does not use local macros: you indicate by a placeholder (by default `X`) where each member of the list belongs.

◇ `for` does not use braces or allow separate lines in specifying commands to be executed. Commands follow a colon. Multiple commands are separated by backslashes.

◇ `for` has a few special options of its own. There are further options and features, especially the use of multiple lists, processed in parallel.

`for` divides experienced users into ‘`for`’ and ‘against’ factions. Many users appreciate its conciseness, but code can be difficult to read, and thus difficult to understand and above all to debug.

One key difference: structures using `foreach` and `forvalues` are best thought of as a series of commands under the control of a specified loop. Any local macros will be substituted just before each command is executed. Therefore, their contents may vary through the loop.

In contrast, `for` is best thought of as a single command at the time it is issued, no matter how many command lines follow the colon. Thus any local macros will be substituted just once, immediately before Stata tries to execute the `for` command as a whole. Attempting to try to manipulate local macros within `for` is thus almost always problematic.

It can be difficult or even impossible to nest `for` commands.

The placeholder (by default `X`) might occur as part of the command line. When this happens, or if you want to do it anyway, you need to specify another placeholder, as in

```
for P in num 2/4 : gen y_P = y^P
```

or

```
for POWER in num 2/4 : gen y_POWER =
y^POWER
```

A placeholder need not be a single symbol.

The `in` syntax here is the reverse of that of `foreach`. (!!!)

You will need to protect intended backslashes by inclusion within quotes.

`for` is implemented as a Stata program defined in an `ado` file. `foreach` and `forvalues` are implemented in C code as part of the Stata executable. The overhead of interpretation means that `for` will be much slower than `foreach` or `forvalues`.

Executive summary

◇ The strangest feature of local macros is their name. Think of them as strings given names which are always substituted in a command line by their contents before Stata tries to execute the command.

◇ If you know `for` and like it, then stick with it. But remember that when the going gets tough, the tough get going.

◇ Check out `foreach` and `forvalues` and add them to your repertoire. They grow easily as you attempt more difficult problems.

"The Answer to the Great Question Of Life, the Universe, and Everything" said Deep Thought, "is..."